An Abstract Model of Database Concurrency Control Algorithms

by

Michael J. Carey

ERL MEMORANDUM NO. M83/6

14 January 1983

ELECTRONICS RESEARCH LABORATORY

An Abstract Model of Database Concurrency Control Algorithms

Michael J. Carey

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

An abstract model of concurrency control algorithms is presented. The model facilitates implementation-independent descriptions of various algorithms, allowing them to be specified in terms of the information that they require, the conditions under which blocking or restarts are called for, and the manner in which requests are processed. The model also facilitates comparisons of the relative storage and CPU overheads of various algorithms based on their descriptions. Results are given for single-site versions of two-phase locking, basic timestamp ordering, and serial validation. Extensions which will allow comparisons of multiple version and distributed algorithms are discussed as well.

1. Introduction

Considerable algorithm development has occurred in the area of concurrency control for both single-site and distributed database systems. The vast majority of the proposed algorithms are based on one of three mechanisms: *locking* [Mena78, Rose78, Gray79, Lind79, Ston79], *timestamps* [Reed78, Thom79, Bern80, Bern81], or commit-time *validation* (or certification) [Bada79, Casa79, Baye80, Kung81, Ceri82]. It is also possible to form a large number of algorithms by combining these mechanisms [Bern80, Bern81]. Thus, there are a large number algorithms to choose from. Unfortunately, little is known that would assist an implementor in making this choice.

Several recent studies have addressed the problem of evaluating the performance of alternative concurrency control algorithms. These include qualitative, analytical, and simulation studies. Bernstein and Goodman performed a comprehensive qualitative study which discussed performance issues for a

This work was supported by the Air Force Office of Scientific Research Grant AFOSR-78-3596 and the Naval Electronic Systems Command Contract NESC-N00039-81-C-0569.

number of distributed locking and timestamp algorithms [Bern80]. Results of analytical studies of locking performance have been reported by Irani and Lin [Iran79] and Potier and Leblanc [Poti80]. Simulation studies of locking done by Ries and Stonebraker provide insight into granularity versus concurrency tradeoffs [Ries77, Ries79a, Ries79b]. Analytical and simulation studies by Garcia-Molina [Garc79] provide some insight into the relative performance of several variants of locking as well as a voting algorithm [Thom79] and a ring algorithm [Elli77]. Simulation studies by Lin and Nolte [Lin82] provide some comparative performance results for locking and several timestamp algorithms. A recent thesis by Galler [Gall82] provides a combination of a new analytical technique for locking, some qualitative techniques for comparing algorithms, and some simulation results for locking versus timestamps which contradict those of Lin and Nolte.

While certainly interesting, these performance studies fail to offer definitive results regarding the selection of a concurrency control algorithm. The analytical and simulation studies examine transaction throughput and response time characteristics under various workloads and system parameter settings, assuming a fixed cost (sometimes zero) for processing each concurrency control request. Little or no consideration has been given to the relative storage and CPU overheads required by the various algorithms. The studies involve lengthy analyses, large simulation programs, or both, with the underlying system models and assumptions varying from study to study. As a result, no single comprehensive analytical or simulation study of the many proposed algorithms has been undertaken, and cross-comparisons of different studies are difficult or impossible. Only Bernstein and Goodman [Bern80] and Galler [Gall82] have attempted comprehensive comparative studies, and their work thus far has been too qualitative to be conclusive.

In this paper we report on a current effort to provide a uniform model of concurrency control algorithms. The model is designed to facilitate a comprehensive comparative study, providing a uniform framework for describing and evaluating alternative concurrency control algorithms [Care83]. Here we describe our model and techniques for analyzing the relative storage and CPU overheads of various concurrency control algorithms. Section 2 presents our model, and section 3 explains how algorithms are described under the model, presenting descriptions of single-site versions of two-phase locking, basic timestamp ordering, and serial validation. Section 4 shows how the model may

be used to analyze relative storage and CPU overheads for algorithms, giving results for the algorithms described in section 3. In section 5, we describe extensions to the model and analysis techniques for multiple version and distributed concurrency control algorithms. Section 6 presents our conclusions thus far and describes our intended future work.

2. The Basic Model

The concurrency control subsystem of most database management systems can be thought of as a special-purpose scheduler [Casa79, Papa79, Bern80, Bern82a]. It accepts begin, data access, and commit requests from transactions, and decides whether to allow, postpone, or reject these requests. Concurrency control schemes of this sort are called *dynamic* and *syntactic* schemes, as they make decisions based on information as it becomes available, and the information used does not involve knowledge about the semantics of the transactions or the semantics or structure of the database. We restrict our attention to this class of concurrency control algorithms.

Our model of single-site concurrency control algorithms contains a single *concurrency control scheduler*. This scheduler keeps information about the history of requests received to date. We refer to this information as the *concurrency control database*, and we will treat it conceptually as a simple, relational database, ignoring the multitude of data structures which might be used in its implementation. For a particular concurrency control algorithm, the scheduler obeys a well-defined set of rules which tell it how to respond to incoming requests, based both on the requests themselves and on the contents of the concurrency control database. For reasons of simplicity, conciseness, and implementation independence, we formulate these rules as relational database queries. Our model is summarized in Figure 1.

