Efficient Concurrency Control for Broadcast Environments

Jayavel Shanmugasundaram[*]
jai@cs.wisc.edu

Arvind Nithrakashyap[†]
anithrak@us.oracle.com

Rajendran Sivasankaran
sivasank@cs.umass.edu

Krithi Ramamritham
krithi@cs.umass.edu

Department of Computer Science
University of Massachusetts
Amherst, MA 01002

Abstract

A crucial limitation in environments where data is broadcast to clients is the low bandwidth available for clients to communicate with servers. Advanced applications in such environments do need concurrency control support. However, given the asymmetric communication capabilities and the needs of clients in mobile environments, traditional correctness requirements and mechanisms to achieve them are too restrictive, unnecessary, and impractical in such environments. Hence weaker alternatives that are sufficient and practical for such environments are needed. We propose a correctness criterion that allows read-only transactions to read *current* and *consistent* data "off the air", i.e., without contacting the server, say, to obtain locks. Update transactions, however, need to validate their updates at the server. Even though the proposed correctness criterion is weaker than serializability, it is shown that determining legality of histories with respect to this consistency requirement is NP-Complete. So, *APPROX*, a polynomial time algorithm, to efficiently detect (a subset of) legal histories is outlined. *F-Matrix*, a protocol to implement *APPROX* in broadcast disk environments, is then proposed. Our protocol requires additional control information to be broadcast with the data in order to ensure correctness and so a key issue is how to minimize the overhead entailed by this additional information. To this end, we propose *R-Matrix*, a simpler version of *F-Matrix*, to implement *APPROX*. Experimental results confirm the hypothesis that the weaker correctness criterion leads to substantially lower response times compared to using serializability, as exemplified by the concurrency control algorithm used in the *Datacycle* architecture. Furthermore, even though *F-Matrix* and *R-Matrix* show comparable performance in many situations, overall, *F-Matrix* leads to better response times and is also more scalable than *R-Matrix* and *Datacycle*.

---

[*]Currently at the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706
[†]Currently at Oracle Corporation, 500 Oracle Parkway 4op1385a, Redwood Shores, CA 94065

1 Introduction

Many emerging database applications, especially those with numerous concurrent clients, demand the broadcast mode for data dissemination. For example, in electronic commerce applications, such as auctions, it is expected that a typical auction might bring together millions of interested parties even though only a small fraction may actually offer bids. Updates based on the bids made must be disseminated promptly and consistently. Fortunately, the relatively small size of the database, i.e., the current state of the auction, makes broadcasting feasible. But, the communication bandwidth available for a client to communicate with servers is likely to be quite restricted. Thus, using the broadcast medium to transmit the current state of the auction while allowing for clients to communicate their updates to the current state of the auctions using low bandwidth uplinks with the servers is an attractive approach. The problem addressed in this paper and techniques outlined are motivated by such applications. In particular, we are concerned with the problem of providing readers with *current* and *consistent* data while ensuring the consistency of updates.

Broadcast-based data dissemination is also likely to be a major mode of information transfer in mobile computing and wireless environments [4, 13, 26]. Many such systems have been proposed [20] and [16] and commercial systems such as Vitria [21] already implement broadcasting. As these systems continue to evolve, they will be used to run sophisticated applications, many of which will involve data whose consistency must be maintained in spite of updates, some of which may originate from mobile clients. Other applications of broadcasting, include stock trading, next generation road traffic management systems and automated industrial plants [19, 24]. Given the limited amount of bandwidth available for clients to communicate with the broadcast server in such environments, achieving data consistency efficiently in such an environment is a challenging research issue.

[12] and [2] are among the few papers motivated by similar considerations. Herman et. al. [12] discuss transactional support in the Datacycle architecture, which is also an asymmetric bandwidth environment. However, they use serializability as the correctness criterion, which we show is very expensive to achieve in such environments. In [2], the authors discuss the tradeoffs between currency of data and performance issues when some of the broadcast data items are updated by processes running on the server. However, the updates do not have transactional semantics associated with them either at the server or at the clients. The updates are made only by processes running on the server, while the processes on clients are assumed to be read-only.

In this paper, we propose a correctness criterion for transaction processing in broadcast environments where the updates can also originate from the clients but these are deferred until the server validates the updates and commits them. We present concrete concurrency control protocols for realizing this correctness criterion in the context of broadcast disks [26]. These protocols permit read-only transactions running on mobile clients to always *read current and consistent values without contacting the server* (to acquire locks or to validate their reads), i.e., they are able to read the values "off-the-air". In each cycle of the broadcast the server sends the latest committed value of data items, thus satisfying the requirement of sending current data to clients, along with some control information. Client reads can proceed after consulting the control information. One of the issues concerns the nature of control information sent along with the data such that a client's read transaction, which can span multiple cycles, can be sure that the information it has read is consistent. Our main contributions are:

- motivating the need for and developing consistency requirements that (a) are weaker than serializability, and (b) are applicable in broadcast environments.
- showing that determining whether a history is legal with respect to the weaker consistency requirements is NP-complete.
- developing a polynomial time algorithm, namely *APPROX*, that can check for a subset of the legal histories.
- proposing a practical realization of *APPROX* called *F-Matrix*, along with a lower overhead version of *F-Matrix* called *R-Matrix*.
- evaluating the performance of *F-Matrix* and *R-Matrix* and comparing them with the performance of the algorithm used in *Datacycle* and an ideal (but non-realizable) version of *F-Matrix*. The results confirm the hypothesis that a weaker consistency requirement leads to substantially better read response times.

In an earlier work [19], we had performed a preliminary study of correctness criteria appropriate for broadcast environments. This paper builds upon the earlier work by formulating a specific correctness criterion, providing a formal analysis of the correctness criterion, proposing concrete mechanisms to enforce the correctness criterion and evaluating these mechanisms.

The rest of the paper is organized as follows. In section 2, we outline why existing client server concurrency control techniques are inappropriate for broadcast environment and motivate, with examples, the need for a correctness criterion weaker than serializability. Also, we formally characterize the correctness criterion and discuss its applicability to broadcast based databases. In section 3, we show that checking for all legal histories that satisfy the correctness criterion is NP-complete and present the polynomial time approximation algorithm *APPROX*. We also outline the mechanisms required to implement the algorithm in broadcast disk environments. In section 4, we evaluate the performance of these mechanisms and in Section 5, we summarize the paper and outline future work.

2 Correctness of Transactions in Broadcast Environments

In this section, we first outline why existing concurrency control techniques for client server systems are not appropriate for broadcast environments. We then motivate the need for relaxing serializability as a correctness criterion in broadcast environments and propose a correctness criterion that is more appropriate in such environments. Finally, we syntactically characterize the set of histories satisfying this correctness criterion and study its relationship to other existing correctness criteria.

2.1 Inapplicability of Existing Client Server Concurrency Control Techniques

There has been some work dealing with concurrency control in client server architectures [7, 22, 23, 9]. All of these techniques are based primarily on caching the data items read/written by client transactions. The main characteristics of these solutions that make them inappropriate for broadcast environments are: (a) the server has to know the contents of the caches of clients (b) pessimistic protocols use locking by client transactions for concurrency control. Characteristic (a) makes the protocols unviable in broadcast environments because the server has to keep track of a the cache contents of a large number of clients. More importantly, the clients have to inform the server whenever a new or more recent data item is read off the broadcast disk onto the client cache. This is likely to lead to expensive client to server communication and swamp the server even for hot read-only data items having currency requirements. Characteristic (b) makes the protocols infeasible in broadcast environments because the server is likely to get swamped with lock requests from a large number of clients. Also, precious client server bandwidth would be wasted for acquiring locks even for read-only transactions.

There have been some multiversion concurrency control techniques proposed in the context of client server systems [11] that allow read-only transactions to read older versions of data. Apart from the fact that the currency requirement is not adequately met by using old version of data, the server has to keep track of the cache contents (slices) at the clients. This is unacceptable in broadcast environments, as detailed above. To the best of our knowledge, the Datacycle approach [12] is the only concurrency control technique proposed in the literature in the context of broadcast environments. We show that our approach has better performance than that of the Datacycle approach. This is primarily due to the use of a correctness requirement which is weaker than serializability (that nevertheless still preserves consistency) and is appropriate for broadcast environments.

2.2 Motivation for Weakening Serializability

Serializability is intrinsically a global property - the effect of concurrent transaction execution should be as though *all* the transactions executed in some serial order. This global property of serializability (i.e., involving all transactions) is particularly difficult to achieve in broadcast environments, where transactions may be executed at distributed clients and servers. This is because ensuring a global property like serializability in a distributed environment either (a) requires excessive communication between the distributed entities or (b) requires the protocol that ensures the global property to be overly conservative thus disallowing certain correct executions. The first alternative is expensive in broadcast environments because of the limited bandwidth that clients have available to communicate with the server. The second alternative leads to unnecessary transaction aborts, which is again undesirable. The following example illustrates the basic problem.

Example 1. Assume that in a broadcast environment, clients only know the local transaction execution history and history of updates at the server. Consider two stock trading transactions t_1 and t_3 at two different clients A and B respectively that read the stock prices of IBM and Sun. Also let t_2 and t_4 be transactions at the server that update the prices of IBM and Sun respectively. Now consider the following execution history:

$$r_1(IBM)\ w_2(IBM)\ c_2\ r_3(IBM)\ r_3(Sun)\ w_4(Sun)\ c_4\ r_1(Sun)\ ...\ (1.1)$$

If transactions running on clients do not inform the server about the operations performed by them – a reasonable assumption given the limited uplink bandwidth from clients to the server – then the server would only be aware of the history of its own operations:

$$w_2(IBM)\ c_2\ w_4(Sun)\ c_4$$

If a server broadcasts this history along with the data, client A would be aware of the history:

$r_1(IBM)\ w_2(IBM)\ c_2\ w_4(Sun)\ c_4\ r_1(Sun)$

and Client B would be aware of the history:

$w_2(IBM)\ c_2\ r_3(IBM)\ r_3(Sun)\ w_4(Sun)\ c_4$

If both t_1 and t_3 commit, then the server and both the clients would see serializable histories. However, the global history would not be serializable. Thus, either t_1 or t_3 would have to be aborted. However, since the read operations performed by a client transaction are not communicated to other clients or the server, and assuming that there exists no way to inform clients t_1 and t_3 except by expensive message passing, *both* t_1 and t_3 would have to be aborted – since each client must assume the worst-case history at the other. This is wasteful since the abortion of either t_1 or t_3 would have ensured a serializable history. Unnecessary aborts would occur even if the system history is

$r_1(IBM)\ w_2(IBM)\ c_2\ w_4(Sun)\ c_4\ r_1(Sun)$ ... (1.2)

because Client A would be aware of this history and would not be able to distinguish it from the previous case. In this case too, t_1 would have to be aborted. A similar argument can be made for t_3. Essentially, in the absence of communication from read-only transactions, to preserve serializability, the read-only transactions will have to be aborted even in cases like history 1.2 because other clients would have to assume worst case scenarios as in history 1.1.

We can thus see that serializability in broadcast environments is either very expensive to achieve or would involve unnecessary aborts. So, we propose a non-global correctness criterion that (a) ensures consistent reads by clients and (b) preserves the consistency of updates and the database.

2.3 The Correctness Criterion

Though weaker than serializability, the proposed criterion maintains the consistency of the database and of the values read by the transaction. We first give the intuition behind the correctness criterion via examples before formally defining it.

2.3.1 Intuition Behind the Correctness Criterion

The following example shows how serializability can be weakened by allowing different read transactions to see different serial orders of execution of update transactions.

Example 2. Consider the following history of transaction executions for transactions t_1, t_2, t_3 and t_4, where t_1 and t_3 are transactions which perform only read operations in the history. Data items IBM and Sun are accessed by these transactions. (Note that this example is the same as example 1 with the exception of commit c_1 of t_1.)

$r_1(IBM)\ w_2(IBM)\ c_2\ r_3(IBM)\ r_3(Sun)\ c_3\ w_4(Sun)\ c_4\ r_1(Sun)\ c_1$

The serialization graph for this history is given in Figure 1(a). This history is not serializable since there is a cycle in the serialization graph. However, the final state of the database after this execution is exactly the same as that of the serial execution of $t_2; t_4$, the two transactions which perform updates in the history. That is, the final state of the database is consistent.If the history is considered up to time c_3, the serialization graph *based on the history known to* t_3 is depicted in Figure 1(b). At this point, t_3 can be committed without any inconsistency, since t_3 views a consistent database (i.e., it views the transactional update to IBM's stock) and the database state at this point is also consistent (i.e., it represents the state reached by a serializable execution of the transaction that updates IBM's stock). Further, at c_1, a serialization graph can be formed as in Figure 1(c) with t_1 and all the committed update transactions. The operations of read transaction t_3 are not in the history when the serialization graph is constructed by t_1 since t_1 is not made aware of the operations of read-only transaction t_3. Therefore, in Figure 1(c), t_1 views a serializable history with the history formed by the operations from update transactions and itself. Again, at this point, the state of the database is consistent.

