On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

(1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in primary memory, so that it is necessary to swap parts of the database from secondary memory as needed.
(2) Even if the entire database can be present in primary memory, there may be multiple processors.

In both cases the hardware will be underutilized if the degree of concurrency is too low.

However, as is well known, unrestricted concurrent access to a shared database will, in general, cause the integrity of the database to be lost. Most current

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.
Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

approaches to this problem involve some type of locking. That is, a mechanism is provided whereby one process can deny certain other processes access to some portion of the database. In particular, a lock may be associated with each node of the directed graph, and any given process is required to follow some locking protocol so as to guarantee that no other process can ever discover any lack of integrity in the database temporarily caused by the given process.

The locking approach has the following inherent disadvantages.

(1) Lock maintenance represents an overhead that is not present in the sequential case. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must, in general, use locking in order to guarantee that the data being read are not modified by other transactions at the same time. Also, if the locking protocol is not deadlock-free, deadlock detection must be considered to be part of lock maintenance overhead.

(2) There are no general-purpose deadlock-free locking protocols for databases that always provide high concurrency. Because of this, some research has been directed at developing special-purpose locking protocols for various special cases. For example, in the case of B-trees [1], at least nine locking protocols have been proposed [2, 3, 9, 10, 13].

(3) In the case that large parts of the database are on secondary memory, concurrency is significantly lowered whenever it is necessary to leave some congested node locked (a congested node is one that is often accessed, e.g., the root of a tree) while waiting for a secondary memory access.

(4) To allow a transaction to abort itself when mistakes occur, locks cannot be released until the end of the transaction. This may again significantly lower concurrency.

(5) Most important for the purposes of this paper, *locking may be necessary only in the worst case.* Consider the following simple example: The directed graph consists solely of roots, and each transaction involves one root only, any root equally likely. Then if there are n roots and two processes executing transactions at the same rate, locking is *really* needed (if at all) every n transactions, on the average.

In general, one may expect the argument of (5) to hold whenever (a) the number of nodes in the graph is very large compared to the total number of nodes involved in all the running transactions at a given time, and (b) the probability of modifying a congested node is small. In many applications, (a) and (b) are designed to hold (see Section 6 for the B-tree application).

Research directed at finding deadlock-free locking protocols may be seen as an attempt to lower the expense of concurrency control by eliminating transaction backup as a control mechanism. In this paper we consider the converse problem, that of eliminating locking. We propose two families of concurrency controls that do not use locking. These methods are "optimistic" in the sense that they rely for efficiency on the hope that conflicts between transactions will not occur. If (5) does hold, such conflict will be rare. This approach also has the advantage that it is completely general, applying equally well to any shared directed graph structure and associated access algorithms. Since locks are not used, it is deadlock-free (however, starvation is a possible problem, a solution for which we discuss).

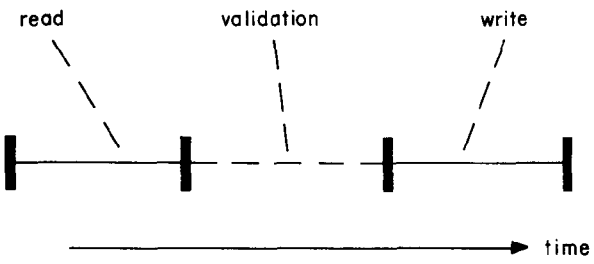

Fig. 1. The three phases of a transaction.

It is also possible using this approach to avoid problems (3) and (4) above. Finally, if the transaction pattern becomes query dominant (i.e., most transactions are read-only), then the concurrency control overhead becomes almost totally negligible (a partial solution to problem (1)).

The idea behind this optimistic approach is quite simple, and may be summarized as follows.

(1) Since reading a value or a pointer from a node can never cause a loss of integrity, reads are completely unrestricted (however, returning a result from a query is considered to be equivalent to a write, and so is subject to validation as discussed below).
(2) Writes are severely restricted. It is required that any transaction consist of two or three phases: a *read phase*, a *validation phase*, and a possible *write phase* (see Figure 1). During the read phase, all writes take place on local copies of the nodes to be modified. Then, if it can be established during the validation phase that the changes the transaction made will not cause a loss of integrity, the local copies are made global in the write phase. In the case of a query, it must be determined that the result the query would return is actually correct. The step in which it is determined that the transaction will not cause a loss of integrity (or that it will return the correct result) is called *validation*.