2.1. Transaction Requests

Our model allows three types of requests from transactions: *BEGIN*, *END*, and *ACCESS*. The first two mark the beginning and the end of transaction execution, and the latter indicates that the requesting transaction wishes to access one or more objects. A given transaction may make a number of *ACCESS* requests in the course of its execution. When the scheduler receives a request, it also receives a collection of (*obj-id*, *mode*) pairs indicating the objects and access modes (read or write), if any, associated with the current request. We

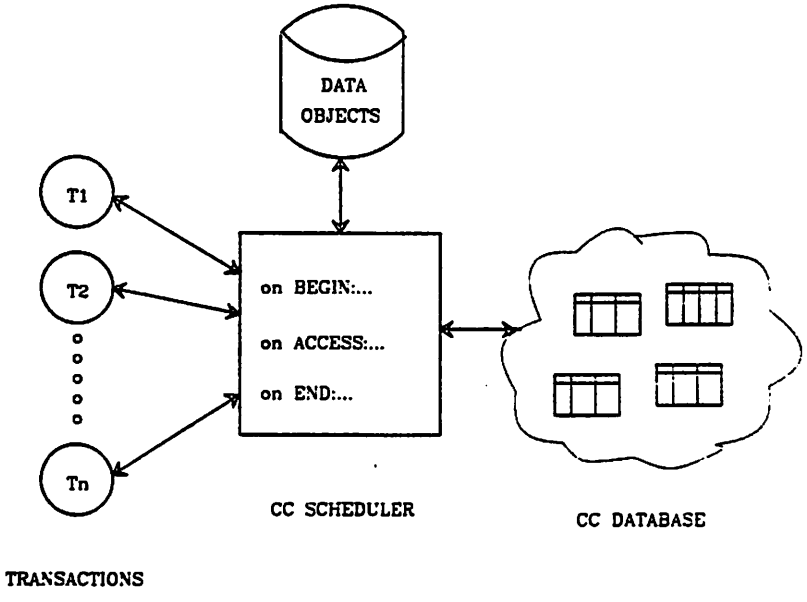

Figure 1: Concurrency control model.

refer to this collection as a relation, the *REQ* relation, for the purpose of formulating concurrency control algorithms as queries. It is assumed in our model that transactions abide by the responses received from the scheduler, accessing data objects accordingly. It is also assumed that writes are written to a deferred update list [Gray79], and that they are installed as new data values at transaction commit time, so that concurrency control algorithm descriptions need not be concerned with such details.

2.2. The Concurrency Control Database

The concurrency control database, shown in Figure 2, consists of four relations. The *XACT* relation contains transaction state information, specifying the transaction identifier, state (ready, blocked, committed, aborted), and timestamp of each current or recent transaction. The *ACC* relation contains information about accesses to objects, specifying the object identifier, access mode (read or write), transaction identifier, and timestamp for each current or recent access. This relation plays the role of a concurrency control table, such as a lock table or a timestamp table, in algorithm descriptions. The *BLKD* relation contains information about any blocked transactions, containing the transaction

identifiers of these transactions and of the transactions which they are waiting for. It is assumed that the deletion of an entry from the *BLKD* relation unblocks the corresponding transaction implicitly, allowing it to begin where it previously left off. The *HIST* relation stores histories of *ACCESS* requests which are conditionally granted, where the concurrency control decision is to be deferred until transaction commit time (such as in optimistic concurrency control algorithms). Entries in this relation specify the transaction identifiers, object identifiers, and access modes associated with such requests.

Not all concurrency control algorithms use all of the relations in the concurrency control database, as this set of relations is intended to represent the collection of all possible information which algorithms might choose to make use of. For the same reason, not all concurrency control algorithms use all of the fields of these relations. Thus, the portion of the concurrency control database used by an algorithm is specified as part of its description.

XACT(xact-id,state,ts)
ACC(obj-id,mode,xact-id,ts)
BLKD(blocked-id, cause-id)
HIST(xact-id,obj-id,mode)

Figure 2: Concurrency control database.

2.3. Algorithm Descriptions

Concurrency control algorithms are described in three parts under our model. These are:

(1) A list of the concurrency control database relations and fields used by the algorithm.

(2) A pair of views, *BLKCFL* and *RSTCFL*, which define the situations where blocking or restarting are called for, respectively.

(3) Three query sets, describing the actions to be taken on receipt of *BEGIN*, *ACCESS*, and *END* requests. These query sets access the concurrency control database and *REQ* relation associated with the current request and are presumed to execute atomically when invoked. We borrow from the QUEL query language [Ston76] for our query syntax, deviating or adding high-level macro-operations where QUEL fails to fulfill our needs.