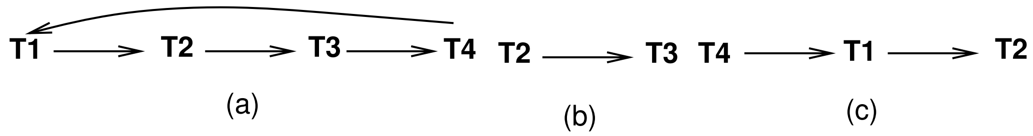


Figure 1: Serialization Graphs

In the above example, although the global history is not serializable, each read transaction reads consistent values and views some serial order of the update transactions while all update transactions are serializable. The crux of this weakening of serializability is that each read transaction can view a different serial execution order of the update transactions and itself. Thus, in the example, t_1 and t_3 view different orders of updates to the stock prices of IBM and Sun. This, however, does not cause any inconsistency because the updates to the stock prices of IBM and Sun were not meant to be atomic, i.e., were not performed as part of the same update transaction. On the other hand, if the updates were performed by the same transaction, then this would be unacceptable because the mutual consistency of the two updates has to be preserved. Due to this reason, the update transactions have to be globally serializable.

The following example shows how serializability can be further weakened so that read-only transactions need not be serializable with respect to all the update transactions, but only with respect to a select subset of update transactions.

Example 3: Consider the following history that is a modification of the history used in Example 2 (it has the additional operation $w_1(DEC)$, so that t_1 is not read-only any more).

$$r_1(IBM)\ w_2(IBM)\ c_2\ r_3(IBM)\ r_3(Sun)\ c_3\ w_4(Sun)\ c_4\ r_1(Sun)\ w_1(DEC)\ c_1$$

The above history is not correct even given the weakened serializability notion introduced above. This is because the read-only transaction t_3 is not serializable with respect to *all* the update transactions, i.e., t_1, t_2, and t_4 (given the serialization graph shown in Figure 1(a)). However, the read-only transaction t_3 does read consistent values (the transactional update to IBM's stock) because it sees the effects of only the update transaction t_2 on the initial state of the database. Further, all the update transactions are serializable in the order t_3;t_4;t_1.

From the above example, we can see that in order to maintain consistency, it is sufficient if read-only transactions are serializable with respect to only the update transactions they *directly or indirectly read from* (provided that all update transactions are serializable). This constitutes the second weakening of serializability that still preserves correctness.

In summary, the proposed correctness criterion weakens serializability in two important ways:

1. Different read-only transactions can see the effects of update transactions in different serial orders.
2. Each read-only transaction need not be serialized with respect to all the update transactions but only with respect to the set of update transactions it directly or indirectly read from. Update transactions, however, have to be serializable in the traditional sense.

In addition to the correctness properties listed above, the *currency requirement* also needs to be met for applications. Thus, the mechanisms that enforce this correctness criterion have to ensure that the data accessed by the clients is current, that is, reflect the most recent updates to the data.

Whereas we independently discovered the above correctness criterion when we began investigating appropriate correctness criteria for broadcast environments, our correctness criterion bears some similarities to those proposed in [15] and is similar to the "update consistency" correctness proposed in [6]. These correctness criteria, however, were proposed in the context of different system architectures and so the mechanisms proposed for implementing them did not take into account the nature of the broadcast medium ([15] and [6] propose locking based protocols which, as outlined before, are infeasible in broadcast environments). Moreover, unlike our work, a formal characterization of the correctness criterion was not presented. We provide not only a formal analysis of the correctness criterion but also propose and evaluate mechanisms by which this criterion could be exploited in the context of broadcast environments.

2.3.2 Formalizing the Correctness Criterion

A history $\mathcal{H}$ of operation events and transaction management events satisfies the weakened requirements iff:

1. The final state of the database after the occurrence of all the operations of committed transactions in $\mathcal{H}$, in the same order as in $\mathcal{H}$, is the same as the state of the database on running each committed *update* transaction in $\mathcal{H}$ in some serial order, S.
2. Given such an S, at least one of the following two statements hold:
   - Each transaction in S reads the same values of objects and performs the same transformation on the database as it does in the history $\mathcal{H}$
   - There exists some suffix of transactions in S such that when these transactions are run in the same order as in S on any state reached by the serial executions of transactions on the initial state of the database, they read the same values of objects and perform the same transformation on the database as they did in the history $\mathcal{H}$.
3. Each committed transaction in $\mathcal{H}$ reads the same values of objects as in the state after some serial execution of some subset of committed transactions in $\mathcal{H}$ (which set of committed transactions is not important).
4. Requirements 2 and 3 also hold for each prefix $\mathcal{H}'$ of $\mathcal{H}$.

The first requirement states that the final state of the database should be the same as that obtained by running all the committed update transactions in some serial order.

The second requirement states that all the update transactions in the serial order of execution that affect the final state of the database should read the same values of objects in the serial order as they do in the history. Thus, the updates performed by update transactions are propagated to the database in a consistent fashion.

The third requirement states that each transaction is serialized after a subset of committed update transaction. This serial order need not be consistent with the serialization order of update transactions. Note that this requirement neither says that read-only transactions have to be serialized with respect to all the update transactions nor places any restriction on the subset of update transactions with respect to which a read-only transaction needs to be serialized.

The final requirement ensures the prefix commit closed property of the correctness criterion.

It is easy to see that the proposed correctness criterion is weaker than serializability, i.e., every history that is serializable also satisfies the new correctness criterion. We now illustrate, through examples, that certain non-serializable histories also satisfy the new correctness criterion.

Consider the history in Example 2. As mentioned earlier, this history is not serializable. However, this history satisfies the new correctness criterion. This is because the final state of the database is the same as the state obtained after running the update transactions t_2 and t_4 in some serial order (satisfies requirement 1). Also, each update transaction (vacuously) reads the same values of objects in the history as it does in the serial order (satisfies requirement 2). Further, each read-only transaction is serialized after the update transaction it reads from - t_1 after update transaction t_4 and t_3 after update transaction t_2 (satisfies requirement 3). The above three properties can also be verified for every committed prefix of the history (satisfies requirement 4).

As another example, consider the history in Example 3. This history is again not serializable. However, all update transactions are serializable (satisfies requirements 1 and 2). Also, the read-only transaction t_3 is serializable with respect to the update transaction t_2 it read from (satisfies requirement 3). Further, every prefix of the history satisfies requirements 1 through 3 (satisfies requirement 4). The history thus satisfies the new correctness criterion.

2.3.3 Formal Characterization of Histories Satisfying the Correctness Criterion

We syntactically characterize the set of histories that satisfy our correctness criterion. This characterization captures the essence of the correctness criterion and is useful in determining the complexity of recognizing histories satisfying the correctness criterion (in Section 3.1).

Let t be a transaction which executes in a history $\mathcal{H}$. Then, the *set of live transactions with respect to t in the history $\mathcal{H}$*, $LIVE_{\mathcal{H}}(t)$, is the minimal set closed under the following two rules: (a) t is in $LIVE_{\mathcal{H}}(t)$ and (b) If t' is in $LIVE_{\mathcal{H}}(t)$, then all transactions t'' such that t' reads the value of an object written by t'' in $\mathcal{H}$ are also in $LIVE_{\mathcal{H}}(t)$. Intuitively, the set of live transaction with respect to a transaction t is the set of transactions that t directly or indirectly reads from. The *update subhistory* of a history $\mathcal{H}$, $\mathcal{H}_{update}$, is a projection of the history $\mathcal{H}$ which includes all and only the operations performed by transactions that perform a write operation in $\mathcal{H}$.

The following theorem[1] characterizes the set of histories that a scheduler can determine to satisfy requirements 1 to 3.

[1]In the interest of space, only the statement of this theorem (and the ones that follow) is included in the main body of the text. The proofs of all the theorems can be found in the appendix.

Theorem 1 A scheduler can determine that a history $\mathcal{H}$ satisfies requirements 1 through 3 iff (a) $\mathcal{H}_{update}$ is view serializable [25, 5, 18] and (b) for each read-only transaction t, the subhistory formed by the transactions in $LIVE_{\mathcal{H}}(t)$ is also view serializable (ignoring the final database state).

Because of (a), all update transactions are view serializable and because of (b), each read-only transaction is serialized after all the update transactions it directly or indirectly reads from.

2.4 Applicability of Correctness criterion to Broadcast Environments

The first observation is that there is a "decoupling" between the correctness imposed on update transactions and read-only transactions. What this means is that the new correctness criterion is not a global property like serializability. This could be effectively exploited in a broadcast environment where update transactions could be validated at the server while the read-only transactions can be validated at the clients without the clients having to send any information to the server.

The second interesting feature is that from a correctness point of view, unlike in traditional serializability, read-only transactions are independent of each other. This is because different read-only transactions can see different serial orderings of update transactions. So, information about read-only transactions executing at a client need not be conveyed to other clients. Thus, the performance of the system for read-only transactions would be unaffected by an increase in the number of clients because the clients are independent of each other and do not need to communicate with the server.

The third useful feature is that a read-only transaction need be serializable only with respect to update transactions it directly or indirectly reads from. This property is extremely useful when processing update transactions because previous reads by read-only transactions are unaffected by new update transactions. Example 3 illustrates this point. This offers the much needed autonomy between the clients and the server.

We thus see that the proposed correctness criterion addresses the problems with serializability as discussed in Section 2 and accounts for the special characteristics of broadcast environments and the lack of sufficient uplink capability from the clients to the server. The only issue about the new correctness criterion that might seem to be of potential concern is the fact that transactions executing at the same client can see different serial orders of execution of update transactions. There are two cases to be considered here. In the first case, concurrently executing read-only transactions at a client see different serial orders of update transactions (consider Example 2 where t_1 and t_3 execute at the same client). This, however, does not lead to any inconsistencies because the transactions are allowed to see different orders of updates precisely because the updates are unrelated. Furthermore, because the mechanisms proposed to implement the correctness criterion also satisfy the currency requirement, they ensure that if a read-only transaction (say t_i) starts executing after the completion of another read-only transaction (say t_j) at a client, then t_i and t_j see the serial ordering of transactions they both depend on (directly or indirectly) in a consistent fashion. Thus, a transaction that is executed based on the results of another transaction will not see an inconsistent history.

The second case is when read-only transactions and update transactions see different serial orders of execution (all update transactions themselves are globally serializable). Here again, the problem arises only when a read-only (update) transaction starts executing after the completion of an update (read-only transaction). But, because the mechanisms that we use to enforce the correctness criterion satisfy the currency requirement, they ensure that the two transactions see consistent serial orderings of the transactions they both depend on. Thus, the new correctness criterion along with the implementation proposed in this paper are well suited for broadcast environments.

3 Mechanisms to Guarantee Correctness

In this section, we address the practical issues involved in enforcing the proposed correctness criterion in broadcast environments. The problem of determining if a history satisfies the correctness criterion is **NP-Complete**. We hence propose a polynomial time approximation algorithm, *APPROX*, to efficiently determine legal histories. A mechanism to implement this algorithm in broadcast disk [4] environments, *F-Matrix*, is also described. We then propose, *R-Matrix*, a simpler (in terms of space, time and effectiveness in determining legal histories) version of *F-Matrix*. Finally, we outline how client caching techniques for broadcast environments can be adapted and used in conjunction with the proposed mechanisms.

3.1 Computational Problems Relating to the Correctness Criterion

A history is *legal* iff the history satisfies requirements 1 through 3 (see theorem 1).

Theorem 2 The problem of determining whether a history $\mathcal{H}$ is legal is **NP-Complete**.

So, it is very unlikely that an efficient (practical) algorithm exists to determine all and only legal histories. However, because view serializability is **NP-Complete**, in practice, a stronger but polynomially checkable property, conflict serializability [5], is enforced. The question that now arises is: What is the complexity of determining whether a history $\mathcal{H}$ is legal, given that $\mathcal{H}_{update}$ is conflict serializable? We can show that this problem is also **NP-Complete**. In fact, we can prove that even if all the update transactions are *serially* (not to be confused with serializably) executed, it is still **NP-Complete** to determine whether a history is legal. This means that there probably does not exist an efficient way to determine whether a history is legal, when using virtually any serializability based concurrency control algorithm for update transactions.

Theorem 3 Consider a history $\mathcal{H}$ such that $\mathcal{H}_{update}$ is a serial history. Then the problem of determining whether $\mathcal{H}$ is legal is **NP-Complete**.

3.2 A Simple Approximation Algorithm

Given the intractability result of the previous section, we now propose a polynomial time algorithm that accepts a set of histories that is a proper subset of legal histories. The fact that only a proper subset of legal histories is accepted implies that the algorithm accepts only legal histories though some legal histories may not be accepted.

The approximation algorithm *APPROX* determines that a history is legal iff both of the following conditions hold:

1. $\mathcal{H}_{update}$ is conflict serializable.

2. For every read only transaction t_R in the history $\mathcal{H}$, $S_{\mathcal{H}}(t_R)$ is acyclic. Here $S_{\mathcal{H}}(t_R)$ is the serialization graph consisting of all the transactions (including t_R) that t_R directly or indirectly read from[2].

The intuition behind this algorithm is to replace *all* requirements for view serializability with conflict serializability. Since conflict serializability is an efficient alternative to view serializability, we would expect the algorithm *APPROX* to be efficient. The following two theorems indicate that *APPROX* efficiently recognizes legal histories.

Theorem 4 *APPROX* accepts a proper subset of the set of legal histories.

Theorem 5 *APPROX* is in **P**.

3.3 Mechanisms for Enforcing the Correctness Criterion in Broadcast Disk Environments

In this section, we first discuss how *APPROX* can be implemented in broadcast disks – which is a specific instance of a broadcast environment. We propose a simpler algorithm that approximates *APPROX* but which is much more space efficient. We then qualitatively compare the algorithms with the existing Datacycle concurrency control algorithm [12]. Finally, we study how client caching can be incorporated into the proposed concurrency control mechanisms. The mechanisms proposed in this section handle both read-only and update transactions.