If, in a locking approach, locking is only necessary in the worst case, then in an optimistic approach validation will fail also only in the worst case. If validation does fail, the transaction will be backed up and start over again as a new transaction. Thus a transaction will have a write phase only if the preceding validation succeeds.

In Section 2 we discuss in more detail the read and write phases of transactions. In Section 3 a particularly strong form of validation is presented. The correctness criteria used for validation are based on the notion of serial equivalence [4, 12, 14]. In the next two sections concurrency controls that rely on the serial equivalence criteria developed in Section 3 for validation are presented. The family of concurrency controls in Section 4 have serial final validation steps, while the concurrency controls of Section 5 have completely parallel validation, at however higher total cost. In Section 6 we analyze the application of optimistic methods to controlling concurrent insertions in B-trees. Section 7 contains a summary and a discussion of future research.

2. THE READ AND WRITE PHASES

In this section we briefly discuss how the concurrency control can support the read and write phases of user-programmed transactions (in a manner invisible to the user), and how this can be implemented efficiently. The validation phase will be treated in the following three sections.

We assume that an underlying system provides for the manipulation of objects of various types. For simplicity, assume all objects are of the same type. Objects are manipulated by the following procedures, where n is the name of an object, i is a parameter to the type manager, and v is a value of arbitrary type (v could be a pointer, i.e., an object name, or data):

create	create a new object and return its name.
delete(n)	delete object n.
read(n, i)	read item i of object n and return its value.
write (n, i, v)	write v as item i of object n.

In order to support the read and write phases of transactions we also use the following procedures:

copy(n)	create a new object that is a copy of object n and return its name.
exchange($n1$, $n2$)	exchange the names of objects $n1$ and $n2$.

The concurrency control is invisible to the user; transactions are written as if the above procedures were used directly. However, transactions are required to use the syntactically identical procedures *tcreate*, *tdelete*, *tread*, and *twrite*. For each transaction, the concurrency control maintains sets of object names accessed by the transaction. These sets are initialized to be empty by a *tbegin* call. The body of the user-written transaction is in fact the read phase mentioned in the introduction; the subsequent validation phase does not begin until after a *tend* call. The procedures *tbegin* and *tend* are shown in detail in Sections 4 and 5. The semantics of the remaining procedures are as follows:

```
tcreate = (
  n := create;
  create set := create set ∪ {n};
  return n)

twrite(n, i, v) = (
  if n ∈ create set
      then write(n, i, v)
  else if n ∈ write set
      then write(copies[n], i, v)
  else (
      m := copy(n);
      copies[n] := m;
      write set := write set ∪ {n};
      write(copies[n], i, v)))

tread(n, i) = (
  read set := read set ∪ {n};
  if n ∈ write set
      then return read(copies[n], i)
```

else
return *read*(*n*, *i*))

tdelete(*n*) = (
delete set := *delete set* ∪ {*n*}).

Above, *copies* is an associative vector of object names, indexed by object name. We see that in the read phase, no global writes take place. Instead, whenever the first write to a given object is requested, a copy is made, and all subsequent writes are directed to the copy. This copy is potentially global but is inaccessible to other transactions during the read phase by our convention that all nodes are accessed only by following pointers from a root node. If the node is a root node, the copy is inaccessible since it has the wrong name (all transactions "know" the global names of root nodes). It is assumed that no root node is created or deleted, that no dangling pointers are left to deleted nodes, and that created nodes become accessible by writing new pointers (these conditions are part of the integrity criteria for the data structure that each transaction is required to individually preserve).

When the transaction completes, it will request its validation and write phases via a *tend* call. If validation succeeds, then the transaction enters the write phase, which is simply

for $n \in$ *write set* **do** *exchange*(*n*, *copies*[*n*]).

After the write phase all written values become "global," all created nodes become accessible, and all deleted nodes become inaccessible. Of course some cleanup is necessary, which we do not consider to be part of the write phase since it does not interact with other transactions:

(**for** $n \in$ *delete set* **do** *delete*(*n*);
for $n \in$ *write set* **do** *delete*(*copies*[*n*])).

This cleanup is also necessary if a transaction is aborted.

Note that since objects are virtual (objects are referred to by name, not by physical address), the *exchange* operation, and hence the write phase, can be made quite fast: essentially, all that is necessary is to exchange the physical address parts of the two object descriptors.

