Reverse Serializability as A Correctness Criterion for Optimistic Concurrency Control

Hyeokmin Kwon and Songchun Moon

Department of Information and Communication Engineering
Korea Advanced Institute of Science and Technology
207-43, Cheongryang, Dongdaemun, Seoul 130-012, Republic of Korea

If buffer retention effect is taken into account, validation schemes for optimistic concurrency control (OCC) should be approached in some different point of view: rather than killing the conflicting transactions immediately in the middle of their execution, it would be better to allow them to run to their completion for bringing the required data objects into main memory. In this respect, we propose a new validation scheme for OCC called reordering serial equivalence (RSE) by introducing reverse serializability, which ensures the correctness of RSE when we allow the serialization in the reverse order of transactions’ commits.

1. Introduction

With ever-increasing CPU and I/O processing speeds, a database management system is required to support a substantially enhanced level of concurrency. This may cause a high level of data contention. High data contention could also arise from the use of long or complex transactions. In order to conform to this high data contention, there is a new trend to make use of the access invariance [4], which means that restarted transaction will with high probability perform the same operations on the same set of objects as its first run. With sufficient buffer, data objects referenced by the aborted transactions can continue to be kept in memory and be available for accesses during rerun without I/O operations. As a result, it is possible to reduce the abort probability significantly during rerun [4, 5]. With the advent of this concept, there is a renewed interest in OCC, since it is inherently effective to prefetch the required data objects without blocking during the first run. Most variations [2-3, 6] of the original OCC [1] focused mainly on how to reduce the restart overhead. They achieved this by simply aborting the conflicting transactions as early as possible.

However, if buffer retention effect is taken into account, validation schemes for OCC should be approached in some different point of view: rather than killing the conflicting transactions immediately in the middle of their execution, it would be better to allow them to run to their completion for bringing the required data objects into main memory. In this respect, attention should be paid to design a validation scheme capable of recognizing more serializable histories after completion of transaction executions. We, therefore, introduce a new correctness criterion for OCC called reverse serializability, which allows the serialization in the reverse order of transactions’ commits.

2. Related Works and Problems

The OCC schemes [1-3, 6] are designed to get rid of locking and deadlock detection overhead in conventional locking schemes. These schemes consist of three phases: read, validation, and write. Transactions are executed without any restriction during the read phase, but before a transaction is allowed to commit, it is validated to ensure that it preserves database consistency. If validation succeeds, the transaction is allowed to commit. If not, a conflict resolution strategy should be applied. The OCC schemes can be classified into two groups on the basis of their validation schemes: kill-based validation and die-based validation. In the former approach, committing transactions force conflicting transactions to be restarted immediately. The broadcast OCC schemes in [2, 3] are included in this group. In the latter approach, a transaction is restarted only when it issues commit request and any transaction in conflict with this transaction previously committed. The OCC schemes and the snapshot validation [6] are based on this approach. However, the snapshot validation is somewhat different with the original OCC. Whenever any transaction \( T \) terminates, all active transactions validate against
$T_i$'s write set. If any transaction fails in validation, it dies by itself. Therefore, it is possible to detect conflicting transactions and abort them in the middle of their execution like the broadcast OCC.

Although it is possible to save processing time of hardware resources by reducing the amount of wasted work in the broadcast OCC and the snapshot validation, there appear some trade-offs due to the effect of buffering. That is, these schemes are not capable of bringing all required data objects in main memory during the first run. Therefore, it is not well suitable to the concurrency control (CC) schemes that could make use of buffer retention effect. In practice, the original OCC outperforms the well-refined broadcast OCC in studies of [4, 5], where the buffer retention effect is considered. However, since the original OCC has a serious weakness with respect to validation, there is a high risk of transaction abort during the first run. To illustrate this, let us consider a set of transactions that are concurrently executed (see Example 1).