Before we go into the specifics of the algorithm implementation, we first outline some relevant characteristics of broadcast disks. Broadcast disks [26] are a form of data dissemination systems that are well suited for wireless and mobile computing. The server periodically broadcasts all the data items in the database. The clients view this broadcast as a disk and can read the values of data items being broadcast. In order to write onto objects, the clients contact the server with the appropriate information. In this paper, we consider only single speed disks. Our protocol requires additional control information information to be broadcast in order to ensure correctness and so the key issue is to minimize the overheads entailed by this additional information.

3.3.1 The *F-Matrix* Implementation

In this section, we describe one implementation of *APPROX*, namely *F-Matrix* (Full Matrix), that is appropriate for broadcast disk environments. We outline the server functionality, the client functionality, the nature of the control information transmitted from the server to the clients, the client read-only transaction validation protocol and the details on how the control information is computed at the server.

Server Functionality

The server performs the following functions:

1. During each cycle, broadcasts the latest *committed* values of all data items at the beginning of the cycle. Note that this implies that the server has to maintain two versions of objects, one the latest committed version and the other the last written version[3].

[2] For a more precise definition of $S_{\mathcal{H}}(t_R)$, see Appendix C

[3] The maintenance of two versions of objects has some commonality with multiversion concurrency control [5]. Our concern here is about clients which maintain only a single version.

2. Using a concurrency control mechanism, ensures the conflict serializability of all transactions submitted to the server (some of these may originate at the clients and submitted for validation as described later). The exact information a client must provide along with its update transaction is discussed under client functionality. We do assume, that if a transaction commits, then all transactions from which that transaction reads have previously committed.
3. Transmits a *control matrix* during each cycle that helps clients determine whether read-only transactions read consistent values. The control matrix will be described shortly.

In summary, the server is responsible for transmitting the latest (to within a cycle) committed values of data items, ensuring the conflict serializability of transactions submitted to it and transmitting enough control information so that read-only transactions at clients can validate their reads – which may span multiple cycles.

Client Functionality

Clients handle two types of transactions: read-only transactions and update transactions. Read, write, commit and abort operations performed by a transaction are handled as follows:

- *Read Operation:* Before a read operation is performed on a data item broadcast during a cycle, the control information transmitted during that cycle is consulted to determine whether the read operation can proceed (the exact details about the nature of this check will be described shortly). If the read operation cannot proceed the transaction is aborted.
- *Write Operation:* When a data item is written, the write is performed on a local copy of the data item in the client. No checks are made.
- *Commit:* If the transaction has not performed any write operation, then the commit operation does not have to do anything and the commit succeeds. In case the transaction has performed a write operation, a list of all the objects written and the values written are sent to the server. In addition, the list of all read operations performed and the cycle numbers in which they are performed are sent to the server. The server checks to see whether the update transaction can be committed using any of the concurrency control protocols outlined under server functionality. If so, the transaction is committed, else it is aborted. This method of handling update transactions is similar to the method proposed in [14].
- *Abort:* If the transaction has not performed any write operation, then the abort does nothing (other than stopping further execution of the transaction and discarding its writes).

Nature of Control Information

We now describe the nature of the control information transmitted by the server and show how it is updated at the end of each cycle. The control information matrix at any point in time is an $n \times n$ matrix, C, where n is the number of objects. If objects are assumed to be have ids ob_1 through ob_n, each entry $C(i, j)$ is set to a cycle number determined as follows.

Let $\mathcal{H}$ be the history of execution of the committed update transactions at the server. Also, let t_j be the last committed update transaction that wrote onto ob_j. We assume that a transaction t_0 writes onto all data items at cycle 0 (before the beginning of the broadcast). Then:

$$C(i, j) = \max_{t' \in LIVE_{\mathcal{H}}(t_j) \wedge \text{t' writes object } ob_i} (\text{cycle number in which t' committed})$$

As defined earlier, $LIVE_{\mathcal{H}}(t_j)$ refers to the set of transactions (including t_j) that t_j directly or indirectly reads from. Thus, $LIVE_{\mathcal{H}}(t_j)$ is the set of transactions that "affect" the latest committed value of ob_j (because t_j writes the latest committed value of ob_j). $C(i,j)$ is thus the latest cycle number in which some transaction that affects the latest committed value of ob_j and also writes to ob_i, commits. The following example illustrates how the entries in the C matrix are determined.

Example 4: Consider the following history:

$$w_1(ob_1)w_1(ob_2)c_1r_2(ob_1)w_2(ob_1)c_2r_3(ob_2)w_3(ob_2)c_3$$

and assume that the commit operation c_i occurs during the i^{th} broadcast cycle .

In the above scenario $C(1,1) = 2$ because t_2 was the last transaction to write onto ob_1 (thus "affecting" the value of ob_1) and it committed during cycle 2 of the broadcast. For similar reasons, $C(2,2) = 3$. The value of $C(1,2) = 1$. This is because t_3 was the last committed transaction to write onto ob_2 and $LIVE_{\mathcal{H}}(t_3) = \{t_1, t_3\}$ and t_1 is the only transaction in $LIVE_{\mathcal{H}}(t_3)$ to write onto ob_1 and it does so during cycle 1 of the broadcast. For similar reasons, $C(2,1) = 1$.

So far, we have assumed that each entry of the C matrix has to store the entire cycle number. This could be avoided if we know the maximum number of cycles that a transaction could span (max_cycles). In that case, we need to store only cycle numbers from 0 to max_cycles and perform modulo $max\_cycles + 1$ arithmetic and comparisons. This would reduce the size of each entry in the C matrix. The issue of how the control information is to be split across the broadcast cycles also needs to be addressed. One effective way to partition the C matrix is to broadcast the j^{th} column right after broadcasting the object ob_j. This would minimize the amount of time the client would have to wait for control information, as will become apparent in the next section.

Client Read-Only Transaction Validation Protocol

For a read-only transaction t at the client, the following protocol is followed before each read operation. Let R_t be the set of $(ob_i, cycle)$ pairs such that transaction t previously read object ob_i from broadcast cycle $cycle$, i.e., t read the latest committed value of ob_i at the beginning of cycle $cycle$. Then a read operation on an object ob_j is allowed to proceed in a cycle iff the following condition, *read-condition(ob_j)*, holds (thereby preserving conflict serializability):

$$\forall (ob_i, cycle) \in R_t \quad (C(i,j) < cycle)$$

where C is the matrix at the beginning of the current cycle. If the *read-condition* fails, the transaction is aborted. Intuitively, a read operation performed by a transaction t on object ob_j is allowed to be performed iff no transaction in $LIVE(t)$ wrote onto ob_i after t read from the ob_i.

Theorem 6 A read-only transaction t_R is allowed to commit by the protocol described above if $S(t_R)$ is acyclic.

Theorem 7 For a read-only transaction t_R, if $S(t_R)$ is acyclic, then the protocol detailed above allows t_R to commit.

Computing Control Information

As a transaction commits, the entries affected by it are updated by the server. For simplicity, we consider only the simple case where the entries are updated as per a serialization order (see Section 5 for other options).

We now show how the matrix C can be incrementally constructed. At the beginning of the broadcast, each entry in C is set to the cycle number 0. Let C_{old} be the matrix that considers all the committed transactions in a serial order up to a certain point in time in a broadcast cycle c_1. Now, let t be a newly committed transaction that occurs immediately after all previously committed transactions in the update transaction serialization order. Also, let $c_2 \geq c_1$ be the cycle in which t commits. Let RS_t be the set of objects read by t and let WS_t be the set of objects written by t. Then the new matrix C_{new} is computed as follows:

- $C_{new}(i,j) = c_2$ if $ob_i, ob_j \in WS_t$

  Since transaction t writes to objects ob_i, ob_j and it is the last transaction to write onto ob_j in the serial order of execution, t is the last transaction that affects the latest value of ob_j and also writes to ob_i. Thus, the entry is set to the cycle in which t commits.
- $C_{new}(i,j) = \max_{ob_k \in RS_t}(C_{old}(i,k))$, $ob_i \notin WS_t$ and $ob_j \in WS_t$

  Since transaction t reads from objects in RS_t, the transactions t depends on is the set of all transaction that directly or indirectly affected the latest values of items written to the objects in RS_t. Thus, each entry is updated to reflect this fact. If the transaction t does not read any object, then this C entry is set to 0.
- $C_{new}(i,j) = C_{old}(i,j)$ otherwise

  If an object ob_j is not updated, entries in column j are untouched.

Theorem 8 The above incremental algorithm preserves the semantics of the C matrix.

There are a few potential drawbacks of *F-Matrix*. One is that computing the matrix may be expensive in terms of server time. We, however, do not expect this problem to be as severe. In practice, because *APPROX* is based on a validation based approach to effecting clients' updates, it is unlikely to perform well when the number of update transactions run at the server is large (for the same reason that an optimistic method is unlikely to perform well in high contention environments). Thus, the computation of the matrix may very well be feasible within the working limits of *APPROX*.

The other potential drawback of the *F-Matrix* mechanism is that the bandwidth required to transmit the C matrix during each broadcast cycle. The space requirements for the C matrix in *F-Matrix* is $n^2 \times \log(max\_cycles)$ bits per broadcast cycle, where n is the number of objects in the database and max_cycles is the maximum number of broadcast cycles spanned by a transaction. This cost can be significant for large values of n. Quantitatively, if the size of each object is s bits, then the fraction $\frac{n^2 \times \log(max\_cycles)}{n^2 \times \log(max\_cycles) + n \times s} = \frac{n \times \log(max\_cycles)}{n \times \log(max\_cycles) + s}$ of the broadcast cycle time is spent on control information. If s is small, this overhead can be significant. The overhead may not be significant for large values of s. This problem is actually slightly worse than it seems. This is because the following theorem holds regardless of the compression technique used for the C matrix.

Theorem 9 In the worst case, the number of bits to be transmitted in order to represent the C matrix during each broadcast cycle is at least $\frac{(n^2 - 4 \times n + 3) \times \log(max\_cycles)}{4}$.

The above theorem, however, assumes for simplicity that the entire C matrix is transmitted in each broadcast cycle. The number of bits to be transmitted may be drastically reduced if we transmit only changes (deltas) over the previous C matrix transmission (see Section 5).

3.3.2 The *Datacycle* and *R-Matrix* Implementations

In the previous section, we described the *F-Matrix* mechanism to implement the *APPROX* algorithm in broadcast disk environments and also outlined some of the potential overheads of computing and transmitting the C matrix. In order to get around the drawbacks of *F-Matrix* mentioned above, *F-Matrix* could be modified as follows. Instead of viewing each object as a separate entity, we can partition them into groups of objects. The C matrix will now not be a $n \times n$ matrix, where n is the number of database objects, but would be a $n \times g$ matrix, where g is the number of groups in the database, thus reducing the size of the control information. Further, the fact that there are fewer entries to update would make it more efficient to update the matrix.

Each entry of the modified C matrix, $MC(i, s)$, where s is a set of objects, is defined by:

- $MC(i, s) = \max_{ob_j \in s}(C(i, j))$

Intuitively, for the purposes of the matrix, we consider all database items in the set s to be indistinguishable. In the client, the check is performed as in the *F-Matrix* case, with the following modification to account for the grouping of objects. Let R_t be the set of $(ob_i, cycle)$s such that transaction t previously read object ob_i in cycle $cycle$. Then *read-condition(ob_j)* becomes:

$$\forall (ob_i, cycle) \in R_t \forall s \quad (ob_j \in s \Rightarrow \quad MC(i, s) < cycle)$$

where MC is the matrix at the beginning of the current cycle and the domain of s is the set of database partitions. s being tunable, we have a spectrum of mechanisms implemented by varying the size of the database partitions (in the extreme case, where all database partitions are singleton sets, we have the the implementation *F-Matrix*). There is a tradeoff in choosing the size of the partition: increasing the partition size would mean that certain unnecessary conflicts would be detected, while reducing the partition size would imply increased control information overhead. In this paper, we concentrate on the two extreme cases: (a) all database partitions are singleton sets (*F-Matrix*) and (b) all database items fall under one partition.

In case (b), the matrix entry associated with each object is just the latest cycle number in which a committed value was written to it (i.e., the matrix is in fact just a vector). Thus, *read-condition(ob_j)* is simplified to:

$$\forall (ob_i, cycle) \in R_t \quad (MC(i, db) < cycle)$$

where db is the set of all the objects in the database. In other words, a transaction can proceed to read an object only if no previously read value has been updated. This corresponds to the Datacycle approach for concurrency control [12] and ensures serializability. For the rest of this paper, we call this approach *Datacycle*. The *Datacycle* approach, however, does not fully realize the potential of the matrix reduction. We can in fact weaken *read-condition(ob_j)* for a read on an object ob_j performed by a transaction that read the first data item at cycle c_1 to:

$$\forall (ob_i, cycle) \in R_t \quad (MC(i, db) < cycle) \quad \vee \quad (MC(j, db) < c_1)$$

where db is the set of all the objects in the database. We call this modified algorithm *R-Matrix* (for reduced matrix). The idea behind this algorithm is that a transaction reads consistent values if either (a) no values it has previously have been overwritten by other transactions or (b) (even though previously read values may have been overwritten by other transactions) the current value being read has not been overwritten since the beginning of the transaction.

Theorem 10 The *R-Matrix* implementation accepts only schedules accepted by *APPROX*.