3. Using The Model

In this section, we demonstrate the descriptive use of our single-site model by showing how two-phase locking [Gray79], basic timestamp ordering [Bern81], and serial validation [Kung81] may be described under the model. In doing so, we take several liberties with the QUEL syntax. First, we omit range statements. Second, we define the macro-operations shown in Figures 3 through 5. The *BLOCK* operation blocks a specified transaction, recording its transaction identifier and the identifier of the transaction which it is waiting for in the *BLKD* relation. The *EXPUNGE* operation deletes all of the information associated with a specified transaction, and is used at transaction commit or restart time. The *RESTART* operation restarts a specified transaction. We assume the existence of a fourth macro-operation, *CYCLE*(*xact-id*), which searches for cycles of blocked transactions in the *BLKD* relation involving a specified transaction and returns true if and only if a cycle is found. (This last operation cannot be specified in QUEL in a convenient manner.) Finally, we assume the existence of several convenient global variables, such as *req-xact-id*, the transaction identifier for the current requestor. Other such variables will be assumed and commented upon as they seem reasonable and convenient.

3.1. Two-Phase Locking

In *two-phase locking* (2PL) [Gray79], the concurrency control scheduler maintains a lock table. Transactions set read and write locks on objects before accessing them, and they release their locks at commit time. A transaction may set a read lock on an object as long as no other transaction has a write lock set on the object, and a transaction may set a write lock an object if no other transaction has a read or write lock set on the object. When a transaction tries to set a lock and fails, it must wait until the lock is released and then try again. Deadlocks are a possibility, and must either be prevented or detected and broken by restarting one of the transactions involved.

We will use the the linear-time deadlock detection algorithm of Agrawal, Carey, and DeWitt [Agra82] for this example. In this algorithm, when a transaction T_i is forced to wait for a lock on some object X, it blocks on exactly one of the transactions T_j which hold locks on X. If there are more than one, it picks one arbitrarily. As shown in [Agra82], if deadlocks are checked each time a transaction must wait, the *CYCLE*(*xact-id*) operation (ie., the deadlock detector) can operate in a very efficient manner. Figures 6 through 8 give a

```
BLOCK(xact-id1,xact-id2) =
{
  replace XACT(state = "blocked")
    where XACT.xact-id = xact-id1
  append to BLKD(xact-id1,xact-id2)
}
```

Figure 3: Definition of *BLOCK* macro-operation.

```
EXPUNGE(xact-id) =
{
  delete XACT
    where XACT.xact-id = xact-id
  delete ACC
    where ACC.xact-id = xact-id
  delete BLKD
    where BLKD.blocked-id = xact-id
    or BLKD.cause-id = xact-id
  delete HIST
    where HIST.xact-id = req-xact-id
}
```

Figure 4: Definition of *EXPUNGE* macro-operation.

```
RESTART(xact-id) =
{
  replace XACT(state = "aborted")
    where XACT.xact-id = xact-id
  EXPUNGE(xact-id)
}
```

Figure 5: Definition of *RESTART* macro-operation.

description of 2PL using our model.

The subset of the concurrency control database needed for 2PL is specified in Figure 6. In Figure 7, the conditions under which blocking and restarts are required are defined as views. The *BLKCFL* view says that a block conflict has occurred if there is an *ACC* relation entry for one of the current requests, and either the current request is a read request and the *ACC* entry is a write entry, or else the current request is a write request (in which case the mode of the *ACC* entry does not matter). In other words, the *ACC* relation serves as a lock table, and a transaction must block if an incompatible lock is already set on an object that it wants to access. The *RSTCFL* view says that a restart conflict has occurred if there is a cycle in the *BLKD* relation involving the current requesting transaction. In other words, a transaction must restart if it is the cause of a deadlock.

Figure 8 gives the query sets for processing requests under 2PL. When a *BEGIN* request arrives, the state of the requesting transaction is set to indicate

```
XACT(xact-id,state)
ACC(xact-id,mode,obj-id)
BLKD(blocked-id,cause-id)
```

Figure 6: Concurrency control database for 2PL.

```
define view BLKCFL(xact-id = ACC.xact-id)
  where REQ.obj-id = ACC.obj-id
  and ACC.xact-id != req-xact-id
  and ((REQ.mode = "read" and ACC.mode = "write")
    or (REQ.mode = "write"))

define view RSTCFL(xact-id = BLKD.xact-id)
  where CYCLE(BLKD.blocked-id)
  and BLK.blocked-id = req-xact-id
```

Figure 7: Block and restart conflict views for 2PL.

```
on BEGIN:
  append to XACT(req-xact-id,"ready")

on ACCESS:
  replace ACC(mode = REQ.mode)
    where not any(BLKCFL)
    and ACC.obj-id = REQ.obj-id
    and ACC.xact-id = req-xact-id
  append to ACC(req-xact-id,REQ.mode,REQ.obj-id)
    where not any(BLKCFL)
    and not any(ACC.obj-id
      where ACC.obj-id = REQ.obj-id
      and ACC.xact-id = req-xact-id)
  BLOCK(req-xact-id,BLKCFL.xact-id)
    where any(BLKCFL)
    and BLKCFL.xact-id = min(BLKCFL.xact-id)
  RESTART(req-xact-id)
    where any(BLKCFL) and any(RSTCFL)

on END:
  replace XACT(state = "committed")
    where XACT.xact-id = req-xact-id
  EXPUNGE(req-xact-id)
```