Finally, we note that the concept of two-phase transactions appears to be quite valuable for recovery purposes, since at the end of the read phase, all changes that the transaction intends to make to the data structure are known.

3. THE VALIDATION PHASE

A widely used criterion for verifying the correctness of concurrent execution of transactions has been variously called serial equivalence [4], serial reproducibility [11], and linearizability [14]. This criterion may be defined as follows.

Let transactions $T_1, T_2, \ldots, T_n$ be executed concurrently. Denote an instance of the shared data structure by d, and let D be the set of all possible d, so that each T_i may be considered as a function:

$$T_i : D \rightarrow D.$$

If the initial data structure is d_{initial} and the final data structure is d_{final}, the concurrent execution of transactions is correct if some permutation π of $\{1, 2, \ldots, n\}$ exists such that

$$d_{\text{final}} = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \cdots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_{\text{initial}}), \tag{1}$$

where "$\circ$" is the usual notation for functional composition.

The idea behind this correctness criterion is that, first, each transaction is assumed to have been written so as to individually preserve the integrity of the shared data structure. That is, if d satisfies all integrity criteria, then for each T_i, $T_i(d)$ satisfies all integrity criteria. Now, if d_{initial} satisfies all integrity criteria and the concurrent execution of $T_1, T_2, \ldots, T_n$ is serially equivalent, then from (1), by repeated application of the integrity-preserving property of each transaction, d_{final} satisfies all integrity criteria. Serial equivalence is useful as a correctness criterion since it is in general much easier to verify that (a) each transaction preserves integrity and (b) every concurrent execution of transaction is serially equivalent than it is to verify directly that every concurrent execution of transactions preserves integrity. In fact, it has been shown in [7] that serialization is the weakest criterion for preserving consistency of a concurrent transaction system, even if complete syntactic information of the system is available to the concurrency control. However, if semantic information is available, then other approaches may be more attractive (see, e.g., [6, 8]).

3.1 Validation of Serial Equivalence

The use of validation of serial equivalence as a concurrency control is a direct application of eq. (1) above. However, in order to verify (1), a permutation π must be found. This is handled by *explicitly* assigning each transaction T_i a unique integer *transaction number* $t(i)$ during the course of its execution. The meaning of transaction numbers in validation is the following: there must exist a serially equivalent schedule in which transaction T_i comes before transaction T_j whenever $t(i) < t(j)$. This can be guaranteed by the following validation condition: for each transaction T_j with transaction number $t(j)$, and for all T_i with $t(i) < t(j)$, one of the following three conditions must hold (see Figure 2):

(1) T_i completes its write phase before T_j starts its read phase.
(2) The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.
(3) The write set of T_i does not intersect the read set *or* the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

Condition (1) states that T_i actually completes before T_j starts. Condition (2) states that the writes of T_i do not affect the read phase of T_j, and that T_i finishes writing before T_j starts writing, hence does not overwrite T_j (also, note that T_j cannot affect the read phase of T_i). Finally, condition (3) is similar to condition (2) but does not require that T_i finish writing before T_j starts writing; it simply requires that T_i not affect the read phase *or* the write phase of T_j (again note that T_j cannot affect the read phase of T_i, by the last part of the condition). See [12] for a set of similar conditions for serialization.

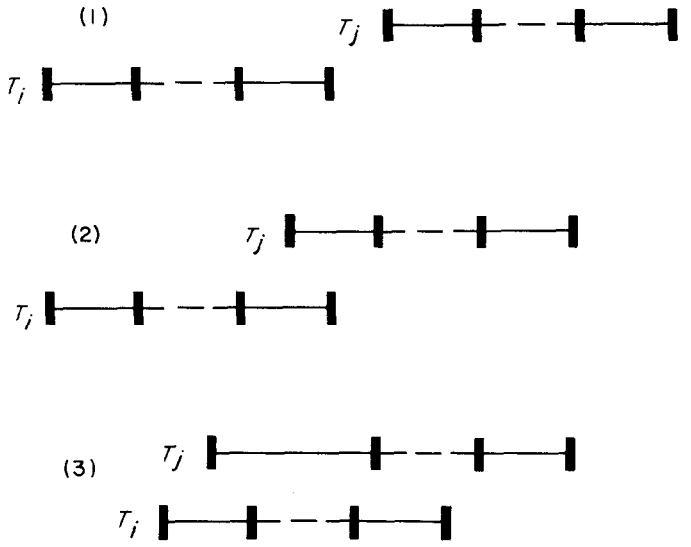

Fig. 2. Possible interleaving of two transactions.