Optimizations similar to those done in *Datacycle* could also be done for *R-Matrix*. Thus, a bit could be set by hardware if any of the previously read values of a transaction are changed. For a future read operation invoked on an object, if the bit is set and if the object being read has been changed during or after the cycle in which the first read operation was performed by the transaction, the transaction is aborted. Otherwise, the read operation is allowed to proceed. Note that this mechanism, besides reducing the number of unnecessary aborts of read-only transactions, has a nice "stability" property unlike *Datacycle* where a transaction may be aborted even though it does not perform any further reads.

3.3.3 Client Caching

We now outline the issues involved in client caching and illustrate how our concurrency control mechanisms can be extended to work with client caches.

Client caching in broadcast environments [1] is typically done to improve the client application performance by storing frequently accessed data items in the client cache. Thus, the client does not have to wait to read the data off the broadcast and can directly read it from the local cache. Since the client cache is typically not large enough to hold all the items being broadcast, appropriate cache replacement policies are used [1]. In the presence of updates, however, the issue becomes more complex. This is because the data in the client caches may be "inconsistent" [3] with the data being broadcast due to updates at the server. Mechanisms to deal with this problem in the context of broadcast environments have been proposed [2].

The mechanisms described in [2] are for maintaining the consistency of data items read, on a per object basis. In order to maintain the mutual consistency of the data items read by a transaction, we only need to store the columns of the matrix corresponding to the data items cached, along with the cycle in which the data items were cached. In this way, a transaction that accesses cached data items (in addition to possibly data items read off the broadcast medium) has all the information necessary for validation when using the *F-Matrix* and *R-Matrix* mechanisms.

In summary, client caching in broadcast environments has dealt with maintaining per object consistency. With minor modifications, our mechanisms be extended to maintain inter-object consistency by storing the relevant columns of the concurrency control matrix.

4 Simulation Results

This section describes the experimental setup for the simulations. The simulations were aimed at comparing the performance of four different algorithms (*Datacycle*, *F-Matrix*, *R-Matrix* and *F-Matrix-No*) for concurrency control of client read-only transactions in broadcast disk environments. The first three algorithms are described in the previous section. The last algorithm is used as a baseline and implements the functionality of *F-Matrix*, but ignores the cost of broadcasting the control information associated with each object in the database. Hence, the broadcast cycle lengths are shorter for runs of this algorithm as compared to those for *F-Matrix*. We do not consider the effects of caching in this performance study but plan to explore this as part of future work (see Section 5).

Parameter	Default Value
Client Transaction Length (number of read operations)	4
Server Transaction Length (number of read/write operations)	8
Transaction Rate at Server (rate of completion of transactions at server)	1 in 250000 bit-units
Number of Objects in Database	300
Size of Objects in Database	1 KB
Server Read Operation Probability	0.5
Mean Client Inter-Operation Delay (exponential distribution)	65536 bit-units
Mean Client Inter-Transaction Delay (exponential distribution)	131072 bit-units
Client Restart Delay (after an abort)	0 bit-units
Timestamp Size	8 bits

Table 1: Parameters for the Simulation

The performance of these algorithms can be compared relative to the following metrics:

- **Transaction Response Time** – the time the transaction is submitted by a client to the time the transaction commits. This includes the time involved in restarting the transaction (perhaps more than once, if necessary), if it aborts.
- **Transaction Restart Ratio** – the number of times a transaction, before it completes successfully, is aborted and restarted because the data it read was inconsistent.

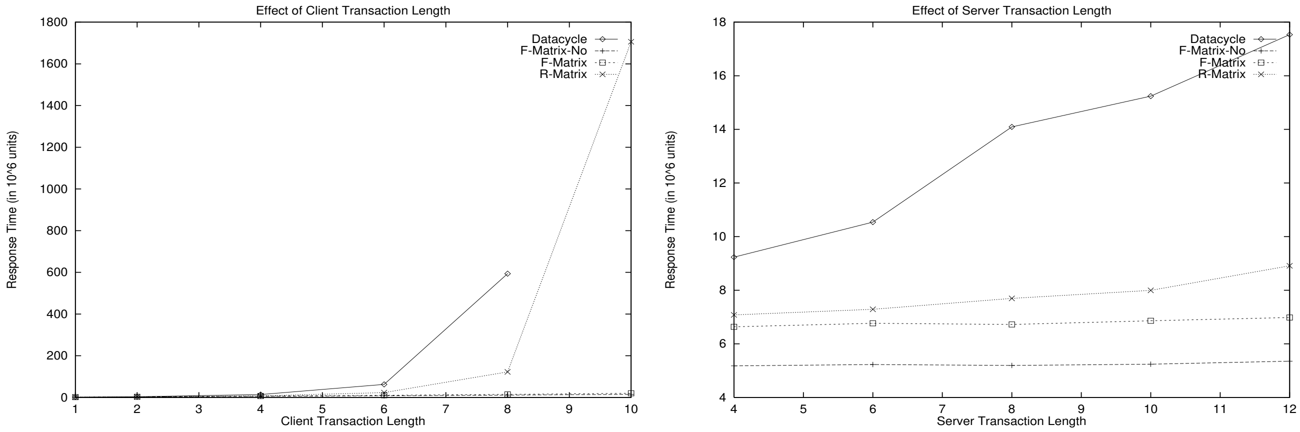
In the experiments, we observed that there was a very high correlation between the graphs depicting transaction restart ratio and response time (as was expected). Thus, in this paper, we present only the graphs that show the transaction response time.

4.1 Experimental Setup

Our simulator consists of a server, a client, a broadcast disk structure and an event queue. Read-only transactions alone execute on the client. The server executes update transactions, which read and write to the database. Only one client was simulated because the performance of the outlined concurrency control mechanisms for read-only transactions is independent of the number of clients. The objects that the transactions access are determined using a uniform distribution on the objects in the database. Table 1 lists the simulation parameters.

The server fills the broadcast disk with the data at the beginning of a cycle. To *simulate* the database objects being broadcast at different times, a time entry exists for each object. This is set to the time at which the object is actually broadcast. The client waits until this time to read the corresponding object. Each cycle consists of a broadcast of all the objects in the database along with the associated control information. For *F-Matrix*, the control information consists of an $(n \times n)$ matrix, where n is the number of objects. Each column of the matrix is broadcast along with the corresponding object. For *R-Matrix* and *Datacycle*, the control information consists of an array of length n. Each element in the array is broadcast along with the corresponding object.

The time to broadcast one bit (one bit-unit) was used as the unit of time. All response times were measured in terms of this unit. We systematically evaluated the algorithms with respect to different settings of the first 5 parameters. For each simulation, the parameter that was being



(a) Effect of Client Transaction Length

(b) Effect of Server Transaction Length

Figure 2: Effect of Transaction Length

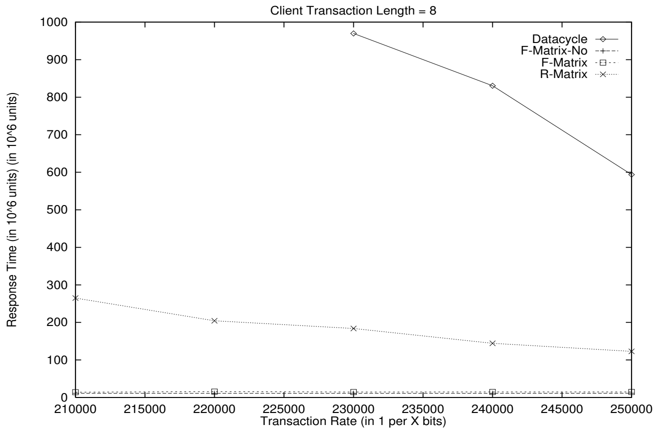
studied was varied while the other parameters were fixed at the default value. Each experiment consisted of 1000 client transactions and the simulations were run until all of them committed successfully. The data was derived from the last 500 transactions to ensure that it was "steady-state" data. In these experiments, 95% confidence intervals were obtained, whose widths are less than 10% of the point estimates of the response times.

For all the experiments, for *F-Matrix* the fraction of the cycle used for the control information (in bits) is given by $\frac{n^2 \times TS}{n^2 \times TS + n \times OBJ} = \frac{n \times TS}{n \times TS + OBJ}$, where TS is the size of a timestamp and OBJ is the size of an object. For a timestamp size of 8 bits and an object size of one kilo Byte and 300 objects in the database, the overheads work out to about 23%. For *R-Matrix* and *Datacycle*, the fraction of cycle time used for control information is $\frac{n \times TS}{n \times TS + n \times OBJ} = \frac{TS}{TS + OBJ}$. For the same parameters as above, the overheads in this case are only about 0.1%.

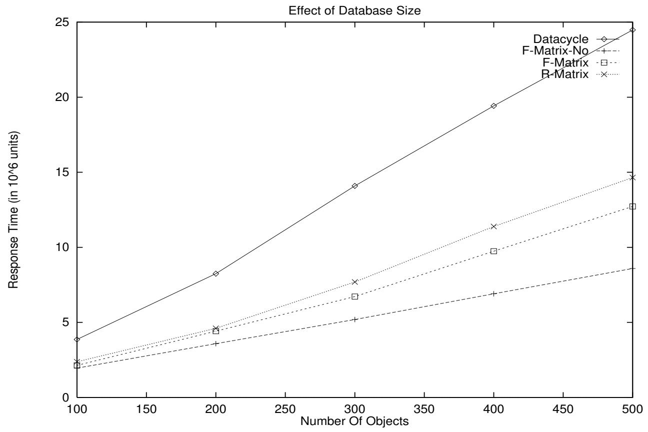
4.2 Effect of Client Transaction Length

Until client transaction length 4, all the algorithms have similar performance (see Figure 2(a)). But after that, and especially beyond a value of 6, the *Datacycle* algorithm performs very poorly so much so that for a length of 10, its response time was outside the limits of the Y-axis and hence is not shown. Even though *R-Matrix* is far better than *Datacycle*, *F-Matrix* shows even better behavior. As can be seen from the graph, for a client transaction of 8, *R-Matrix* has a response time of 122.68×10^6 while *F-Matrix* has a response time of 14.6×10^6 units (about 12% of *R-Matrix*'s response time). This is because *F-Matrix* reduces the number of client transaction aborts (to almost zero) by using the weaker read condition made possible by the extra control information that is broadcast along with the data. Also, note that the performance curve for *F-Matrix* is flat, indicating that it scales very well with increasing client transaction sizes.

These curves (because of the scale of the Y-axis) do not bring out the small differences that do exist between the performance of *F-Matrix* and *F-Matrix-No*. These differences, magnified in Figure 3(b), arise because in *F-Matrix-No* the time it takes to transmit the control information is ignored. So, read transactions wait for a shorter time for the broadcast data to be available for



(a) Effect of Server Transaction Rate



(b) Effect of Number of Objects

Figure 3: Effect on Response Time

reads. The number of update transactions per cycle is also reduced and hence, there are lesser read conflicts in each cycle. These factors lower the response times for the ideal *F-Matrix-No*.

4.3 Effect of Server Transaction Length

Longer server transactions lead to more updates at the server for each cycle. Hence, the response time tends to increase with server transaction length in Figure 2(b). However, *F-Matrix* shows very little increase in response time compared to *Datacycle* and even *R-Matrix*. Once, again this demonstrates the scalability properties of *F-Matrix*.

4.4 Effect of Transaction Rate at Server

In Figure 3(a), the server transaction rate (X-axis) decreases as we go from left to right. This graph shows that, as expected, response time tends to improve with a decrease in server transaction rates. It can also be seen that *F-Matrix* has a performance that is very close to that of *F-Matrix-No* and is better that *R-Matrix*, which in turn is much better than *Datacycle*. Also, *F-Matrix* displays almost no degradation in performance as server transaction rates increase. This scalability is in complete contrast with Datacycle which has very poor performance at higher transaction rates.

4.5 Effect of Number of Objects and Size of Objects

As the number of objects in the database increases, the probability of the transactions accessing an object decreases. The length of the cycle, however, increases with the number of objects. The increase in the control information to be sent (owing to the increase in the number of objects) also increases the length of the broadcast cycle. This increases the number of server transactions per cycle and hence, increases the number of possible conflicts. For these reasons, the response times increase with the size of the database. But, as can be seen from Figure 3(b), the relative ordering of the four protocols remains the same with *F-Matrix* displaying the best response times (about 9.6×10^6 units for a database size of 400, as compared to nearly 11.3×10^6 for *R-Matrix*) among the three practical protocols. The rate of increase in response time with increase in the number of objects is also smaller for *F-Matrix*.

The relative behavior of the algorithms with increasing size of objects is similar to Figure 3(b).

4.6 Summary of Results

R-Matrix outperforms *Datacycle* and *F-Matrix* outperforms *R-Matrix* in all the experiments. Thus, the experimental results confirm the hypothesis that a weaker abort condition would lead to better response times. Furthermore, *F-Matrix* is highly *scalable* with respect to client transaction length, server transaction length, and server transaction rate. All the protocols display increased response times with increase in broadcast cycle lengths (either due to increase in the number of objects or due to increase in the size of the objects), but the rate of increase in the case of *F-Matrix* is lower than that of *R-Matrix* as well as *Datacycle*. These impressive performance features of *F-Matrix* are observed in spite of *F-Matrix* requiring more bits to transmit the control information in each cycle. Also, in many cases its performance profile is very close to *F-Matrix-No* which ignores all communication costs for the control info.

