A Non-Two-Phase Locking Protocol for Concurrency Control in General Databases.

Partha Dasgupta & Zvi M Kedem
Department of Computer Science
State University of New York
Stony Brook, NY 11794

1. Introduction.

A database is viewed as a collection of data objects which can be read or written by concurrent transactions. Interleaving of updates can leave the database in an inconsistent state. A sufficient condition to guarantee consistency of the database is serializability of the actions (reads or writes) performed by the transactions on the data items, that is, the interleaved execution of the transaction should be equivalent to some serial execution of the the transactions [1,2,7]. Here we will assume serializability as the criterion of correctness.

The 2-phase locking protocol is a well known protocol which produces serializable logs (Eswaran[4]). In database concurrency control we are not interested in all possible serializable logs. We are interested in logs which maximize allowable concurrency. However, 2-phase locking does far scoring high in this criterion. There are serializable logs allowing far more concurrency than any 2-phase locked log.

To achieve greater concurrency, we need to have more information about transaction behavior. One approach is to structure the database as a DAG (directed acyclic graph) (Kedem[5]). In this case non-2-phase behavior is attainable, and due to exploitation of the structure of the database to constrain transaction behavior it appears to provide higher concurrency. Another approach is to know in advance the readset and writeset of the transactions. This approach has been used for deadlock avoidance but not for gaining on concurrency [1]. Knowing the exact readset (or writeset) of a transaction is not always feasible, however a superset of the readset (or writeset) can be statically determined. We will assume this strategy and demonstrate how this information can be used to achieve higher concurrency.

2. The Algorithm.

A transaction T acts upon a set of data elements D. A data element x ∈ D is in the readset (Rd) of a transaction T (that is, x ∈ Rd(T)) if the transaction does a read operation on x. Any element written by the transaction T is contained in the writeset (Wr) of T. Note that neither Rd(T) need be a subset of Wr(T) or vice versa, and Rd(T) and Wr(T) may be supersets of the data items actually read and written by the transaction T.

2.1 Locking.

Each transaction is required to declare its readset and writeset to the transaction manager before it issues any actions. Since the readset (and writeset) may be supersets of the data items actually read (and written), they can be statically determined during query compilation.

<table>
<thead>
<tr>
<th>old lock</th>
<th>WHITE</th>
<th>BLUE</th>
<th>GREEN</th>
<th>YELLOW</th>
<th>RED</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>BLUE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>GREEN</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>YELLOW</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>RED</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

The algorithm uses five types of locks, White, Blue, Green, Yellow and Red. The White and Blue locks are the weakest locks. They are used as markers, and are compatible with all other locks (see Fig 1). The Green lock is the shared lock used for reading. The Yellow locks are used for locking data items which will be updated at a later stage. The Red lock is the exclusive lock used for writing, and is compatible only with the White and Blue locks. Neither the Green locks nor the Red locks are held over extended lengths of time. Only Yellow, Blue and White locks exist nearly as long as the transaction does.

A transaction (T), upon arrival, declares its readset Rd(T), and its write set Wr(T) to the transaction manager. The Transaction Manager schedules the transaction and obtains all the necessary locks. Then the transaction manager starts the transaction. The transaction, upon commencement is not required to make any locking requests (i.e. locking is transparent to the transaction).

2.2 Transaction Manager Actions.

The following is a outline of how a Transaction Manager handles a transaction, after it arrives on the system.

→ Arrival Point (Transaction T arrives)
   i) Get Yellow locks on Wr(T),
   ii) Get Green locks on Rd(T) ∪ Wr(T)
   iii) Do validation and lock inheritance processing (explained later)

→ Locked Point
   iv) Read values of Rd(T) into local storage,
   v) Downgrade all Green locks to White locks,
   vi) Start transaction processing.

→ Start Point
   i) T commences processing,
   ii) if T issues read(x), then return the value of x from local storage,
   iii) if T issues write(x), then update x in local storage.

This research was partially supported by NSF under grants MCS 81-04882 and MCS 81-10097.
of the above (except the setting of a lock). These algorithms can be executed concurrently with all the activities of the other given locks (and Blue) locks on all the data items. While (and
restarted. If the transaction passes validation, then the Transaction Manager has to acquire some White and Blue locks. T is Blue) locked by transactions in After(T). Finally all transactions T' are added to After(T).

The validation is simply checking whether Before(T) \cap After(T) is empty. If not then the transaction is rescheduled or restarted. If the transaction passes validation, then the Transaction Manager has to acquire some White and Blue locks. T is given White (and Blue) locks on all the data items White (and Blue) locked by transactions in After(T). Finally all transactions in Before(T) get White (Blue) locks on the readset (writeset) of T, and on all data items White (Blue) locked by T.

2.4 Deadlocks

Deadlocks, though not absent, are easy to deal with. Two forms of deadlocks are possible in this protocol. Waiting for locks to be granted could lead to a deadlock. However as all the locks that need to be obtained are known in advance, this form of deadlock can be avoided by using the all at once strategy used in Operating Systems.

The other form of deadlock is unavoidable. As a transaction T waits for a transaction T' in After(T) to get validated, transaction T' could be waiting for T in After(T') to be validated, leading to a deadlock. However as this deadlock spans only those transactions which are waiting to be validated, we can argue that due to the small number of transactions in this predicament, the chances of a deadlock is low, and so is the cost of detecting such deadlocks. Also, as these transactions have not started any processing, aborting any one of them (to break deadlocks) will not be costly.

3 Properties

The proof of correctness is omitted for spaces restrictions. We present some properties of the protocol, as exemplified by the Lemmas used to prove its correctness. The precedence relation amongst transactions are caused by read-write, write-read and write-write conflicts (see Bernstein[1]). These conflicts happen when the transaction actually reads or writes. However for ease of modeling we will assume that the arcs are created earlier, after the conflicting locks are obtained. As just obtaining a lock will not really cause an arc, especially if the transaction gets rescheduled after failing validation, we define that the arc is created when the transaction reaches its locked point.