3.2 Assigning Transaction Numbers

The first consideration that arises in the design of concurrency controls that explicitly assign transaction numbers is the question: how should transaction numbers be assigned? Clearly, they should somehow be assigned in order, since if T_i completes before T_j starts, we *must* have $t(i) < t(j)$. Here we use the simple solution of maintaining a global integer counter *tnc* (transaction number counter); when a transaction number is needed, the counter is incremented, and the resulting value returned. Also, transaction numbers must be assigned somewhere before validation, since the validation conditions above require knowledge of the transaction number of the transaction being validated. On first thought, we might assign transaction numbers at the beginning of the read phase; however, this is not optimistic (hence contrary to the philosophy of this paper) for the following reason. Consider the case of two transactions, T_1 and T_2, starting at roughly the same time, assigned transaction number n and $n + 1$, respectively. Even if T_2 completes its read phase much earlier than T_1, before being validated T_2 must wait for the completion of the read phase of T_1, since the validation of T_2 in this case relies on knowledge of the write set of T_1 (see Figure 3). In an optimistic approach, we would like for transactions to be validated immediately if at all possible (in order to improve response time). For these and similar considerations we assign transaction numbers at the end of the read phase. Note that by assigning transaction numbers in this fashion the last part of condition (3), that T_i complete its read phase before T_j completes its read phase if $t(i) < t(j)$, is automatically satisfied.

3.3 Some Practical Considerations

Given this method for assigning transaction numbers, consider the case of a transaction T that has an arbitrarily long read phase. When this transaction is

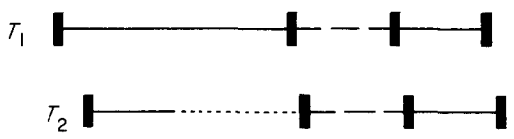

Fig. 3. Transaction 2 waits for transaction 1 in ....

validated, the write sets of all transactions that completed their read phase before T but had not yet completed their write phase at the start of T must be examined. Since the concurrency control can only maintain finitely many write sets, we have a difficulty (this difficulty does not arise if transaction numbers are assigned at the beginning of the read phase). Clearly, if such transactions are common, the assignment of transaction numbers described above is unsuitable. Of course, we take the optimistic approach and assume such transactions are very rare; still, a solution is needed. We solve this problem by only requiring the concurrency control to maintain some finite number of the most recent write sets where the number is large enough to validate almost all transactions (we say write set a is more recent than write set b if the transaction number associated with a is greater than that associated with b). In the case of transactions like T, if old write sets are unavailable, validation fails, and the transaction is backed up (probably to the beginning). For simplicity, we present the concurrency controls of the next two sections as if potentially infinite vectors of write sets were maintained; the above convention is understood to apply.

One last consideration must be mentioned at this point, namely, what should be done when validation fails? In such a case the transaction is aborted and restarted, receiving a new transaction number at the completion of the read phase. Now a new difficulty arises: what should be done in the case in which validation repeatedly fails? Under our optimistic assumptions this should happen rarely, but we still need some method for dealing with this problem when it does occur. A simple solution is the following. Later, we will see that transactions enter a short critical section during *tend*. If the concurrency control detects a "starving" transaction (this could be detected by keeping track of the number of times validation for a given transaction fails), the transaction can be restarted, but without releasing the critical section semaphore. This is equivalent to write-locking the entire database, and the "starving" transaction will run to completion.

4. SERIAL VALIDATION

In this section we present a family of concurrency controls that are an implementation of validation conditions (1) and (2) of Section 3.1. Since we are not using condition (3), the last part of condition (2) implies that write phases must be serial. The simplest way to implement this is to place the assignment of a transaction number, validation, and the subsequent write phase all in a critical section. In the following, we bracket the critical section by "⟨" and "⟩." The

concurrency control is as follows:

```
tbegin = (
   create set := empty;
   read set := empty;
   write set := empty;
   delete set := empty;
   start tn := tnc)

tend = (
   ⟨finish tn := tnc;
    valid := true;
    for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
          then valid := false;
    if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
    if valid
        then (cleanup)
        else (backup)).
```