Figure 8: Request processing queries for 2PL.

that it is ready to run. When an *ACCESS* request arrives, the *BLKCFL* view is materialized. If no block conflicts exist, then the *ACC* relation is updated to indicate that locks have been granted on all requested objects. If a block conflict does exist, the requesting transaction is blocked on one of the conflicting transactions (the one with the smallest transaction identifier is arbitrarily picked here), and the *RSTCFL* view is materialized. If a restart conflict exists, the requesting transaction is restarted. This corresponds to granting

requests if no locks interfere, blocking a transaction if one or more locks are unobtainable, and restarting a transaction if it becomes the cause of a deadlock condition.

3.2. Basic Timestamp Ordering

In *basic timestamp ordering* (BTO) [Bern81], the concurrency control scheduler assigns timestamps to transactions according to their startup order. It maintains a table of read and write timestamps for objects, recording the timestamps of the latest reader and writer for each object. (Entries with timestamps older than the oldest active transaction need not be kept in the table.) A read request for an object is granted as long as no newer write timestamp exists for the object, and a write request is granted as long as no newer read or write timestamp exists for the object. If a request is rejected, the requesting transaction is restarted. Deadlock is impossible, although cyclic restarts are a possibility [Date82].

For the purpose of this example, read requests will be processed as they arrive, and all write requests will be processed together just prior to transaction commit time. This simplifies the considerations involved in making BTO work with two-phase commit, as otherwise some scheduling would be required to prevent transactions from reading objects for which a write request has been processed but the associated deferred update has not yet taken place. Figures 9 through 11 give a description of BTO using our model. The global variable *req−ts* is assumed to contain the timestamp of the current requestor. The macro-operation *CURRENT−TS*() is assumed to return the current timestamp value, implicitly increasing its value for the next time around and setting the global variable *current−ts* to the value of the current timestamp. The global variable *oldest−ts* is assumed to contain the timestamp of the oldest active transaction. The global variable *req−type* is assumed to indicate the type of the current request.

While this description appears a bit lengthy, its semantics are actually relatively simple. The *ACC* relation plays the role of the timestamp table for *BTO*. The "**append to** *ACC*..." portion of the *ACCESS* request query set in Figure 11 handles the case where there is no current timestamp for a requested object, recording a new one, and the "**replace** *ACC*..." portion of the *ACCESS* request query set handles the case where there is a current timestamp for the object, updating it as called for by the BTO algorithm. The *HIST* relation is used to

```
XACT(xact-id,state,ts)
ACC(ts,mode,obj-id)
HIST(xact-id,obj-id)
```

Figure 9: Concurrency control database for BTO.

```
define view RSTCFL(obj-id = ACC.obj-id)
  where (REQ.obj-id = ACC.obj-id
    and ACC.ts > req-ts
    and (REQ.mode = "read" and ACC.mode = "write)
  or (HIST.obj-id = ACC.obj-id
    and HIST.xact-id = req-xact-id
    and ACC.ts > req-ts
    and req-type = END)
```

Figure 10: Restart conflict view for BTO.

defer write timestamp checking until commit time, with similar timestamp checking and updating involving the *HIST* relation occurring in the *END* request portion of the description.

3.3. Serial Validation

In *serial validation* (SV) [Kung81], the concurrency control scheduler keeps track of the writesets of recently committed transactions. Transactions run freely until commit-time, at which point each transaction is submitted to a validity test to see if committing it will leave the database in a consistent state. For a committing transaction T_i, the test considers all recently committed transactions T_{rc}, where a recently committed transaction is one that committed since T_i started running. The test results in T_i being committed iff $readset(T_i) \cap writeset(T_{rc}) = \phi$ for all T_{rc}, and being restarted otherwise.