**Example 1 (Limitation in Recognizing Serializable Histories and Problem of Early Restart):**

![Figure 1 Serializable Execution Schedule](image)

In this execution schedule, transactions $T_2$ and $T_3$ conflict with transaction $T_1$, since the write set of $T_1$ overlaps with the read sets of $T_2$ and $T_3$. Under the broadcast OCC and the snapshot validation, $T_2$ should be aborted at time $t_2$ (during validation phase of $T_1$) and $T_3$ (after write phase of $T_1$), respectively, since it has read $x$ before $T_1$ writes it. On the other hand, $T_3$ is allowed to commit, since it reads $y$ after $T_1$ has written $y$. In these schemes, there is a drawback that all required data objects are not prefetched into main memory due to the early restart of $T_2$. Using the validation scheme of the original OCC, both $T_2$ and $T_3$ should be aborted at time $t_3$ and $t_4$, respectively. However, all data objects are prefetched during the first run with extra wasted work compared with the broadcast OCC and the snapshot validation. Note that it is not impossible to serialize $T_2$ against $T_1$, if $T_2$ does not read any data object that has already been updated by $T_1$ and the write set of $T_2$ does not overlap with the read set and write set of $T_1$, it is possible for $T_2$ to come before $T_1$ in an equivalent serial order. Therefore, it is possible to serialize the execution schedule shown in Figure 1, which is in the order of $T_1 \rightarrow T_2 \rightarrow T_3$. Nevertheless, none of them allows $T_2$ to commit.

### 3. Reordering Serial Equivalence

We propose a new validation scheme for OCC called RSE that is capable of accepting more serializable histories than [1-3, 6] by introducing reverse serializability, which ensures the correctness of RSE when we allow the serialization in the reverse order of transactions’ commits. Usually OCC schemes try to place a transaction at the end of the serial schedule existing so far. If this is not possible, the transaction is restarted. However, in this scheme the transaction, though it conflicts with already committed transactions, is allowed to commit if it is possible to rearrange the equivalent serial order in the reverse order of their commits. This benefit is achieved at the cost of marking validation. For effectively taking advantage of the buffer retention effect, all transactions are allowed to run to their validation phases. Each transaction $T_i$ keeps track of the read set, $RS_i$, of objects read from the database and the write set, $WS_i$, of objects written by it. In this paper we will assume that $RS_i$ is ordered in the sequence of its action on data objects and the data objects in a transaction's $WS$ are contained also in its $RS$ for the sake of simplicity. The algorithm proposed can be refined if this is not the case.

The validation of RSE consists of the normal validation and the marking validation. The normal validation ensures serializability in the order of transactions' commits by assuring that the read sets of active transactions are always clean. By clean, it means that some transaction $T_i$ does not read any data object before any other transactions, which commit ahead of $T_i$, write it. On the other hand, the marking validation ensures reverse serializability for allowing transactions to commit when they are not serializable in the order of their commits. During the normal validation of $T_i$ against any active transaction $T_j$, if the cleanness of $RS_j$ is not guaranteed, it is impossible to serialize them in the order of their commits. To check the reverse serializability during $T_j$'s validation, $T_i$ takes note of $EOT_i$ (end of transaction $T_i$) in $RS_j$. In this case, $T_i$ is termed a marking transaction and $T_j$ is termed a marked transaction in this paper. A transaction that is not a marked transaction is termed a normal transaction. Normal transactions are always allowed to commit but marked transactions are allowed to commit only if they succeed in marking validation. The basic concept of marking validation is as follows. To serialize marking transaction $T_i$ and its corresponding marked
transaction $T_j$ in the order of $T_j \rightarrow T_n$, the following three rules should be observed. (Note that marking transaction always commits before its corresponding marked transaction commits.)

**Rule 1 (No Reverse Read):** $T_j$ should not read any data object after $T_i$ has modified it.

**Rule 2 (No Dirty Read):** $T_j$ should not write any data object that has been read by $T_i$.

**Rule 3 (No Overwrite):** $T_j$ should not write any data object that has been written by $T_i$.

Rule 1 is guaranteed if $WS_i$ does not intersect with data objects that exist between the marking point of $EOT_i$ in $RS_i$ and the end of $RS_i$. Rules 2 and 3 are guaranteed if the $WS_i$ does not intersect with $RS_i$. (Note that the test $WS_i \cap RS_i$ includes $WS_i \cap RS_i$ under our assumption.) The marking validation ensures the reverse serializability by assuring that these three rules are observed. For allowing marked transaction $T_j$ to commit, these three rules should be satisfied against all its marking transactions.