In the above, the transaction is assigned a transaction number via the sequence *tnc* := *tnc* + 1; *tn* := *tnc*. An optimization has been made in that transaction numbers are assigned only if validation is successful. We may imagine that the transaction is "tentatively" assigned a transaction number of *tnc* + 1 with the statement *finish tn* := *tnc*, but that if validation fails, this transaction number is freed for use by another transaction. By condition (1) of Section 3.1, we need not consider transactions that have completed their write phase before the start of the read phase of the current transaction. This is implemented by reading *tnc* in *tbegin*; since a "real" assignment of a transaction number takes place only after the write phase, it is guaranteed at this point that all transactions with transaction numbers less than or equal to *start tn* have completed their write phase.

The above is perfectly suitable in the case that there is one CPU and that the write phase can usually take place in primary memory. If the write phase often cannot take place in primary memory, we probably want to have concurrent write phases, unless the write phase is still extremely short compared to the read phase (which may be the case). The concurrency controls of the next section are appropriate for this. If there are multiple CPUs, we may wish to introduce more potential parallelism in the validation step (this is only necessary for efficiency if the processors cannot be kept busy with read phases, that is, if validation is not extremely short as compared to the read phase). This can be done by using the solution of the next section, or by the following method. At the end of the read phase, we immediately read *tnc* before entering the critical section and assign this value to *mid tn*. It is then known that at this point the write sets of transactions *start tn* + 1, *start tn* + 2, . . . , *mid tn* must certainly be examined in the validation step, and this can be done outside the critical section. The concurrency control is thus

```
tend := (
    mid tn := tnc;
    valid := true;
```

```
for t from start tn + 1 to mid tn do
if (write set of transaction with transaction number t intersects read set)
    then valid := false;
⟨finish tn := tnc;
 for t from mid tn + 1 to finish tn do
    if (write set of transaction with transaction number t intersects read set)
      then valid := false;
 if valid
    then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
 if valid
    then (cleanup)
    else (backup)).
```

The above optimization can be carried out a second time: at the end of the preliminary validation step we read *tnc* a third time, and then, still outside the critical section, check the write sets of those transactions with transaction numbers from *mid tn* + 1 to this most recent value of *tnc*. Repeating this process, we derive a family of concurrency controls with varying numbers of stages of validation and degrees of parallelism, all of which however have a final indivisible validation step and write phase. The idea is to move varying parts of the work done in the critical section outside the critical section, allowing greater parallelism.

Until now we have not considered the question of read-only transactions, or queries. Since queries do not have a write phase, it is unnecessary to assign them transaction numbers. It is only necessary to read *tnc* at the end of the read phase and assign its value to *finish tn*; validation for the query then consists of examining the write sets of the transactions with transaction numbers *start tn* + 1, *start tn* + 2, . . . , *finish tn*. This need not occur in a critical section, so the above discussion on multiple validation stages does not apply to queries. This method for handling queries also applies to the concurrency controls of the next section. Note that for query-dominant systems, validation will often be trivial: It may be determined that *start tn* = *finish tn*, and validation is complete. For this type of system an optimistic approach appears ideal.

5. PARALLEL VALIDATION

In this section we present a concurrency control that uses all three of the validation conditions of Section 3.1, thus allowing greater concurrency. We retain the optimization of the previous section, only assigning transaction numbers after the write phase if validation succeeds. As in the previous solutions, *tnc* is read at the beginning and the end of the read phase; transactions with transactions numbers *start tn* + 1, *start tn* + 2, . . . , *finish tn* all may be checked under condition (2) of Section 3.1. For condition (3), we maintain a set of transaction ids *active* for transactions that have completed their read phase but have not yet completed their write phase. The concurrency control is as follows (*tbegin* is as in the previous section):

```
tend = (
  ⟨finish tn := tnc;
   finish active := (make a copy of active);
   active := active ∪ {id of this transaction}⟩;
   valid := true;
```

```
for t from start tn + 1 to finish tn do
    if (write set of transaction with transaction number t intersects read set)
      then valid := false;
for i ∈ finish active do
    if (write set of transaction T_i intersects read set or write set)
      then valid := false;
if valid
    then (
      ⟨write phase⟩;
      ⟨tnc := tnc + 1;
       tn := tnc;
       active := active—{id of this transaction}⟩;
      ⟨cleanup⟩)
    else (
      ⟨active := active—{id of transaction}⟩;
      ⟨backup⟩)).
```

In the above, at the end of the read phase *active* is the set of transactions that have been assigned "tentative" transaction numbers less than that of the transaction being validated. Note that modifications to *active* and *tnc* are placed together in critical sections so as to maintain the invariant properties of *active* and *tnc* mentioned above. Entry to the first critical section is equivalent to being assigned a "tentative" transaction number.