Rather than write a description of serial validation as it was presented in [Kung81], we will describe a more efficient version with different but provably equivalent semantics. In our version, transactions will be assigned a startup timestamp and a commit timestamp (though only their startup timestamps will be stored). Write timestamps will be maintained for all data objects, and the write timestamp for an object X will be the commit timestamp of its most recent (successfully committed) writer. A transaction will be allowed to commit if and only if the write timestamp of each object X in its readset is smaller than its startup timestamp. It is fairly easy to show that this test is equivalent to the original readset/writeset test of [Kung81], and it is clearly more efficient. A formal equivalence proof is presented in [Care83]. Figures 12 through 14 give a

```
on BEGIN:
  append to XACT(req-xact-id,"ready",CURRENT-TS())

on ACCESS:
  replace ACC(ts = max(ACC.ts,req-ts)
    where not any(RSTCFL)
    and REQ.mode = "read"
    and ACC.mode = "read"
    and ACC.obj-id = REQ.obj-id
  append to ACC(req-ts,REQ.mode,REQ.obj-id)
    where not any(RSTCFL)
    and REQ.mode = "read"
    and not any(ACC.obj-id
      where ACC.obj-id = REQ.obj-id
        and ACC.mode = "read")
  append to HIST(req-xact-d,REQ.obj-id)
    where REQ.mode = "write"
  RESTART(XACT.xact-id)
    where XACT.xact-id = REQ.xact-id
    and any(RSTCFL)
    and REQ.mode = "read"

on END:
  replace XACT(state = "committed")
    where XACT.xact-id = req-xact-id
    and not any(RSTCFL)
  replace ACC(ts = max(ACC.ts,req-ts)
    where not any(RSTCFL)
    and ACC.mode = "write"
    and ACC.obj-id = HIST.obj-id
    and HIST.xact-id = req-xact-id
  append to ACC(req-ts,HIST.obj-id,"write")
    where not any(RSTCFL)
    and HIST.xact-id = req-xact-id
    and not any(ACC.obj-id
      where ACC.obj-id = HIST.obj-id
        and ACC.mode = "write")
  RESTART(XACT.xact-id)
    where XACT.xact-id = req-xact-id
    and any(RSTCFL)
  delete HIST
    where HIST.xact-id = req-xact-id
  delete XACT
    where XACT.xact-id = req-xact-id
  delete ACC
    where ACC.ts < oldest-ts
```

Figure 11: Request processing queries for BTO.

description, somewhat simpler than the previous descriptions, of SV using our model.

```
XACT(xact-id,state,ts)
ACC(ts,obj-id)
HIST(xact-id,mode,obj-id)
```

Figure 12: Concurrency control database for SV.

```
define view RSTCFL(obj-id = HIST.obj-id)
  where HIST.obj-id = ACC.obj-id
  where HIST.xact-id = req-xact-id
  where HIST.mode = "read"
  and ACC.ts > req-ts
```

Figure 13: Restart conflict view for SV.

```
on BEGIN:
  append to XACT(req-xact-id,"ready",CURRENT-TS())

on ACCESS:
  append to HIST(req-xact-id,REQ.mode,REQ.obj-id)

on END:
  replace XACT(state = "committed")
    where XACT.xact-id = req-xact-id
    and not any(RSTCFL)
  RESTART(XACT.xact-id)
    where XACT.xact-id = req-xact-id
    and any(RSTCFL)
  replace ACC(ts = current-ts)
    where not any(RSTCFL)
    and HIST.mode = "write"
    and ACC.obj-id = HIST.obj-id
    and HIST.xact-id = req-xact-id
  append to ACC(obj-id = HIST.obj-id,ts = current-ts)
    where not any(RSTCFL)
    and HIST.mode = "write"
  and HIST.xact-id = req-xact-id
    and not any(ACC where ACC.obj-id = HIST.obj-id)
  delete HIST
    where HIST.xact-id = req-xact-id
  delete XACT
    where XACT.xact-id = req-xact-id
  delete ACC
    where ACC.ts < oldest-ts
```

Figure 14: Request processing queries for SV.

4. Algorithm Overhead Comparisons

In this section, we present techniques for comparing the relative overhead characteristics of various concurrency control algorithms. The storage and CPU overheads are compared via a simple complexity analysis, based on implementation-independent units of CPU and storage cost and influenced to some extent by ideas presented in [Bern80]. We illustrate the use of our

techniques by using them to analyze and compare the three algorithms described in the previous section.

To facilitate these cost analyses, we will use a performance model based on a set of simple parameters. Let R be the average readset size for transactions, and let F_w be the average fraction of the readset also included in the writeset. Each transaction thus makes an average of $R(1+F_w)$ data access requests. (We assume the writeset to be a subset of the readset for each transaction, and we assume that transactions do not make the same request twice.) Let T_a be the average number of transactions in the system. Let F_b be the average fraction of blocked transactions, and let F_{rc} be the factor which, when multiplied by T_a, yields the average number of recently committed transactions. (A recently committed transaction is one which committed since the startup time of the oldest transaction still running).

The blocking and restart characteristics of algorithms will influence the parameters F_b and F_{rc}, so they will vary from algorithm to algorithm. The parameter F_w is determined solely by the transaction mix. To bound these parameters, note that $0 \leq F_b \leq 1$ and $0 \leq F_w \leq 1$. For the parameter F_{rc}, however, all that is certain is that $F_{rc} \geq 0$, as F_{rc} is determined by the variance in running times for transactions in the transaction mix. For example, a very long transaction mixed with a collection of short transactions would result in a large value for F_{rc}.