F-Matrix and *R-Matrix* represent two ends of a spectrum. *F-Matrix* has the overhead of transmitting control information of size $O(n^2)$, where n is the number of objects. *R-Matrix* presents a cheaper alternative, though it does not perform or scale as well as *F-Matrix*. In general, the choice of the algorithm to be used would depend on the capacity of the broadcast medium as well as the profile (number of objects, frequency of updates, etc.) of the database that is to be used.

5 Conclusions and Future Work

There are several possible applications of broadcast based database systems. One such application involves a server that stores stock trading data for a large stock market. It continuously broadcasts information about current prices and trading volumes of various financial instruments and current values of various market indexes. Clients (brokers, stock traders, market regulators) use mobile workstations to access this data and perform a wide variety of financial transactions. Since it is important to keep data (e.g., stock trades or liability of a broker) consistent at both server and clients, operations performed by clients must have suitable transactional semantics. Catering to such requirements through efficient concurrency control is the subject of this paper. Figure 4 shows the development of ideas in this paper.

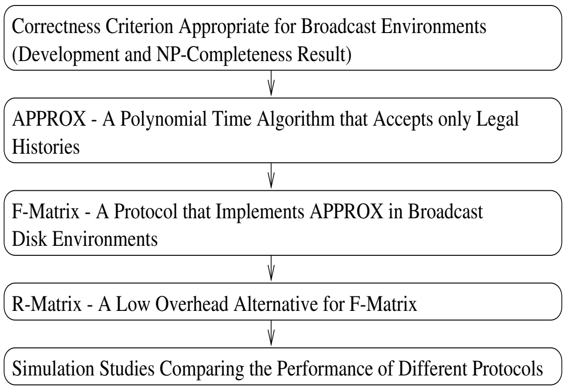


Figure 4: Development of Ideas

We plan to follow a number of threads in terms of future work: efficient parallel computation and incremental transmission of the control matrix, quantitative evaluation of the performance in the presence of caching at the clients, and extensions to optimize for update transactions at clients.

References

[1] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments," Proceedings of the ACM SIGMOD Conference, California, May 1995.

[2] S. Acharya, M. Franklin and S. Zdonik, "Disseminating Updates on Broadcast Disks," *Proceedings of 22nd VLDB Conference*, Mumbai(Bombay), India, 1996.

[3] R. Alonso, D. Barbara, H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," ACM Transactions on Database Systems, vol. 15, no. 3, September 1990.

[4] R. Alonso, H. Korth, "Database Systems Issues in Nomadic Computing," Proceedings of the ACM SIGMOD Conference, Washington D.C, June 1993, pp. 388-392.

[5] P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, Massachusetts, 1987.

[6] P.M. Bober, M.J. Carey, "Multiversion Query Locking," Proceedings of the VLDB Conference, Vancouver, Canada, August 1992.

[7] M.J. Carey, M.J. Franklin, M. Livny, E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures,"Proceedings of the ACM SIGMOD Conference, Denver, June 1991.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms," McGraw-Hill Book Company, 1990.

[9] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," Ph.D. Thesis, Department of Computer Sciences, University of Wisconsin-Madison, 1993.

[10] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco, 1979.

[11] S. Gukal, E. Omiecinski, U. Ramachandran, "Transient Versioning for Consistency and Concurrency in Client-Server Systems," Proceedings of the Conference on Parallel and Distributed Information Systems (PDIS), Florida, December 1996.

[12] G. Herman, G. Gopal, K.C. Lee, A. Weinreb, "The Datacycle Architecture for Very High Throughput Database Systems," Proceedings of the ACM SIGMOD Conference, New York, 1987.

[13] T. Imielinski and B. R. Badrinath, "Mobile wireless computing: challenges in data management," *Communications of the ACM, Vol. 37, No. 10*, October 1994, pp. 18-28.

[14] S. Kumar, E. Kwang, D. Agrawal, "Caprera: An Activity Framework for Transaction Processing on Wide-Area Networks," Proceedings of the VLDB Conference, Athens, Greece, August 1997.

[15] H. Garcia-Molina and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Transactions on Database Systems, Vol. 7, No. 2*, June 1982, pp. 209-234.

[16] B. Oki, M. Pfluegl, A. Siegel, D. Skeen, "The Information Bus - An Architecture for Extensible Distributed Systems," Proceedings of the SOSP Conference, North Carolina, December 1993.

[17] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," Journal of the ACM, Vol. 26, No. 4, October 1979, pp. 631-653.

[18] C. H. Papadimitriou, "The Theory of Database Concurrency Control," Computer Science Press, 1988.

[19] J. Shanmugasundaram, A. Nithrakashyap, J. Padhye, R. Sivasankaran, M. Xiong, K. Ramamritham, "Transaction Processing in Broadcast Disk Environments," Advanced Transaction Models and Architectures, S. Jajodia and L. Kerschnerg (eds.), Kluwer Academic Publishers, 1997.

[20] S. Shekar, D. Liu, "Genesis and Advanced Traveler Information Systems (ATIS): Killer Applications for Mobile Computing," MOBIDATA Workshop, New Jersey, 1994.

[21] White Paper, Vitria Technology Inc. (http://www.vitira.com).

[22] W. Wang, L. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture," Proceedings of the ACM SIGMOD Conference, June 1991.

[23] W. Wilkinson, M.A. Nemat, "Maintaining Consistency of Client Cached Data," Proceedings of the VLDB Conference, Brisbane, Australia, August 1990.

[24] P. Xuan, S. Sen, O.J.Gonzalez-Gomez, J. Fernandez and K. Ramamritham, "Broadcast on Demand - Efficient and Timely Dissemination of Data in Mobile Environments," IEEE Real-Time Technology and Applications Symposium, pp. 38-48, June 1997.

[25] M. Yannakakis, "Serializability by Locking," Journal of the ACM, Vol. 31, No. 2, April 1984.

[26] S. Zdonik, R. Alonso, M. Franklin, S. Acharya, "Are Disks in the Air Just Pie in the Sky," Proceedings of the Workshop of Mobile Computing Systems and Applications, Santa Cruz, California, December 1994.

In this paper, we assume that an execution trace of a transaction is a totally ordered sequence of events (which may be reads or writes on database items, commit or abort). This execution trace is determined by the associated transaction program, which could have conditional statements. Also, we assume that each history of transaction programs has an initial transaction t_0, which writes onto all the data items accessed (read or written) by transactions in the history. This transaction t_0 does not read any data item.

A Proof of Theorem in Section 2.3.3

We assume that all read operations performed by a transaction precede all write operations performed by the same transaction. We also assume that a transaction does not read from or write to the same object more than once. Further, we assume that objects can be created dynamically or equivalently, that either the scheduler does not know the number of database objects or that there are potentially infinite objects. These assumptions are made in order to keep the abstract characterization simple. The computational results proved in Section 3.1, however, are unaffected by these simplifying assumptions. Further, the mechanism outlined in Section 3 are general and do not rely on these assumptions.

In order to proceed with the formal characterization of the correctness criterion, we make the following assumptions about transactions and the knowledge available to the scheduler that determines legal histories.

Assumption 1 A transaction is a deterministic program, i.e., its actions depend only on the values it reads from the database.

Assumption 2 The scheduler which determines legal histories knows nothing about the internals of a transaction except that each transaction terminates when run in isolation on a consistent state.

Assumption 3 The scheduler which determines legal histories does not know the values read from and written to the database.

Assumption 4 The scheduler does not know the identities of transaction programs. In particular, the scheduler cannot detect that two instances of the same transaction program in a history are indeed two instances of the same transaction program.

Before we formally characterize the correctness criterion, we define a few terms.

Definition 1 The *reads from relation* associated with a history $\mathcal{H}$, $READS\_FROM_{\mathcal{H}}$, is the set $\{(t', ob, t'')|$t' reads the value written to variable *ob* by t" in $\mathcal{H}\}$.

The $READS\_FROM$ relationship captures the relationship between two transactions, one of which reads the value of a data item written by the other.

Definition 2 Let $\mathcal{H}$ be some history. Then the *affects set* of a read operation $r_t[ob]$ in $\mathcal{H}$, denoted by $AS_H(r_t[ob])$, is the set of operations recursively defined by the following rules:

- $r_t[ob]$ is in $AS_H(r_t[ob])$
- If $r_{t'}[ob]$ is in $AS_H(r_t[ob])$ and $r_{t'}[ob]$ reads from $w_{t''}[ob]$ in $\mathcal{H}$, then $w_{t''}[ob]$ is in $AS_H(r_t[ob])$
- If $w_{t'}[ob]$ is in $AS_H(r_t[ob])$ then all operations of the form $r_{t'}[ob']$ where ob' is some object and $r_{t'}[ob'] \rightarrow_{\mathcal{H}} w_[t'][ob]$ are in $AS_H(r_t[ob])$
- No other operations other than those included by the above three rules are in $AS_H(r_t[ob])$

Intuitively, the affects set of a read operation is the set of operations that "affected" (directly or indirectly) the value of the data item read.

Definition 3 Let $\mathcal{H}$ be some history. Then the *affects set* of a write operation $w_t[ob]$ in $\mathcal{H}$, denoted by $AS_H(w_t[ob])$, is the set of operations recursively defined by the following rules:

- $w_t[ob]$ is in $AS_H(w_t[ob])$
- If $w_{t'}[ob]$ is in $AS_H(w_t[ob])$ then all operations of the form $r_{t'}[ob']$ where ob' is some object and $r_{t'}[ob'] \rightarrow_{\mathcal{H}} w_[t'][ob]$ are in $AS_H(w_t[ob])$
- If $r_{t'}[ob]$ is in $AS_H(w_t[ob])$ and $r_{t'}[ob]$ reads from $w_{t''}[ob]$ in $\mathcal{H}$, then $w_{t''}[ob]$ is in $AS_H(w_t[ob])$
- No other operations other than those included by the above three rules are in $AS_H(w_t[ob])$

Intuitively, the affects set of a write operation is the set of operations that "affected" (directly or indirectly) the value of the data item written.

Lemma 1 Let $w_{t'}[ob]$ be the operation $r_t[ob]$ read from in a history $\mathcal{H}$. Then $AS_H(r_t[ob]) = \{r_t[ob]\} \cup AS_H(w_{t'}[ob])$.

Proof: Follows from definitions 2 and 3. ■

Lemma 2 Let R be the set of read operation performed by a transaction t before it performs an operation $w_t[ob]$ in a history $\mathcal{H}$. Then $AS_H(w_t[ob]) = \{w_t[ob]\} \cup \bigcup_{p \in R}(AS_H(p))$.

Proof: Follows from definitions 2 and 3. ■

The following lemma states sufficient conditions for determining whether a read (write) operation in two histories reads (writes) the same value from (to) an object.

Lemma 3 Consider two histories H_1 and H_2 which are produced by running transactions on some initial state of the database. Given the above assumptions, two read operations on an object ob, $r_{t_1}[ob] \in H_1$ and $r_{t_2}[ob] \in H_2$, read the same value of ob if:

1. $AS_{H_1}(r_{t_1}[ob]) - \{r_{t_1}[ob]\} = AS_{H_2}(r_{t_2}[ob]) - \{r_{t_2}[ob]\}$

2. $r_{t_1}[ob]$ and $r_{t_2}[ob]$ read from the same operation in H_1 and H_2.

3. Each read operation in $AS_{H_1}(r_{t_1}[ob]) - \{r_{t_1}[ob]\}$ (and hence also in $AS_{H_2}(r_{t_2}[ob]) - \{r_{t_2}[ob]\}$) reads from the same write operation in both histories H_1 and H_2.

Also, given the above assumptions, two write operations on an object *ob*, $w_{t_1}[ob] \in H_1$ and $w_{t_2}[ob] \in H_2$, write the same value to *ob* if:

1. $t_1 = t_2$

2. $AS_{H_1}(w_{t_1}[ob]) = AS_{H_2}(w_{t_2}[ob])$

3. Each read operation in $AS_{H_1}(w_{t_1}[ob])$ (and hence also in $AS_{H_2}(w_{t_2}[ob])$) reads from the same write operation in both histories H_1 and H_2.

Proof: We prove the theorem only for write operations. The proof for the case of read operations is very similar. Let there be n operations in $AS_{H_1}(w_{t_1}[ob]))$. Consider a topological order of the operations in H_1. We can associate an index $1 \leq i_{op} \leq n$ with each operation $op \in AS_{H_1}(w_{t_1}[ob]))$ such that for two distinct operations op_1 and op_2 in $AS_{H_1}(w_t[ob])$, $i_{op_1} < i_{op_2}$ iff op_1 occurs before op_2 in the topological order of H_1. We now prove by induction on the index i_{op} that

1. Every read operation $r_{t'}[ob']$ in $AS_{H_1}(w_{t_1}[ob]))$ reads the same value of *ob* in H_1 and H_2.

2. Every write operation $w_{t'}[ob']$ in $AS_{H_1}(w_{t_1}[ob]))$ writes the same value for *ob* in H_1 and H_2.

Basis. Consider the case when the operation with index 1 is a read operation $r_{t'}[ob']$. In this case, no transaction which did not abort before $r_{t'}[ob']$ was performed could have written on to *ob* in H_1. This is because, if such a write operation did exist, $r_{t'}[ob']$ would not have been the operation with index 1. Also, since $r_{t'}[ob]$ reads from the same operation in both H_1 and H_2, no such write operation could exist in H_2 also. Hence $r_{t'}[ob']$ in both H_1 and H_2 reads the initial state of the object *ob* and hence reads the same state of the object *ob*.