Note that a transaction might become a marking transaction of other transactions irrespective of the condition of their conflicts. To illustrate this, assume that $T_i$ has already committed and marked $T_j$. When $T_j$ is allowed to commit later, the serial order between them is $T_j \rightarrow T_i$. However, assume that a normal transaction $T_f$ commits before $T_j$. Since $T_f$ is a normal transaction, it is allowed to commit in serial order $T_f \rightarrow T_j$. Therefore, the total serial order should be $T_f \rightarrow T_j \rightarrow T_i$. In this case, it is not necessary to validate $T_j$ against $T_i$ if $T_j$ commits ahead of $T_i$, since it does not matter whether or not $T_j$ has read any data object before $T_i$ writes it. Instead, $T_j$ takes note of $EOT_i$ in $RS_i$ for $T_f$ to check the reverse serializability during its validation. The procedures for RSE validation are shown in detail in Algorithm 1.

Each transaction $T_j$ keeps track of the set of transactions that have marked $T_j$ in $Marker_set$. Data structure $Marker_set$ maintains the transactions that have been marked by $T_j$ but have not yet committed. It is maintained for effectively managing $RS_i$, $WS_i$, and $Marker_set$. Whenever any transaction $T_j$ commits or is restarted, $T_j$ is deleted in its corresponding $Marker_sets$ if they exist. $RS_i$, $WS_i$, and $Marker_set$ is cleaned up if $Marker_set$ becomes empty. These procedures are omitted for the simplicity of algorithm but they can be implemented during validation easily and efficiently.

There is another terminology, mark inheritance, which means that $Marker_set$ when marked transaction $T_i$ marks other transaction $T_j$. Assume that a marked transaction $T_i$ marks $T_j$. In this case the marking transactions of $T_i$ should stand behind $T_j$ in serial order. When $T_j$ is allowed to commit later, it should stand before $T_i$. Therefore, $T_j$ should stand before all marking transactions of $T_i$. Hence, when a marked transaction marks other transaction, mark inheritance is needed.

**Algorithm 1 (RSE Validation):**

```
Validate(T) {
  if $T$ is a marked transaction then
    if Marking_validate($T$) = REJECT then
      return(ABORT);
  For each marked transaction $T_j$ among active transactions {
    if $Marker_set_j \supseteq Marker_set_i$ then
      if $(WS_i \cap RS_j) \neq NULL$ then
        { Mark_inherit($T_i$, $T_j$); mark $EOT_i$ in $RS_i$; } -- note 1
      else
        mark $EOT_i$ in $RS_i$; } -- note 2
  For each normal transaction $T_j$ among active transactions {
    if $(WS_i \cap RS_j) \neq NULL$ then
      { mark $EOT_i$ in $RS_j$; if $T_j$ is a marked transaction then
        Mark_inherit($T_i$, $T_j$); } }
  return(COMMIT); } /* end of Validate

Marking_validate($T$) {
  for each $T_i$ in $Marker_Set$, {
    if ($((WS_i \cap After_EOT_i) \cap RS_i) = NULL || ((RS_i \cap WS_i) = NULL))$ then
      return(REJECT); }
  return(ACCEPT); }

Mark_inherit($T_i$, $T_j$) {
  $Marker_set_j = Marker_set_j \cup Marker_set_i$;
  for each $T_k$ in $marker_set_i$,
    $Marked_set_k = Marked_set_k \cup T_j$; }

Expressions:
After_EOT_i, RS_i: Set of data objects that exist between the marking point of $EOT_i$ in $RS_i$ and the end of $RS_i$.
```