4.1. Storage Overhead

We analyze the sizes of the relations in the concurrency control database for various algorithms in order to compare their storage overheads. We take one field of one tuple of one relation as the unit of storage cost for this analysis. Given an algorithm, the tuple widths of the relations in the concurrency control database are explicit in the description, and the cardinalities of the relations are determined by the nature of the query sets in the description. The overall database size is simply the sum of the width-cardinality products for the relations in the database. Both upper and lower bounds on the storage overhead of algorithms are quite easily determined in our model.

We consider the 2PL algorithm first. The *XACT* relation represents a storage cost of $2T_a$, and the *BLKD* relation represents a cost of $2F_b T_a$. For the *ACC* relation, a storage cost of $3T_a(1-F_w)R$ is incurred for storing read locks (note that only one lock is set on objects that are to be written). For storing

write locks, the cost can vary from as low as $3F_wR$, in the case where all T_a transactions write the same objects, to as high as $3T_aF_wR$, in the case where no two transactions write the same object. Thus, we have:

$$STO_{2PL} \leq 2T_a(1+F_b)+3T_aR \quad (1a)$$

$$STO_{2PL} \geq 2T_a(1+F_b)+3T_aR(1-F_w)+3F_wR \quad (1b)$$

Similar reasoning yields the following results for BTO and SV:

$$STO_{BTO} \leq 3T_a(1+F_{rc})R(1+F_w)+T_a(3+2F_wR) \quad (2a)$$

$$STO_{BTO} \geq 3R(1+F_w)+T_a(3+2F_wR) \quad (2b)$$

$$STO_{SV} \leq 2T_a(1+F_{rc})RF_w+3T_a(1+R(1+F_w)) \quad (3a)$$

$$STO_{SV} \geq 2RF_w+3T_a(1+R(1+F_w)) \quad (3b)$$

Given the bounds on F_b and F_w, we can draw some conclusions about the relative storage overheads of the algorithms. From equations (1a), (2a), and (3a), one can conclude that 2PL has the smallest worst-case storage overhead of the three algorithms, which is $(4+3R)T_a$. The worst-case storage overheads of the other two algorithms are dependent on the parameter F_{rc}, which is unbounded. A more detailed analysis of these equations reveals that the worst-case storage overhead of SV is strictly smaller than that of BTO (assuming comparable F_{rc} values for the two algorithms), and that, if $F_b \leq 1/2$, 2PL is certain to have a smaller worst-case storage overhead than both SV and BTO. The worst-case storage overhead occurs when transactions do not compete for the same data items, which is likely to be the case for real mixes of transactions [Gray81]. Thus, F_b is likely to be small for 2PL, leading to the conclusion that 2PL dominates SV, and SV in turn dominates BTO, with respect to worst-case storage overhead.

A comparison of equations (2b) and (3b) reveals that, with respect to best-case storage overhead, BTO dominates SV for $T_a \geq 3$. Comparing equations (1b) and (3b), we find that, if $F_b \leq 1/2$, 2PL is certain to dominate SV as well. Finally, a comparison of equations (1b) and (2b) indicates that BTO dominates 2PL unless $F_w \geq 3/5$ and $F_b \leq 1/2$. Since the best-case overheads apply when transactions tend to conflict (access the same objects), this combination of F_w and F_b is impossible; if F_w is large, transactions will be competing for write locks on these shared objects, and lots of blocking will occur. Hence, BTO dominates 2PL

with respect to best-case storage overhead. To summarize the overall storage overhead results, then, SV is the worst of the three algorithms. 2PL is best in terms of worst-case storage overhead, indicating that it is superior under low-conflict transaction mixes. BTO is best in terms of best-case storage overhead, meaning that it is best under high-conflict transaction mixes.

4.2. CPU Overhead

We analyze the number of operations involved in executing the query sets for various algorithms in order to compare their CPU overheads. We take one tuple access, insertion, or replacement in one relation as the unit of CPU cost for this analysis, assuming that the CPU time required is proportional to the number of table lookups, as proposed (in different terms) by Bernstein and Goodman [Bern80]. We do not assess CPU cost for accesses to the REQ relation, as this is simply our model of the way transactions pass requests to the scheduler.

Unfortunately, analyzing the CPU overhead of a given concurrency control algorithm is, in the general case, considerably more complex than analyzing the storage overhead of the algorithm. In this paper we consider only the no-conflict CPU overhead [Bada81], the CPU overhead experienced by a transaction which does not conflict in any way with other concurrent transactions. Since actual conflicts are reported to be rare [Gray81], the no-cost CPU overhead should be a reasonable "first-order" metric. We leave for future work the problem of generalizing the analysis to include the additional sources of CPU overhead associated with transactions which must restart or repeat requests due to blocking.