Now consider the case when the operation with index 1 is a write operation $w_{t'}[ob']$. In this case, this operation occurs before the first read operation of t' in H_1 (and by our assumption about transactions, this means that t' does not perform a read operation on the database). This is because if there exists a read operation $r_{t'}[ob'']$ before $w_{t'}[ob']$ in either H_1 or H_2, then $r_{t'}[ob'']$ would have been in $AS_{H_1}(w_{t'}[ob']))$ and would have had a lower index. $w_{t'}[ob']$ also occurs before any read operation of t' in H_2 by assumption 1. Hence, we can infer that $w_{t'}[ob']$ will write the same value to *ob* in both H_1 and H_2 (again by assumption 1).

Induction Hypothesis. Assume that the claim holds for all operations in $AS_{H_1}(w_t[ob]))$ with index less than or equal to k, such that $1 \leq k \leq n-1$.

Induction Step. Consider the operation with index $k+1$. If this operation is a read operation $r_{t'}[ob']$, then we know that $r_{t'}[ob']$ either reads the initial state of ob in both H_1 and H_2 or reads the state of the object written to by some operation $w_{t''}[ob']$ which occurs before $r_{t'}[ob']$ in both H_1 and H_2. In the first case, it is easy to see that the values of ob read by $r_{t'}[ob']$ in H_1 and H_2 are the same. In the second case too, they are the same since we can infer that $w_{t''}[ob']$ writes the same value in both H_1 and H_2, by induction hypothesis.

Now consider the case when the operation is a write operation $w_{t'}[ob']$. By assumption 1 and by the induction hypothesis, we can conclude that all read operations performed by t' which occur before $w_{t'}[ob']$ in H_1 also occur in the same order and see the same values in H_2 and vice-versa. Again by assumption 1, this means that $w_{t'}[ob']$ writes the same value to ob' in both H_1 and H_2.

Since $w_{t_1}[ob]$ is in $AS_{H_1}(w_{t_1}[ob])$ (and hence, also in $AS_{H_2}(w_{t_1}[ob])$), it follows that it writes the same value to ob in both H_1 and H_2. ■

Definition 4 [17] A *polygraph* $P = (N, A, B)$ is a digraph (N, A) together with a set B of bipaths, that is, pairs of arcs - not necessarily in A - of the form $((v, u), (u, w))$ such that $(w, v) \in A$.

Alternatively, a polygraph (N, A, B) can be viewed as a family $\mathcal{D}(N, A, B)$ of digraphs. A digraph (N, A') is in $\mathcal{D}(N, A, B)$ iff $A \subseteq A'$ and for each bipath $(a_1, a_2) \in B$, A' contains at least one of a_1, a_2.

Definition 5 [17] A polygraph (N, A, B) is *acyclic* iff there is an acyclic digraph in $\mathcal{D}(N, A, B)$.

Definition 6 The *polygraph associated with a transaction t in a history $\mathcal{H}$*, $P_{\mathcal{H}}(t)$, is the polygraph (N, A, B) where:

1. $N = LIVE_{\mathcal{H}}(t)$

   The set of nodes is the set of all the transactions that t directly or indirectly reads from in the history $\mathcal{H}$.

2. $A = \{(t', t'') | t', t'' \in N \wedge \exists ob((t', ob, t'') \in READS\_FROM_{\mathcal{H}})\}$

   There is an arc from a transaction t' to a transaction t'' iff t'' reads the value of some object from t' in the history $\mathcal{H}$.

3. $B = \{((t', t''), (t''', t')) | t', t'', t'' \in N \wedge \exists ob((t'', ob, t''') \in READS\_FROM_{\mathcal{H}} \wedge w_{t'}[ob] \in \mathcal{H})\}$

   There is a bipath of the form $((t', t''), (t''', t'))$ iff t' writes to some data item which t''' reads from t'' in $\mathcal{H}$.

Intuitively, a transaction polygraph associated with a transaction t is acyclic if t is serializable with respect to all the transactions it directly or indirectly reads from. When the history is evident from the context, we denote the polygraph associated with a transaction t by $P(t)$. The following two lemmas illustrate the relationship between the polygraph associated with a transaction in a history and the existence of serial orders involving that transaction.

Lemma 4 Consider a transaction t in a history $\mathcal{H}$. If $P_{\mathcal{H}}(t)$ is acyclic, then there exists a serial ordering of a subset of transactions occurring in $\mathcal{H}$, that includes t, such that each transaction in the serial ordering performs the same read operations and each read operation reads from the same write operation in the serial ordering as it does in $\mathcal{H}$.

Proof: Let $P_{\mathcal{H}}(t) = (N, A, B)$. Since $P_{\mathcal{H}}(t)$ is acyclic, there must exist a topological order of all the transactions in N which satisfies all the edges and bipaths of the polygraph. Consider any such topological order, say TO. We now prove by induction on the position of a transaction in TO that each transaction in N performs the same sequence of operations in the serial order TO as it does in the history $\mathcal{H}$. Further, we prove that each operation performed by a transaction in TO has the same affects-set as the corresponding operation in $\mathcal{H}$ and read operations in the affects-sets reads from the same write operations in both TO and $\mathcal{H}$. This implies the above lemma.

Basis: Consider the first transaction in the sequence TO. This transaction t' is either t_0 or a transaction which does not read any value from the database (for if it did, there would be a write-read edge from the transaction it read from to itself and it could not be the first transaction in TO). In either case, we know that t' does not perform any read operation and by assumption 1, we know that t' performs the same sequence of write operations in TO as it does in the history $\mathcal{H}$. Since, the affects-set of write operations of t' is ϕ in either case, the basis is proved.

Induction Hypothesis: Assume that the claim is true for all transactions in position k in the sequence TO, where $1 \leq i$.

Induction Step: Consider the transaction t' at position $k+1$ in the sequence TO. By induction hypothesis, all transactions with index $\leq k$ perform operations in the same order in TO as they did in the history $\mathcal{H}$. Further, the affects-set of the each operation in performed by transactions with index $\leq k$ in TO is the same as the affects-set of the corresponding operation in the history $\mathcal{H}$ and each read operation in affects-set read from the same write operation in both TO and $\mathcal{H}$. We now prove by induction on the index of the sequence of operations performed by t' in $\mathcal{H}$ that t' performs the same sequence of operations in TO and that each operation performed by t' in $\mathcal{H}$ has the same affects-set as the corresponding operation performed by t' in TO, and the read operations in the affect-sets read from the same write operations in both $\mathcal{H}$ and TO.

Basis: Suppose the first operation performed by t' in $\mathcal{H}$ is $r_{t'}[ob]$. In this case, by assumption 1, the first operation performed by t' in TO is also $r_{t'}[ob]$. If $r_{t'}[ob]$ reads from $w_{t''}[ob]$ in $\mathcal{H}$ then, by the definition of $P_{\mathcal{H}}(t')$, $r_{t'}[ob]$ and induction hypothesis (of outer induction), it also reads from $w_{t''}[ob]$ in TO. This is because all transactions which write onto ob in $\mathcal{H}$ either occur before t'' or after t' in the topological order and each. By induction hypothesis (of the outermost induction), we know that $w_{t''}[ob]$ in both $\mathcal{H}$ and TO have the same affects set and that each read operation in the affects-set reads from the same write operation in both histories. Thus, by lemma 1, we know that $r_{t'}[ob]$ has the same affects-set in both $\mathcal{H}$ and TO, and that each read operation in the affects-set reads from the same write operation in both $\mathcal{H}$ and TO.

Suppose the first operation performed by t' in $\mathcal{H}$ is $w_{t'}[ob]$. In this case, by assumption 1, the first operation performed by t' in TO is also $w_{t'}[ob]$. Thus, the affects-set of $w_{t'}[ob]$ is ϕ in both cases (by lemma 2), thus proving the basis.

Induction Hypothesis: Assume that the claim holds for all operations performed by transaction t' with index $\leq l$.

Induction Step: Let $r_{t'}[ob]$ the $(l+1)^{st}$ operation performed by transaction t' in history $\mathcal{H}$. By induction hypothesis (of inner induction), lemma 3 and assumption 1, we can infer that t' performs $r_{t'}[ob]$ as the $(l+1)^{st}$ in TO. By reasoning similar to the basis (of inner induction), we can conclude that $r_{t'}[ob]$ has the same affects-set in both $\mathcal{H}$ and TO, and that each read operation in the affects-set reads from the same write operation in both $\mathcal{H}$ and TO.

If $w_{t'}[ob]$ is the $(l+1)^{st}$ operation performed by t' in history $\mathcal{H}$, then by induction hypothesis (of inner induction), lemma 3 and assumption 1, we can infer that t' performs $w_{t'}[ob]$ as the $(l+1)^{st}$ in TO. Also, by induction hypothesis (of inner induction) and lemma 2, we can infer that $w_{t'}[ob]$ has the same affects-set in both $\mathcal{H}$ and TO, and that each read operation in the affects-set reads from the same write operation in both $\mathcal{H}$ and TO, thus extending the induction. ■

Lemma 5 Consider a transaction t in a history $\mathcal{H}$. If $P_{\mathcal{H}}(t)$ is not acyclic, then there exists no serial ordering including the transactions in $LIVE_{\mathcal{H}}(t)$ such that each transaction in $LIVE_{\mathcal{H}}(t)$ performs the same set of read operations in the serial order and in $\mathcal{H}$ and each such read operation reads from the same write operation in the serial order and in $\mathcal{H}$.

Proof: Let $P_{\mathcal{H}}(t)$ be an acyclic polygraph (N, A, B). Consider any serial ordering S including the transactions in N. There are now two cases: (a) the serial ordering S is not consistent with a topological sort of the digraph (N, A) or (b) the serial ordering S is consistent with a topological sort of the digraph (N, A). We consider each case in turn.

In case (a), there exist two transactions t' and t'' in N such that t'' reads some object *ob* from t' in $\mathcal{H}$ (causing the edge (t', t'') to be present in A) and t'' occurs before t' in the serial order S. Thus, either t'' does not read *ob* in S or it does not read the value of *ob* written by t'.

Now consider case (b). In this case, there exist some transaction t' in the serial order S that violates a bipath of the form $((t', t''), (t''', t'))$. Thus, t' occurs between t'' and t''' in the serial order S. Further, t''' reads the value of some object *ob* from t'', that is also written to by t', in the history $\mathcal{H}$. Now, we have two cases. Either (i) some transaction occurring before t' (including t') in the serial order S that also occurs in $\mathcal{H}$, say t'''', does not perform the same set of read operations as in $\mathcal{H}$ or some of the read operations performed by t'''' read from different write operations in S and $\mathcal{H}$ or (ii) all transactions before t' (including t') that occur in $\mathcal{H}$ perform the same set of read operations in S and $\mathcal{H}$ and each read operation performs reads from the same write operation in S and $\mathcal{H}$. In sub case (i), the claim holds. In sub case (ii), by lemma 3, we can infer that each read operation performed by t' in S (and hence in $\mathcal{H}$) reads the same value as in $\mathcal{H}$. Thus, by assumption 1, t' writes to *ob* in S. Hence t''' cannot read the value of *ob* from t'' in the serial order S. This proves the claim. ■

The following lemma states a sufficient condition that the scheduler can use to determine whether a history satisfies formal requirements 1 through 3.

Lemma 6 A scheduler can determine that a history $\mathcal{H}$ satisfies requirements 1 through 3 if:

1. $\mathcal{H}_{update}$ is view serializable [25, 5].

2. For every read only transaction t_R in the history $\mathcal{H}$, $P_{\mathcal{H}}(t_R)$ is acyclic.

Proof: It is easy to see that part 1 implies formal requirements 1 and 2. This is because, if $\mathcal{H}_{update}$ is view serializable [5], the final state of the database after $\mathcal{H}_{update}$ is the same as the state after running the transactions in $\mathcal{H}_{update}$ in some serial order (this satisfies requirement 1) and further, each read in $\mathcal{H}_{update}$ reads the same value in $\mathcal{H}_{update}$ as it does in the serial order (this satisfies requirement 2). Further, if $\mathcal{H}_{update}$ is view serializable, it also satisfies formal requirement 3 for update transactions. Finally, from lemma 4 and lemma 3, we can infer that formal requirement 3 is satisfied for read only transactions in $\mathcal{H}$. ∎

The following lemma states a necessary condition that should hold for all histories determinable by a scheduler to satisfy requirements 1 through 3.

Lemma 7 A scheduler can determine that a history $\mathcal{H}$ satisfies requirements 1 through 3 only if:

1. $\mathcal{H}_{update}$ is view serializable.

2. For every read only transaction t_R in the history $\mathcal{H}$, $P_{\mathcal{H}}(t_R)$ is acyclic.

Proof: Consider the scenario where each transaction in the history $\mathcal{H}$, on inputs different from what they read in $\mathcal{H}$, (a) writes to a data item not accessed in $\mathcal{H}$ a value that is different from the initial value and further, (b) writes values to all objects accessed in $\mathcal{H}$ that are different from those written by any transaction in $\mathcal{H}$. Also, assume that each write operation to an object in the history $\mathcal{H}$ writes a different value. From the assumptions made, this is a valid scenario.