Definition

If T_i \rightarrow T_j is an arc, then this arc was created when both T_i and T_j reached locked point, and was created by the transaction to reach locked point last.

Lemma 1

If T_i \rightarrow T_j and T_j reached its locked point before T_i did then the arc can only be caused if T_j gets a Yellow lock on data item, and then T_j gets a Green lock on it (and finally T_j converts its Yellow lock to a Red lock).

Lemma 2

If T_i \rightarrow T_j and T_j reached its locked point before T_i did, then T_j is active when T_i reaches its locked point.

Lemma 1 and Lemma 2 show that unlike 2-phase locking the precedence graph can grow "backwards" (see §4). However in order that a transaction T_j, which arrives later than transaction T_i, may actually precede T_i in the precedence order, T_i must be active when T_j arrives.

Lemma 3

If T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n is a chain of transactions, and T_1 is active, then

i) T_i possesses White locks on Rd(T_n) (i.e. Rd(T_n) \subseteq WLS(T_i)).

ii) T_i possesses Blue locks on Wr(T_n) (i.e. Wr(T_n) \subseteq BLS(T_i)).

Lemma 3 shows the most important property of this protocol. This implies that if a transaction T_i is active, it "knows" about the read and write set of all transactions that come after T_i. This property is used to achieve serializability by causing a validation conflict when a cycle is created by some transaction.

Informally, a cycle in the \rightarrow relation is caused if a transaction T_k arrives and takes position before a transaction T_i in the precedence graph as also after a transaction T_{k-1} and T_{k-1} is in the forward path of a chain from T_i.
Since \( T_2 \) has \textit{White} (and \textit{Blue}) lock on the read (and write) sets of \( T_{k-1} \) when \( T_k \) conflicts with \( T_{k-1} \), due to any cause, \( T_k \) becomes a member of \textit{Before}(\( T_k \)). Also when \( T_k \) comes ahead of \( T_1, T_2 \) becomes a member of \textit{After}(\( T_k \)), and the cycle is foiled at validation (as the intersection of \textit{Before}(\( T_k \)) and \textit{After}(\( T_k \)) is not empty).

4 Discussion.

This protocol differs significantly from the 2-phase locking protocol in the way the precedence (\( \rightarrow \)) relation may grow. In the 2-phase locking scheme if a transaction reaches its locked point, all transactions which come before \( T \) in the precedence order must have reached their locked points. That is, the chain can only grow in the forward direction. In fact this is the property of the 2-phase locking scheme that ensures serializability.

In this protocol, the chain of transactions under the precedence order can grow in both directions. Suppose transaction \( T \) has reached its locked point and possesses a \textit{Yellow} lock on \( x \). Now a new transaction \( T' \) arrives and gets a \textit{Green} lock on \( x \). When \( T' \) reaches locked point, the arc \( T' \rightarrow T \) is born. Now, even after \( T \) terminates, as long as \( T' \) is active, \( T'' \) may come and place itself before \( T' \).

Thus the basic mechanism by which 2-phase locking ensures serializability is not present in this scheme. Serializability is ensured, in this case by the \textit{Blue} and \textit{White} locks, and the validation procedure. Intuitively, if \( T_1 \rightarrow \ldots \rightarrow T_n \) is a chain of transactions, then \( T_1 \) "knows" about \( \text{Rd}(T_2) \) and \( \text{Wr}(T_2) \), because it has \textit{White} and \textit{Blue} locks, respectively, on these data items. If any transaction \( T' \) attempts to read any data item in \( \text{Wr}(T_2) \) (or write any item in \( \text{Rd}(T_2) \)) then due to the "trigerring" caused by \textit{Green} (\textit{Yellow}) locking of a \textit{Blue} (\textit{White}) locked item, \( T_1 \) becomes a member of \textit{Before}(\( T' \)) and \( T_1 \) inherits \textit{White} (\textit{Blue}) locks on \( \text{Rd}(T_2) \) (\( \text{Wr}(T') \)). Thus information about the read and write sets flow up a chain in the form of "inherited" \textit{White} and \textit{Blue} locks. Now if \( T' \) may cause a cycle in the \( \rightarrow \) relation by attempting to read an item \textit{Yellow} locked by \( T \), then \( T \) would become a member of \textit{After}(\( T' \)) and violate the validation constraint.

4.1 An Example

Here is an example of a non-2-phase locked log that is allowed by this protocol. This is a serializable log which is termed a \textit{non-strictly-serializable} log. A \textit{strictly serializable} log is a log in which no non-interleaved transaction appear in the same order as they would appear in the serial order (Bernstein[4]). In some serializable logs this is not the case. However all other known concurrency control protocols produce subsets of \textit{strictly serializable} logs.

\[
\begin{align*}
\text{Log} & : R_1(x) \ R_2(y) \ W_1(y) \ R_2(z) \ W_2(z) \\
\text{Serial order} & : T_2 \rightarrow T_1 \rightarrow T_3 \\
\end{align*}
\]

Note that in this history the transactions \( T_1 \) and \( T_3 \) do not interleave and \( T_2 \) completes execution before \( T_3 \) starts. But in the serial order, \( T_2 \) comes before \( T_1 \). This property violates \textit{strict} serializability and thus this log is \textit{non-strictly serializable}.

The detailed trace of this log is shown in Fig. 2.

Thus we conclude that this protocol can achieve more concurrency due to the information available to the transaction manager about the transaction readsets and writset.

5 Acknowledgements

We are indebted to the anonymous referees for their indepth comments and very helpful suggestions.

6 References