One problem with the above is that a transaction in the set *finish active* may invalidate the given transaction, even though the former transaction is itself invalidated. A partial solution to this is to use several stages of preliminary validation, in a way completely analogous to the multistage validation described in the previous section. At each stage, a new value of *tnc* is read, and transactions with transaction numbers up to this value are checked. The final stage then involves accessing *active* as above. The idea is to reduce the size of *active* by performing more of the validation before adding a new transaction id to *active*.

Finally, a solution is possible where transactions that have been invalidated by a transaction in *finish active* wait for that transaction to either be invalidated, and hence ignored, or validated, causing backup (this possibility was pointed out by James Saxe). However, this solution involves a more sophisticated process communication mechanism than the binary semaphore needed to implement the critical sections above.

6. ANALYSIS OF AN APPLICATION

We have previously noted that an optimistic approach appears ideal for query-dominant systems. In this section we consider another promising application, that of supporting concurrent index operations for very large tree-structured indexes. In particular, we examine the use of an optimistic method for supporting concurrent insertions in B-trees (see [1]). Similar types of analysis and similar results can be expected for other types of tree-structured indexes and index operations.

One consideration in analyzing the efficiency of an optimistic method is the expected size of read and write sets, since this relates directly to the time spent in the validation phase. For B-trees, we naturally choose the objects of the read and write sets to be the pages of the B-tree. Now even very large B-trees are only

a few levels deep. For example, let a B-tree of order m contain N keys. Then if $m = 199$ and $N \leq 2 \times 10^8 - 2$, the depth is at most $1 + \log_{100}(N + 1)/2) < 5$. Since insertions do not read or write more than one already existing node on a given level, this means that for B-trees of order 199 containing up to almost 200 million keys, the size of a read or write set of an insertion will never be more than 4. Since we are able to bound the size of read and write sets by a small constant, we conclude that validation will be fast, the validation time essentially being proportional to the degree of concurrency.

Another important consideration is the time to complete the validation and write phases as compared to the time to complete the read phase (this point was mentioned in Section 4). B-trees are implemented using some paging algorithm, typically least recently used page replaced first. The root page and some of the pages on the first level are normally in primary memory; lower level pages usually need to be swapped in. Since insertions always access a leaf page (here, we call a page on the lowest level a leaf page), a typical insertion to a B-tree of depth d will cause $d - 1$ or $d - 2$ secondary memory accesses. However, the validation and write phases should be able to take place in primary memory. Thus we expect the read phase to be orders of magnitude longer than the validation and write phases. In fact, since the "densities" of validation and write phases are so low, we believe that the serial validation algorithms of Section 4 should give acceptable performance in most cases.

Our final and most important consideration is determining how likely it is that one insertion will cause another concurrent insertion to be invalidated. Let the B-tree be of order m (m odd), have depth d, and let n be the number of leaf pages. Now, given two insertions I_1 and I_2, what is the probability that the write set of I_1 intersects the read set of I_2? Clearly this depends on the size of the write set of I_1, and this is determined by the degree of splitting. Splitting occurs only when an insertion is attempted on an already full page, and results in an insertion to the page on the next higher level. Lacking theoretical results on the distribution of the number of keys in B-tree pages, we make the conservative assumption that the number of keys in any page is uniformly distributed between $(m - 1)/2$ and $m - 1$ (this is a conservative assumption since it predicts storage utilization of 75 percent, but theoretical results do exist for storage utilization [15], which show that storage utilization is about 69 percent—since nodes are on the average emptier than our assumption implies, this suggests that the probability of splitting we use is high). We also assume that an insertion accesses any path from root to leaf equally likely. With these assumptions we find that the write set of I_1 has size i with probability

$$p_s(i) = \left(\frac{2}{m+1}\right)^{i-1}\left(1 - \frac{2}{m+1}\right).$$

Given the size of the write set of I_1, an upper bound on the probability that the read set of I_2 intersects the subtree written by I_1 is easily derived by assuming the maximal number of pages in the subtree, and is

$$p_I(i) < \frac{m^{i-1}}{n}.$$

Combining these, we find the probability of conflict p_C satisfies

$$p_C = \sum_{1 \le i \le d} p_s(i) p_I(i)$$

$$< \frac{1}{n}\left(1 - \frac{2}{m+1}\right) \sum_{1 \le i \le d} \left(\frac{2m}{m+1}\right)^{i-1}.$$