Consider the case where $\mathcal{H}_{update}$ is not view serializable. This implies that there is no serial ordering of update transactions such that (a) each read operation in the serial order reads from the same write operation as in $\mathcal{H}$ and (b) the final write operations to objects in both $\mathcal{H}$ and the serial order are the same. In case part (a) is violated, then from the scenario described above, we know that in any serial order of update transactions, a variable that was not accessed in $\mathcal{H}$ is written to, and the value written is different from the initial value. Thus, the final state of the database after $\mathcal{H}$ will not be the same as the final state on executing update transactions in some serial order. This violates formal requirement 1. Now consider the case where part (b) is violated. Again, by the scenario, we know that write operations to objects by different transactions write different values. Thus, the final state of the database after $\mathcal{H}$ will not be the same as the final state on executing update transactions in some serial order. This again violates formal requirement 1.

Now, consider the case where $P_{\mathcal{H}}(t_R)$, where t_R is a read only transaction in $\mathcal{H}$, is not acyclic. From the scenario described above, it is easy to see that t_R can read the same value of objects as in a serial history S involving transaction transactions in $\mathcal{H}$ iff each transaction in $RAS_{\mathcal{H}}(t)$ performs the same set of read operations and each read operation reads from the same write operation in S and $\mathcal{H}$. However, by lemma 5, we know that this is impossible. Thus, formal requirement 3 is violated in this case for the transaction t_R. ∎

¿From Lemmas 6 and 7, we can infer the following theorem.

Theorem 11 A scheduler can determine that a history $\mathcal{H}$ satisfies requirements 1 through 3 iff both of the following conditions hold:

1. $\mathcal{H}_{update}$ is view serializable [25]
2. For every read-only transaction t_R in the history $\mathcal{H}$, $P_{\mathcal{H}}(t_R)$ is acyclic.

B Proofs of Theorems in Section 3.1

Proof of Theorem 2: It is easy to see that determining whether a history $\mathcal{H}$ is legal is in **NP**. This is because checking whether $\mathcal{H}_{update}$ is view serializable is in **NP** [17]. Also, determining $P_{\mathcal{H}}(t_R) = (N, A, B)$ for every read only transaction t_R in $\mathcal{H}$ can be done in polynomial time and determining whether a digraph in $\mathcal{D}(N, A, B)$ is acyclic can be done in non-deterministic polynomial time.

We now prove that determining whether a history is legal is **NP-Hard**. This is done by a reduction from view serializability, which is **NP-Complete** [17]. Consider any history $\mathcal{H}$. Modify the history such that each read transaction t in $\mathcal{H}$ such that it writes onto a unique data item ob_t. Now all transactions are update transactions and it is easy to see that $\mathcal{H}$ is view serializable iff the modified history is legal. ■

We now proceed to prove Theorem 3 which states that the problem of determining legal histories $\mathcal{H}$ such that $\mathcal{H}_{update}$ is serial is **NP-Complete**.

Definition 7 [17] A clause in a boolean formula that is in conjunctive normal form is *mixed* iff it contains both variable and negations of variables.

Definition 8 [17] A boolean formula is *non-circular* iff it is in conjunctive normal form and at most one of the occurrences of each variable is in a mixed clause.

We now describe the construction of a polygraph associated with a non-circular boolean formula φ, $P_\varphi = (N, A, B)$, as follows. This construction follows the one give in [17]. Let ψ have m clauses $C_1, ..., C_m$ involving n variables $x_1, ..., x_n$. Let each clause C_i consist of two or three literals where the j^{th} literal is λ_{ij} and each literal is either a variable or the negation of a variable. N contains the nodes $a_{x_j}, b_{x_j}, c_{x_j}$ for each variable x_j and y_{ik}, z_{ik}, $k = 1..m_i$ for each clause C_i with m_i literals. For each variable x_j, we add the arc (a_{x_j}, b_{x_j}) to A and the bipath $((b_{x_j}, c_{x_j}), (c_{x_j}, a_{x_j}))$ to B. If $\lambda_{ik} = x_j$, we add the arcs (c_{x_j}, y_{ik}) and (b_{x_j}, z_{ik}) to A and the bipath $((z_{ik}, y_{ik}), (y_{ik}, b_{x_j}))$ to B. If $\lambda_{ik} = \bar{x_j}$, then we add the arcs (z_{ik}, c_{x_j}) and (y_{ik}, a_{x_j}) to A and the bipath $((a_{x_j}, z_{ik}), (z_{ik}, y_{ik}))$ to B.

Lemma 8 Consider a non-circular boolean formula φ and a variable x_j that occurs in φ. Let $P_\varphi = (N, A, B)$ be the polygraph associated with φ. φ is satisfiable with the value of x_j set to *false* iff there exists an acyclic digraph $(N, A') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ such that $(b_{x_j}, c_{x_j}) \in A'$.

Proof: This proof is adapted from [17]. Suppose there exists an acyclic digraph $(N, A') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ such that $(b_{x_j}, c_{x_j}) \in A'$. For each variable x_i, exactly one of the edges (c_{x_i}, a_{x_i}) and (b_{x_i}, c_{x_i}) is in A'. Consider the interpretation that if $(c_{x_i}, a_{x_i}) \in A'$ then x_i is assigned the value *true* and $(b_{x_i}, c_{x_i}) \in A'$ then x_i is assigned the value *false*. Note that this implies that x_j is assigned the value *false*. Now if a literal λ_{ik} is given a value *false* by this assignment, the arc (z_{ik}, y_{ik}) is in A'. This is because, otherwise a cycle of the form $(c_{x_l}, y_{ik}, b_{x_l})$ if $\lambda_{ik} = x_l$ or $(z_{ik}, c_{x_l}, a_{x_l})$ if $\lambda_{ik} = \bar{x}_l$ would exist in (N, A'). Thus, (N, A') will not have a cycle of the form $(z_{i1}, y_{i1}, z_{i2}, ..., y_{i3})$ only if at least one literal in each clause is assigned the value *true*. Since we know that x_j is assigned the value *false*, the if part of the lemma follows.

Now suppose that φ is satisfiable with the value of x_j set to *false*. Let T be some such truth assignment. We now construct an acyclic digraph $(N, A') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ as follows. A' is the set of all the arcs in A along with the arcs (c_{x_l}, a_{x_l}) if $T(x_l) = true$, (b_{x_l}, c_{x_l}) if $T_(x_l) = false$, (z_{ik}, y_{ik}) if $T(\lambda_{ik}) = false$, (y_{ik}, b_{x_l}) if $\lambda_{ik} = \bar{x}_l$ and $T(x_l) = true$, and (a_{x_l}, z_{ik}) if $\lambda_{ik} = x_l$ and $T(x_l) = false$. It is easy to see that $(N, A') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$. We now show that it is also acyclic. Since φ is non-circular, (N, A) is acyclic. This is because node sets corresponding to clauses containing only variables or their negations correspond to node sets with only incoming or only outgoing arcs respectively. Since no variable appears in more than one mixed clause, no two node sets corresponding to mixed clauses are reachable from each other. Thus, (N, A) is acyclic. The arcs in $A' - A$ can introduce cycles in the digraph only by the introducing cycles of the form $(z_{i1}, y_{i1}, ..., y_{i3})$. However, this would mean that the clause C_i has no true literals, a contradiction. ■

Lemma 9 Given a satisfiable non-circular boolean formula φ and a truth assignment T for φ, an acyclic digraph $(N, A') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$, where (N, A, B) is the polygraph associated with φ, can be determined in polynomial time.

Proof: The proof is similar to the proof of the only if part of lemma 8. ■

Proof of Theorem 3: By an argument similar to that given in theorem 2, it is easy to see that the problem is in **NP**. We now prove that the problem is **NP-Hard** by a reduction from $3 - SAT$[4], which is known to be **NP-Complete** [10]. The proof borrows ideas from the proof of the **NP-Completeness** of view serializability that appeared in [17].

Consider a formula ψ in conjunctive normal form, with two or three literals in every clause. Now construct another formula ψ' by adding a new literal x, that does not occur in ψ, as a disjunct to every clause in ψ. It is easy to see that (a) ψ' is in conjunctive normal form (b) ψ' is satisfiable (by setting x to *true*) and (c) ψ is satisfiable iff ψ' is satisfiable with the value of x set to *false*. ψ' can now be rewritten as a formula, ψ'', in which each clause has three literals by rewriting each clause of the form $(a \vee b \vee c \vee d)$ as the conjunct $(a \vee b \vee z) \wedge (c \vee d \vee \bar{z})$, where z is a new variable. Again it is easy to see that (a) ψ'' is in conjunctive normal form with three literals in every clause (b) ψ'' is satisfiable (by setting x to *true*) and (c) ψ is satisfiable iff ψ'' is satisfiable with the value

[4] $3 - SAT$ is the problem of determining whether a boolean formula in conjunctive normal form, with three literals in a clause, is satisfiable

of x set to *false*.

Let m be the number of occurrences of a variable z in the formula ψ'' and let $z_2, z_3, ..., z_m$ be new variables. Replacing the second occurrence of z by z_2, the third occurrence of z by z_3, etc. and adding the clauses $(z \vee z_2) \wedge (\bar{z} \vee \bar{z_2}) \wedge (z_2 \vee z_3) \wedge (\bar{z_2} \vee \bar{z_3})$.... Repeating this for all variables we observe that (a) the resulting formula, say φ, is non-circular (b) φ is satisfiable (by setting x to *true*, x_2 to *false*, x_3 to *true*, x_4 to *false* etc. and giving arbitrary truth assignments for other variables) and (c) ψ is satisfiable iff φ is satisfiable with the value of x set to *false*. It can be verified that the above steps can be done in polynomial time.

Now let $P_\varphi = (N, A, B)$ be the polygraph associated with the non-circular boolean formula φ. Construct a new polygraph $P'_\varphi = (N', A', B')$ such that $N' = N \cup \{t_R\}$, $A' = A \cup \{(y, t_R) | y \in N\}$ and $B' = B \cup \{((a_x, c_x), (t_R, a_x))\}$. We now claim that P'_φ is acyclic iff there exists an acyclic digraph $(N, A'') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ such that $(b_x, c_x) \in A''$. For the if part, assume that there exists an acyclic digraph $(N, A'') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ such that $(b_x, c_x) \in A''$. Now consider the digraph (N', A'''), where $A''' = A' \cup \{(a_x, c_x)\}$. It is easy to see that (N', A''') is acyclic because the addition of the node t_R and the arcs (y, t_R) do not harm the acyclicity of (N, A''). Further, the arc (a_x, c_x) cannot create a cycle because the arcs (a_x, b_x) and (b_x, c_x) are already present in A''. For the only if part, assume that P'_φ is acyclic. Thus, there exists an acyclic digraph $(N', A''') \in \mathcal{D}(\mathcal{N}', \mathcal{A}', \mathcal{B}')$. The arc $(a_x, c_x) \in A'''$ for if not, there would be a cycle of the form (a_x, t_R, a_x). Since (N', A''') is acyclic, the arc $(c_x, a_x) \notin A'''$ (for otherwise, there would be a cycle of the form (c_x, a_x, c_x)). Thus, $(b_x, c_x) \in A'''$. Since the set of arcs and bipaths in P'_φ is a superset of the set of arcs and bipaths in P_φ, it is easy to see that there exists an acyclic digraph $(N, A'') \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ such that $(b_x, c_x) \in A''$.

¿From lemma 8 and the above result, we can infer that P'_φ is acyclic iff φ is satisfiable with the value of x set to *false*, i.e., iff ψ is satisfiable. We now proceed to construct a history $\mathcal{H}$ such that $\mathcal{H}_{update}$ is serial and $P_\mathcal{H}(t_R) = P'_\varphi$, where t_R is the only read only transaction in $\mathcal{H}$. Since we know that ψ is satisfiable iff P'_φ is acyclic, by the definition of legal histories, we can infer that ψ is satisfiable iff $\mathcal{H}$ is a legal history. This would complete the reduction.

We construct the history $\mathcal{H}$ as follows. Let all the nodes in P' correspond to distinct transactions in $\mathcal{H}$. To construct the set of read and write operations performed by each transaction, we initially start with the empty set for each transaction. For each arc (u, v) in A', we add the write operation $w_u[y_{uv}]$ to the operation set of u and the read operation $r_v[y_{uv}]$ to the operation set of v. For each bipath $((t, u), (v, t))$, we add the write operation $w_t[y_{uv}]$ to the operation set of t. Since we know a truth assignment for φ, we can determine an acyclic digraph $(N, A) \in \mathcal{D}(\mathcal{N}, \mathcal{A}, \mathcal{B})$ in polynomial time (by lemma 9). Let TO be a topological sort of this acyclic digraph. Now construct the history $\mathcal{H}$ such that transactions are ordered as per the order TO and where all read operations performed by a transaction precede all write operations performed it. Also, add the read operation $r_{t_R}[y_{a_x t_R}]$ immediately after the last write operation of transaction a_x and the remaining read operations of transaction t_R at the end of the history.

It is now easy to see that $\mathcal{H}_{update}$ is serial and that $P'_\varphi = P_\mathcal{H}(t_R)$. Further this construction takes polynomial time. This completes the proof. ■

C Proofs of Theorems in Section 3.2

The following definition, in effect, defines the serialization graph of all the transactions in the history $\mathcal{H}$ that a transaction t directly or indirectly read from.

Definition 9 The *serialization graph associated with a transaction t in a history H*, $S_{\mathcal{H}}(t)$, is the digraph (N, A) where:

1. $N = LIVE_{\mathcal{H}}(t)$

   The set of nodes is the set of all live transactions with respect to t in the history $\mathcal{H}$.

2. $A = X \cup Y \cup Z$ where:

   $X = \{(t', t'') | t', t'' \in N \wedge \exists ob((t', ob, t'') \in READS\_FROM_{\mathcal{H}})\}$

   There is an arc from a transaction t' to a transaction t'' if t'' reads the value of some object from t' in the history $\mathcal{H}$.