We again consider 2PL first. The cost of processing a *BEGIN* request is 1. The cost of materializing the *BLKCFL* view is 1, so the cost of processing $R(1+F_w)$ data access (*ACCESS*) requests is $2R(1+F_w)$ if no blocking occurs. The cost of processing an *END* request is $3+R$ (assuming one *BLKD* access to determine the lack of blocked transactions). Hence, we have:

$$CPU_{2PL} = 4+R(3+2F_w) \qquad (4)$$

Similar analyses can be performed for BTO and SV. The cost of processing an *END* request for BTO and SV depends on the number of timestamps deleted at that time; in the no-conflict case, we assume that all transactions access different data items, meaning that all timestamps associated with a given

transaction must eventually be explicitly deleted. We charge this timestamp deletion overhead to the transaction creating the timestamp, even though deletion may occur at some later point in time. Other details of the CPU analysis for BTO and SV are quite similar to locking, so we do not present them here. We find that:

$$CPU_{BTO} = 3+R(3+7F_w) \quad (5)$$

$$CPU_{SV} = 3+R(4+5F_w) \quad (6)$$

Comparing equation (4) with equation (5), we find that 2PL has a smaller no-conflict CPU overhead than BTO unless F_w is extremely small, in which case 2PL and BTO are comparable. Comparing equation (4) with equation (6), we find that 2PL also has a smaller no-conflict CPU overhead than SV. Comparing equations (5) and (6), we find that BTO has a smaller no-conflict CPU overhead than SV if $F_w < 1/2$, and that SV has a smaller no-conflict CPU overhead if $F_w > 1/2$. Thus, with respect to this CPU overhead metric, 2PL is dominant, BTO is second-best if writing is infrequent, and SV is second-best if writing is frequent.

4.3. Overhead Comparison Summary

In the previous sections, we compared the storage and CPU overheads of 2PL, BTO, and SV. We found 2PL to be the algorithm involving the least storage overhead under low-conflict transaction mixes, with BTO being the best under high-conflict mixes. SV was the worst algorithm with respect to storage overhead. We found 2PL to be the algorithm with the smallest no-conflict CPU overhead. BTO turned out to be second-best with respect to no-conflict CPU overhead if writing is infrequent, with SV being second-best if writing is frequent. These results are summarized in Figure 15. We will pursue these comparisons and investigate tradeoff points in a more rigorous fashion in [Care83]. In particular, we intend to use the storage and CPU results to partition the parameter space into regions where various algorithms are clearly dominant.

5. Model Extensions

In our ongoing study of concurrency control algorithm performance, we are studying multiple version and distributed algorithms as well as single-site algorithms. In this section we briefly describe the extensions required to our model which facilitate these studies.

Results of Overhead Comparisons		
Algorithm	Storage Overhead	CPU Overhead
2PL	best under low conflicts	best no-conflict overhead
BTO	best under high conflicts	second best under infrequent writing
SV	worst of the three schemes	second best under frequent writing

Figure 15: Summary of algorithm overhead results.

5.1. Multiple Versions

Several recent concurrency control algorithm proposals involve maintaining multiple versions of data objects [Reed78, Baye80, Stea81, Chan82, Bern82b]. In order to describe such algorithms within our model, we introduce a new concurrency control database relation, the *OBJ* relation, with *obj–id*, *version–id*, and *obj–value* fields. Each version of each object in the database has a corresponding tuple in this relation. In places where an *obj–id* was called for in single-site algorithms, we use an (*obj–id*, *version–id*) pair in our multiple version model. The analysis techniques can be applied to this extended model in the same manner as for the single-site model, except that C_v units of storage cost are assessed for *obj–value* fields of *OBJ* tuples (to reflect the fact that objects require much more storage than typical concurrency control information).

5.2. Distributed Databases

Many recent concurrency control algorithm proposals are intended for use in distributed database systems [Rose78, Mena78, Ston79, Lind79, Bern80, Bern81, Bern82a, Thom79, Ceri82]. In order to describe distributed concurrency control algorithms within our model, we assume that each site has a concurrency control scheduler with an associated concurrency control database, and that the schedulers interact via messages. To model this interaction, we introduce some new notation for use in writing algorithm descriptions for

distributed systems. Queries of the form <*command*> **where** <*predicate*> *AT−SITES−OF*(*obj−id*) will be used to indicate that the predicate must be true at all sites where the specified object resides, indicating the need for a round-trip message exchange to evaluate the predicate. In cases where the *AT−SITES−OF* clause is left out, just the local site will be involved in evaluating the predicate.

With this extension, algorithm descriptions will be formulated as before, except that the *AT−SITES−OF*(X) set must be described for all objects X. It is this set description which will serve to differentiate primary site, primary copy, and decentralized concurrency control schemes [Bern81, Bern82a] from one another, for example. The overhead analysis techniques carry through, though is necessary to account for the additional overhead when the *AT−SITES−OF* set contains more than a single site. Also, a new type of overhead, message overhead, arises in distributed systems. This overhead may be characterized by analyzing the number of messages required when executing the new query sets on behalf of transactions.