For example, if $d = 3$, $m = 199$, and $n = 10^4$, we have $p_C < 0.0007$. Thus we see that it is very rare that one insertion would cause another concurrent insertion to restart for large B-trees.

7. CONCLUSIONS

A great deal of research has been done on locking approaches to concurrency control, but as noted above, in practice two control mechanisms are used: locking and backup. Here we have begun to investigate solutions to concurrency control that rely almost entirely on the latter mechanism. We may think of the optimistic methods presented here as being orthogonal to locking methods in several ways.

(1) In a locking approach, transactions are controlled by having them wait at certain points, while in an optimistic approach, transactions are controlled by backing them up.
(2) In a locking approach, serial equivalence can be proved by partially ordering the transactions by first access time for each object, while in an optimistic approach, transactions are ordered by transaction number assignment.
(3) The major difficulty in locking approaches is deadlock, which can be solved by using backup; in an optimistic approach, the major difficulty is starvation, which can be solved by using locking.

We have presented two families of concurrency controls with varying degrees of concurrency. These methods may well be superior to locking methods for systems where transaction conflict is highly unlikely. Examples include query-dominant systems and very large tree-structured indexes. For these cases, an optimistic method will avoid locking overhead, and may take full advantage of a multiprocessor environment in the validation phase using the parallel validation techniques presented. Some techniques are definitely needed for determining all instances where an optimistic approach is better than a locking approach, and in such cases, which type of optimistic approach should be used.

A more general problem is the following: Consider the case of a database system where transaction conflict is rare, but not rare enough to justify the use of any of the optimistic approaches presented here. Some type of generalized concurrency control is needed that provides "just the right amount" of locking versus backup. Ideally, this should vary as the likelihood of transaction conflict in the system varies.

REFERENCES

1. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Inf. 1*, 3 (1972), 173–189.
2. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. 9*, 1 (1977), 1–21.

3. ELLIS, C. S. Concurrency search and insertion in 2-3 trees. *Acta Inf. 14*, 1 (1980), 63–86.
4. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
5. GRAY, J. Notes on database operating systems. In *Lecture Notes in Computer Science* 60: *Operating Systems*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, Berlin, 1978, pp. 393–481.
6. KUNG, H. T., AND LEHMAN, P. L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*, 3 (Sept. 1980), 354–382.
7. KUNG, H. T., AND PAPADIMITRIOU, C. H. An optimality theory of concurrency control for databases. In *Proc. ACM SIGMOD 1979 Int. Conf. Management of Data*, May 1979, pp. 116–126.
8. LAMPORT, L. Towards a theory of correctness for multi-user data base systems. Tech. Rep. CA-7610-0712, Massachusetts Computer Associates, Inc., Wakefield, Mass., Oct. 1976.
9. LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. Submitted for publication.
10. MILLER, R. E., AND SNYDER, L. Multiple access to B-trees. Presented at Proc. Conf. Information Sciences and Systems, Johns Hopkins Univ., Baltimore, Md., Mar. 1978.
11. PAPADIMITRIOU, C. H., BERNSTEIN, P. A., AND ROTHNIE, J. B. Computational problems related to database concurrency control. In *Conf. Theoretical Computer Science*, Univ. Waterloo, 1977, pp. 275–282.
12. PAPADIMITRIOU, C. H. Serializability of concurrent updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
13. SAMADI, B. B-trees in a system with multiple users. *Inf. Process. Lett. 5*, 4 (Oct. 1976), 107–112.
14. STEARNS, R. E., LEWIS, P. M., II, AND ROSENKRANTZ, D. J. Concurrency control for database systems. In *Proc. 7th Symp. Foundations of Computer Science*, 1976, pp. 19–32.
15. YAO, A. On random 2-3 trees. *Acta Inf. 2*, 9 (1978), 159–170.

Received May 1979; revised July 1980; accepted September 1980