   $Y = \{(t', t'') | t', t'' \in N \wedge \exists ob w_{t'}[ob]$ occurs before $w_{t''}[ob]$ in $\mathcal{H}\}$

   There is an arc of the form (t', t'') if t' writes to some data item later written to by t'' in $\mathcal{H}$.

   $Z = \{(t', t'') | t', t'' \in N \wedge \exists ob r_{t'}[ob]$ occurs before $w_{t''}[ob]$ in $\mathcal{H}\}$

   There is an arc from a transaction t' to a transaction t'' if t'' writes onto the same object that t' earlier read from.

It is easy to see that $S_{\mathcal{H}}(t_R)$ is acyclic iff t_R is serializable with respect to all the update transactions it directly or indirectly read from (see [5] for the related proof that a history is conflict serializable iff the corresponding serializability graph is acyclic).

Lemma 10 Let $\mathcal{H}$ be a history and let t_R be a read-only transaction in $\mathcal{H}$. Then $P_{\mathcal{H}}(t_R)$ is acyclic if $S_{\mathcal{H}}(t_R)$ is acyclic.

Proof: Consider an acyclic serialization graph $S_{\mathcal{H}}(t_R)$ associated with a transaction t_R in a history $\mathcal{H}$. By definitions 6 and 9, we can see that the set of nodes are the same in both $S_{\mathcal{H}}(t_R)$ and $P_{\mathcal{H}}(t_R)$. Also, we can see that if an arc (t', t'') is in $P_{\mathcal{H}}(t_R)$ then the arc (t', t'') is also in $S_{\mathcal{H}}(t_R)$. Further, we can infer that if a bipath $((t', t''), (t''', t'))$ is in $P_{\mathcal{H}}(t_R)$, then either the arc (t', t'') or the arc (t''', t') is in $S_{\mathcal{H}}(t_R)$. Thus, by definition 5, it follows that $P_{\mathcal{H}}(t_R)$ is acyclic if $S_{\mathcal{H}}(t_R)$ is acyclic. ∎

Proof of Theorem 4: We first prove that the set of histories accepted by $APPROX$ is a subset of the set of legal histories. In order to see this, consider a history $\mathcal{H}$ that is accepted by $APPROX$. This implies that $\mathcal{H}_{update}$ is conflict serializable. But because the set of view serializable histories is a superset of the set of conflict serializable histories [17], this in turn implies that $\mathcal{H}_{update}$ is view serializable. Since $\mathcal{H}$ is accepted by $APPROX$, this means that for each read transaction $t_R \in \mathcal{H}$, $S_{\mathcal{H}}(t_R)$ is acyclic. This, by lemma 10, implies that $P_{\mathcal{H}}(t_R)$ is acyclic. This, $\mathcal{H}$ is a legal history.

Since the choice of $\mathcal{H}$ was arbitrary, we can infer that the set of histories accepted by *APPROX* is included the set of legal histories.

In order to prove that the inclusion is proper, consider the following history:

$r_{t_1}[ob_1]\ r_{t_2}[ob_2]\ w_{t_1}[ob_3]\ w_{t_2}[ob_3]\ w_{t_2}[ob_4]\ w_{t_1}[ob_4]\ w_{t_3}[ob_3]\ w_{t_3}[ob_4]$

This history is legal but is not accepted by *APPROX*. ■

Proof of Theorem 5: Given a history $\mathcal{H}$, the subhistory $\mathcal{H}_{update}$ can be determined in polynomial time. Further, determining whether $\mathcal{H}_{update}$ is conflict serializable can again be done in polynomial time [5]. For every read transaction t_R in $\mathcal{H}$, the serialization graph $S_{\mathcal{H}}(t_R)$ can be determined in polynomial time. Further, there exist polynomial time algorithms for testing the acyclicity of a graph [8]. Thus, *APPROX* is in **P**. ■

D Proofs of Theorems in Section 3.3.1

There is a technicality in what we mean by the history of operations in the broadcast disk sense where operations are not performed at one place but are performed at the clients and the server and where the client transactions read only committed values. We assume that in the global history, a read operation on an object performed by a read-only client transaction t_R occurs right after the commit of the last update transaction that wrote to that object or immediately after the previous read operation performed by t_R, whichever is later. It is easy to see that this interpretation of histories is consistent with the semantics of read operations performed at clients in broadcast disk environments and preserves the notion of conflicts. The ordering of write operations is not important in our analysis.

Proof of Theorem 6: Assume that a read-only transaction t_R is allowed to commit. Since all the update transactions are conflict serializable at the server, we know that the subhistory consisting only of transactions in $LIVE(t_R) - \{t_R\}$ is conflict serializable. If we now add the read operations performed by t_R to this subhistory, the only way this history would not be conflict serializable (i.e., $S(t_R)$ is not acyclic) is if there exists some write operation performed by some transaction $t' \in LIVE(t_R) - \{t_R\}$ on an object ob_i after t_R performs a read operation on ob_i.

However, since t' is in $LIVE(t_R) - \{t_R\}$, there must exist some object ob_j and a transaction t'' such that $t'' \in LIVE(t_R) - \{t_R\}$, $t'' \in LIVE(t')$ and t'' writes to ob_j and t_R reads this value of ob_j. Since t' wrote onto ob_i after t_R read ob_i, t' must have committed in a cycle (say c_2) that is greater than or equal to the cycle (say c_1) in which t_R read ob_i (for if not, t_R would have read the value of ob written by t'). Also, at the cycle (say c_3) at which t_R reads ob_j, t' must have committed (because t_R reads the committed value of ob_j written by t'' and t'' directly or indirectly reads from t'). Thus, by the definition of the matrix C, $c_1 \leq c_2 \leq C(ob, ob') < c_3$ at cycle c_3. Thus, by the protocol, at cycle c_3, the transaction t_R would have been aborted because $C(i,j) \not< c_1$, a contradiction. Thus, $S(t_R)$ is acyclic. ■

Proof of Theorem 7: Assume that $S(t_R)$ is acyclic. Then the subhistory consisting of transactions in $LIVE(t_R)$ is conflict serializable. Now, for the sake of contradiction, assume that the protocol detailed above aborts t_R. From the working of the protocol, we can infer that t_R

must have read an object ob during a cycle c_1 and some transaction $t' \in LIVE(t_R) - \{t_R\}$ that committed at some cycle $c_2 \geq c_1$ must have written to ob. Further, the transaction t_R must have attempted to read an object ob' at some cycle $c_3 > c_2$ and the last transaction (say t'') that wrote onto ob' before the begin of cycle c_3 must have directly or indirectly read from t'. However, this would mean that there exists arcs of the form $t_R \rightarrow t'$, $t' \rightarrow^* t''$, $t'' \rightarrow t_R$ which causes a cycle in the serialization graph $S(t_R)$, a contradiction. Thus, the protocol allows t_R to commit. ∎

Proof of Theorem 8: We prove the by induction of the position of a committed transaction transaction t in a serialization order that the C matrix on being incrementally updated as per the algorithm has the desired values in its entries.

Basis: Initially, when no transaction is considered, the C matrix has all entries set to 0. This trivially preserves the semantics of C.

Induction Hypothesis: Assume that the claim is true for a transaction that is at position k in the serialization order.

Induction Step: Consider the transaction t' at position $k+1$ in the serialization order that commits at cycle c_2. By induction hypothesis, we know that C_{old} satisfies the semantics of the matrix for all transactions that committed before t' in a serial order. For all objects ob_j not in WS_t, the entries $C_{new}(i,j)$ should not change because t does not write to object ob_j. This is preserved by the algorithm.

For all objects ob_i, ob_j that are in WS_t, the entry $C(i,j)$ should be set to the time of the last write to ob_i by a committed transaction that t directly or indirectly reads from (in this case t itself). Thus, $C_{new}(i,j)$ should be set to c_2 as in the algorithm.

For all objects ob_j that are present in WS_t, the values written by t are dependent on the values of objects in RS_t that are read by t. Thus the entry $C_{new}(i,j)$, where $ob_i \notin WS_t$, by induction hypothesis, needs to be set to the maximum value of the entries $C_{new}(i,k)$, where $ob_k \in RS_t$. This is precisely what is done in the algorithm.

Since the above three cases are mutually exclusive and exhaustive, the lemma follows. ∎

Proof of Theorem 9: The idea behind this proof is to show how all C matrices satisfying a condition (the number of such C matrices being quadratic in the number of objects in the database) have a corresponding history of execution. This would prove that at least a quadratic (in the number of objects) number of bits have to be transmitted, in the worst case, to represent the C matrix during each broadcast cycle.

Consider a database with n data items, where n is an odd number (the case where n is even can be dealt with in a similar way), and assume that the objects have ids ob_1 through ob_n. Now, consider a C matrix that is partially specified as follows:

Each entry $C(i,j), 1 \leq i,j \leq \frac{n-1}{2}$ takes on arbitrary values as long as the constraint $C(i,j) \leq C(j,j)$ is satisfied.

It is easy to see that there are at least $(\frac{n-1}{2} \times \frac{n-1}{2} - \frac{n-1}{2}) \times \log(max\_cycles) = \frac{n^2-4\times n+3}{4} \times \log(max\_cycles)$ partial specifications of the C matrix satisfying the above criterion. This is because each entry $C(i,j), 1 \leq i,j \leq \frac{n-1}{2}, i \neq j$ can take on any value from 0 to $max\_cycles - 1$ and each entry $C(j,j), 1 \leq j \leq \frac{n-1}{2}$ can take on the value $max\_cycles - 1$ without violating the constraint.

We now show that for each such partial specification of the C matrix, there exists a fully specified C matrix (with all entries $C(i,j)$ specified) such that it is the same as the C matrix constructed by *F-Matrix* for some history of execution of transactions. This would mean that at least $\frac{n^2-4\times n+3}{4} \times \log(max\_cycles)$ bits would have be transmitted in each cycle, in the worst case, to distinguish between these different histories. This would prove the theorem.

Given a partial specification of the C matrix, we now incrementally construct a history such that the C matrix constructed from this history is consistent with the partial specification. We construct this history by specifying the history of execution of transactions during each broadcast cycle.

The main idea in the construction is to have a "twin" for each object $ob_k, 1 \leq k \leq \frac{n-1}{2}$, the twin being the object ob_{n-k}. The twin of each object is used to avoid unnecessary dependencies between transactions that could arise during the construction of the history. We represent the twin of an object ob_k by the symbol $\bar{ob_k}$.

For each entry in the C matrix of the form $C(i,j)$ where $1 \leq i,j \leq \frac{n-1}{2}$ and $i \neq j$, we add a new transaction t performing the operations $r_t[\bar{ob_j}]\ w_t[ob_i]\ w_t[\bar{ob_j}]\ c_t$ to the end of the history for broadcast cycle $C(i,j)$. This transaction, intuitively, ensures that the value of $\bar{ob_j}$ at the end of the history for cycle $C(i,j)$ depends on a transaction that writes onto object ob_i during cycle $C(i,j)$. Also, the $r_t[\bar{ob_j}]$ operation ensures that the previous dependencies of $\bar{ob_j}$ are not destroyed (thus, the effects of a similar transaction added for an entry $C(k,j) < C(i,j)$ are not lost).

To complete the construction, for each entry $C(j,j), 1 \leq j \leq \frac{n-1}{2}$, we add a new transaction t' performing the operations $r_{t'}[\bar{ob_j}]\ w_{t'}[ob_j]\ c_{t'}$ to the end of the history for cycle $C(j,j)$. This transaction establishes the dependency between an object and its twin. Thus, the latest value of ob_j depends on the same set of transactions that write onto other objects as the latest value of $\bar{ob_j}$ does. It is easy to see from this construction that each entry $C(i,j), 1 \leq i,j \leq \frac{n-1}{2}$ in the C matrix constructed by *F-Matrix* has the same value as in the partially specified C matrix, thus proving the theorem. ∎

E Proof of Theorem in Section 3.3.2

Proof of Theorem 10: We prove by induction on the number of operations performed by a client read-only transaction t_R that t_R commits under implementation *R-Matrix* only if t_R is conflict serializable with respect to all the update transactions from which it directly or indirectly reads from (i.e., the transactions in $LIVE(t_R)$ are conflict serializable). This would prove the lemma.

Basis: For the case where a client read-only transaction t_R performs just one operation, the transactions in $LIVE(t_R)$ is trivially conflict serializable (because all update transactions are conflict serializable at the server).

Induction Hypothesis: Assume that the claim is true for all transactions with k or fewer read operations.

Induction Step: Consider a client read-only transaction with $k+1$ operations, with the $(k+1)^{th}$

operation being invoked on an object *ob*. If *R-Matrix* allows the transaction to commit, then by induction hypothesis, we can infer that the first k operations of the transaction are serializable with respect to all the update transactions the first k operations directly or indirectly read from. Now assume that there exists a cycle in the serialization graph of transactions in $LIVE(t_R)$. This would mean that there is an arc from t_R to a transaction $t \in LIVE(t_R) - \{t_R\}$ (because all update transactions are conflict serializable at the server). But, this in turn would mean that some object read during the first k operations of t_R was updated by t after t_R performed a read operation on the object and further t directly or indirectly affected the value of *ob*. This, however, implies that the value of object *ob* changed after t_R performed its first read operation. The last two statement imply that t_R would not have been allowed to commit under *R-Matrix*, a contradiction. ■