6. Conclusions

We have presented a new model of concurrency control algorithms, one which provides a unified framework for describing and comparing the many algorithm proposals. We have given several sample descriptions, and we have shown how our model facilitates analyses of the relative storage and CPU overheads of algorithms. Our model differs from those of other researchers [Bern80, Bern81a, Gall82] in this respect, as other attempts at uniform concurrency control frameworks have not been able to support both algorithm descriptions and quantitative algorithm comparisons. Finally, we have indicated how we are extending our model to include the domains of multiple version and distributed concurrency control algorithms.

We intend to use this model to perform a comprehensive study of the overheads of various concurrency control algorithms, describing them and comparing their storage, CPU, and message overheads. We have also written a fairly general simulation program, allowing a concurrency control algorithm to be described in terms of a small collection of Pascal routines (called by the simulator as needed), and we will use this simulator to validate our overhead bounds and to study the concurrency properties of algorithms as well.

References

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings of the COMPSAC '79 Conference, Chicago, Illinois, November 1979.

[Bada81] Badal, D., "Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Emeryville, CA, February 1981.

[Baye80] Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems 5(2), June 1980.

[Bern80] Bernstein, P., and Goodman, N., "Fundamental Algorithms for Concurrency Control in Distributed Database Systems", Technical Report, Computer Corporation of America, 1980.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", ACM Computing Surveys 13(2), June 1981.

[Bern82a] Bernstein, P., and Goodman, N., "A Sophisticate's Introduction to Distributed Database Concurrency Control", Proceedings of the Eighth International Conference on Very Large Data Bases, September 1982.

[Bern82b] Bernstein, P., and Goodman, N., "Multiversion Concurrency Control Theory and Algorithms", Technical Report No. TR-20-82, Aiken Computation Laboratory, Harvard University, June 1982.

[Care83] Carey, M., "Modeling and Evaluation of Database Concurrency Control Algorithms", Ph.D. Thesis, Computer Science Division, EECS Department, University of California, Berkeley, (in preparation).

[Casa79] Casanova, M., "The Concurrency Control Problem for Database Systems", Ph.D. Thesis, Computer Science Department, Harvard University, 1979.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February, 1982.

[Chan82] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., "The Implementation of An Integrated Concurrency Control and Recovery Scheme", Proceedings of the ACM-SIGMOD International Conference on Management of Data, March 1982.

[Date82] Date, C., "An Introduction to Database Systems (Volume II)", Addison-Wesley Publishing Company, 1982.

[Elli77] Ellis, C., "A Robust Algorithm for Updating Duplicate Databases", Proceedings of the 2nd Berkeley Workshop on Distributed Databases and Computer Networks, May 1977.

[Gall82] Galler, B., "Concurrency Control Performance Issues" Ph.D. Thesis, Computer Science Department, University of Toronto, September, 1982.

[Garc79] Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database", Ph.D. Thesis, Computer Science Department, Stanford University, June 1979.

[Gray79] Gray, J., "Notes On Database Operating Systems", in "Operating Systems: An Advanced Course", Springer-Verlag, 1979.

[Gray81] Gray, J., Homan, P., Korth, H., and Obermarck, R., "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System", Report No. RJ3066, IBM San Jose Research Laboratory, February 1981.

[Iran79] Irani, K., and Lin, H., "Queueing Network Models for Concurrent Transaction Processing in a Database System, Proceedings of the ACM-SIGMOD International Symposium on Management of Data, 1979.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6(2), June 1981.

[Lin82] Lin, W., and Nolte, J., "Distributed Database Control and Allocation: Semi-Annual Report", Technical Report, Computer Corporation of America, Cambridge, Massachusetts, January 1982.

[Lind79] Lindsay, B., Selinger, P., Galtieri, C., Gray, J., Lorie, R., Price, T., Putzolu, F., Traiger, I., and Wade, B., "Notes on Distributed Databases", Research Report, IBM San Jose Research Center, 1979.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

[Papa79] Papadimitriou, C., "Serializability of Concurrent Updates", Journal of the ACM 26(4), October 1979.

[Poti80] Potier, D., and LeBlanc, P., "Analysis of Locking Policies in Database Management Systems", Proceedings of the Performance '80 Conference, 7th IFIP W.G.7.3 International Symposium on Computer Performance Modeling, Measurement, and Evaluation, Toronto, May 1980.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System", PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Ries77] Ries, D., and Stonebraker, M., "Effects of Locking Granularity on Database Management System Performance", ACM Transactions on Database Systems 2(3), September 1977.

[Ries79a] Ries, D., "The Effects of Concurrency Control on Database Management System Performance", PhD Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1979.

[Ries79b] Ries, D., and Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems 4(2), June 1979.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems", ACM Transactions on Database Systems 3(2), June 1978.

[Stea81] Stearns, R., and Rosenkrantz, D., "Distributed Database Concurrency Controls Using Before-Values", Technical Report, SUNY Albany, February 1981.

[Ston76] Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES", ACM Transactions on Database Systems 1(3), September 1976.

[Ston79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", IEEE Transactions on Software Engineering 5(3), May 1979.

[Thom79] Thomas, R., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Transactions on Database Systems 4(2), June 1979.