If transaction $T_i$ has become a marking transaction of $T_j$ only through mark inheritance, $EOT_i$ is considered to be marked at the starting point of $RS_i$. Although RSE can recognize more serializable histories than [1-3, 6], it could induce much overhead due to marking validation and mark inheritance. In particular, the mark inheritance might be a crucial point on the performance of RSE, since it may cause
the Marker_set to be large. However, if a marked transaction commits soon after its corresponding marking transactions, the overhead of RSE would not be so large. We have a strong expectation that this is a normal case in the sense that \( T_i \) would be a much progressed transaction if \( T_i \) is marked by \( T_j \) \((RS_i \wedge WS_j \neq \text{NULL})\) and thus it is very likely that \( T_i \) would commit soon after its corresponding marking transactions. The marking validation guarantees the reverse serializability by ensuring that the Rules 1, 2 and 3 presented in this section are observed. It means that marked transactions can be regarded as committed before their corresponding marking transactions. Therefore, the correctness proof of this algorithm is simply reduced to the proof of original OCC only if the following holds.

**Lemma 1 (Correctness of RSE Validation Algorithm):** If any transaction \( T_i \) has committed before \( T_j \) but follows \( T_j \) in serial order, then there exists a marking operation of \( T_j \) against \( T_i \) for \( T_i \)'s marking validation.

**Proof:** We prove it by induction. Let \( S \) be a set of transactions that have already committed and their equivalent serial order be arranged in the order of their commits if there is no marking operation between two transactions.

1. **Basic step:** Let \( S \) be \( \{T_1, T_2\} \) and their serial order be \( T_1 \rightarrow T_2 \). Lemma 1 is trivially correct, since \( T_2 \) is a marking transaction of \( T_1 \) if \( T_1 \) is a marked transaction.

2. **Inductive step:** Let \( S \) be \( \{T_1, T_2, \ldots, T_n\} \) and their serial order be \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \). Lemma 1 is correct if \( T_n \) is a marked transaction, \( T_n \) should stand somewhere before \( T_{n-1} \). Let \( T_n \) stand before some transaction \( T_i \) (Figure 2). If \( T_i \) has marked \( T_n \) through mark inheritance, some transaction that stands ahead of \( T_i \) in serial order should have marked \( T_n \). Therefore, it means that \( T_i \) has marked \( T_n \) by itself.

![Figure 2 Serial Order](image)

Since all transactions behind \( T_i \) have committed before \( T_n \) but follow \( T_n \) in serial order, they should have marked \( T_n \). We now prove this. Assume that some transaction \( T_k \) behind \( T_i \) has not marked \( T_n \). Case1) If \( T_i \) is not a marked transaction of \( T_k \); In this case Marker_set includes \( T_k \) but Marker_set does not include \( T_i \) at the time of \( T_k \)'s validation. Therefore, \( T_k \) should have marked \( T_n \) (Case2) If \( T_i \) is a marked transaction of \( T_k \); If \( T_i \) has marked \( T_n \) either in note 1 or note 3, it should inherit its marker set (it includes \( T_k \)) to \( T_n \). Therefore, \( T_k \) should have marked \( T_n \). If \( T_i \) has marked \( T_k \) in note 2, since \( T_k \) has been marked by all transactions behind it at the time of its validation (induction hypothesis), some transaction before \( T_i \) should have marked \( T_n \). Which causes a contradiction to the basic assumption. Therefore, all transactions behind \( T_i \) should have marked \( T_n \). Hence, Lemma 1 holds for \( n+1 \) transactions in \( S \). Hence, if any transaction \( T_n \) has committed before \( T_i \) but follows \( T_i \) in serial order, then there exists a marking operation of \( T_n \) against \( T_i \).

### 4. Conclusions

We propose a new validation algorithm, RSE, that is capable of recognizing more serializable histories than [1-3, 6] by introducing reverse serializability, which allows serialization in the reverse order of transactions' commits. We do not insist that RSE would outperform other OCC schemes. However, we do claim that the approach used in RSE is well matched with the effect of buffering and it shows what kinds of points should be taken into account to allow serialization in the reverse order of transactions' commits. Furthermore, if RSE is well refined, these variations may be capable of outperforming other OCC schemes. For example, if the number of transactions in Marker_set is restricted in a certain number, it may be possible to accept more serializable histories than other OCC schemes at the reasonable cost of overhead. We are going to design these kinds of variations based on RSE and evaluate their performance as well as the performance of RSE using a simulation model.

### References