Partial Rollback in Object-Oriented/Object-Relational Database Management Systems

Won-Young Kim[1] [*] , Kyu-Young Whang[1], Byung Suk Lee[2], Young-Koo Lee[1], and Ji-Woong Chang[1]
[1]Computer Science Department and
Advanced Information Technology Research Center (AITrc)
Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
{wykim, kywhang, yklee, jwchang}@mozart.kaist.ac.kr
[2]Department of Computer Science
University of Vermont, Burlington, Vermont, USA
bslee@cs.uvm.edu

ABSTRACT

In a database management system (DBMS), partial rollback is an important mechanism for canceling only part of the operations executed in a transaction back to a savepoint. Partial rollback complicates buffer management because it should restore the state of the buffers as well as that of the database. Several relational DBMSs (RDBMSs) currently provide this mechanism using page buffers. However, object-oriented or object-relational DBMSs (OO/ORDBMSs) cannot utilize the partial rollback scheme of RDBMSs as is because, unlike RDBMSs, many of them use a dual buffer consisting of an object buffer and a page buffer. In this paper, we propose a thorough study of partial rollback schemes of OO/ORDBMSs with a dual buffer. First, we classify the partial rollback schemes of OO/ORDBMSs into a single buffer-based scheme and a dual buffer-based scheme by the number of buffers used to process rollback. Next, we propose four alternative partial rollback schemes: a page buffer-based scheme, an object buffer-based scheme, a dual buffer-based scheme using a soft log, and a dual buffer-based scheme using shadows. We then evaluate their performance through simulations. The results show that the dual buffer-based partial rollback scheme using shadows provides the best performance. Partial rollback in OO/ORDBMS has not been addressed in the literature; yet, it is a useful mechanism that must be implemented. The proposed schemes are practical ones that can be implemented in such DBMSs.

[*]Current address: Real-Time DBMS Team, Switching & Transmission Technology Lab., Electronics and Telecommunications Research Institute (ETRI)

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Obeject-oriented databases, Transaction processing*

General Terms

Algorithms

Keywords

parital rollback, dual buffer, object-relational DBMS, recovery

1. INTRODUCTION

Transaction rollback is a mechanism for canceling the effect of operations executed within a database transaction[1, 4, 7]. It revokes *all* updates from the beginning of a transaction, often incurring significant cost. In order to alleviate this problem, a *partial rollback* scheme is used, which recovers a transaction to a *savepoint* [4, 7]. A savepoint can be set by either the system or the user at any point in time during a transaction. Applications executing long transactions often need to annul operations executed erroneously or producing unsatisfactory results before the transactions terminate, and to continue from that point on. Partial rollback is very useful in such cases.

Several database management systems (DBMSs) already support partial rollback. Some, typified by relational DBMSs (RDBMSs), are using only page buffers for partial rollback; others are using both object buffers and page buffers, called *dual buffers* in combination. This difference in buffer organizations has a large impact on the performance, as indicated by Subramanian and Krishnamurthy [8]. As we see it, the DBMS industry is moving toward supporting a dual buffer as it is making a transition from RDBMSs to OO/ORDBMSs. Some commercial systems [9, 10] and research prototypes [11] already support dual buffers.

Unfortunately, however, despite the existence of those DBMSs supporting partial rollback, few research results have been published about their performance in relation to the buffer management for partial rollback. Moreover, even those published assume RDBMSs using page buffers [7, 12, 13]. To the extent of the authors' knowledge, there has been no research in the past proposing partial rollback schemes for OO/ORDBMSs with a dual buffer. These shortcomings warrant the need for a comprehensive and comparative study of new partial rollback schemes for different buffer organizations.

In this paper, we propose a thorough study of partial rollback schemes of an OO/ORDBMSs with a dual buffer. We propose four alternative partial rollback schemes and compare their performance. We first classify them into the single buffer-based scheme (SB) and the dual buffer-based scheme (DB) by the number of buffers used to process rollback. In the SB scheme, rollback occurs to one buffer, which subsequently propagates to the other buffer. We further classify the SB scheme into the page buffer-based scheme (PB) and the object buffer-based scheme (OB) by the buffer rolled back. In the DB scheme, the OO/ORDBMS rolls back *both* the page buffer and the object buffer. Based on the method of maintaining recovery data in the object buffer, we further classify the DB scheme into a scheme using the soft log (DB-SL) and one using shadows (DB-SO). As for the performance comparison, we assess the four schemes (i.e., PB, OB, DB-SL, and DB-SO) using extensive simulation.

The rest of this paper is organized as follows. In Section 2 we survey the partial rollback schemes used in RDBMSs, which use page buffers only. In Section 3 we describe the dual buffer organization of the OO/ORDBMS and discuss its effect on partial rollback. In Section 4 we explain the four partial rollback schemes we propose for OO/ORDBMSs. In Section 5 we describe the experiments performed and present the results. Finally, we conclude the paper in Section 6.

2. RELATED WORK

A study on partial rollback schemes of OO/ORDBMSs has not been reported in the literature. Here, we overview the partial rollback schemes of RDBMSs. A partial rollback should recover the states of both the transaction execution and the database. A database state is recovered through a rollback to a savepoint using the log of performed updates. The transaction execution state is represented by such main memory data structures as locks, cursors, and file handles [4]. These data structures are updated frequently as locks are granted or released, as cursors are created or destroyed, and as files are opened or closed. Besides, their sizes are quite small compared with the database size. Thus, a DBMS typically stores a *snapshot* of them at each savepoint and uses the snapshots to recover the transaction execution states[4].

Setting a savepoint and performing a partial rollback in those DBMSs are done as follows [3, 7].

At a savepoint: The DBMS (1) records the transaction execution state and the SaveLSN – the log sequence number (LSN) of the last log record saved so far – in main memory and (2) returns a savepoint identifier to the user.

Using the identifier, the user can request a partial rollback to the savepoint, which is executed as follows.

During a partial rollback: The DBMS (1) undoes, in a reverse order, all updates performed after recording the SaveLSN associated with the savepoint and (2) restores the transaction execution state to the savepoint.

This scheme assumes RDBMSs using only page buffers. However, OO/ORDBMSs cannot utilize the partial rollback scheme of RDBMSs as is because, unlike RDBMSs, many of them use a dual buffer consisting of an object buffer and a page buffer.

3. DUAL BUFFER ORGANIZATION IN OO/ORDBMSS

From the perspective of a partial rollback scheme, one of the outstanding trends in the OO/ORDBMS architecture is the use of a dual buffer [8]. In this section, we describe the dual buffer organization and discuss its effect on partial rollback.

The objective of using the page buffer is to reduce the number of disk accesses. A DBMS manages objects in the page buffer in the unit of disk page. In contrast, the objective of using the object buffer is to accelerate the accesses to objects by caching them in the client to reduce the traffic between the client and server [5]. While the page buffer is shared by multiple users, the object buffer is not [6]. In a client-server environment where a server ships *objects* to a client, the server maintains the page buffer while the client maintains the object buffer [14].

To access an object in a dual buffer, the data page containing the object is swapped in from disk to the page buffer, and the object is in turn swapped in from the page buffer into the object buffer. If the object buffer becomes full, a victim is chosen among the buffered objects. If the chosen object is dirty (i.e., has been updated), it is swapped out to the page buffer, updating the page that contains the object. At this point, if the page is not in the page buffer, it must be read in again from disk, and replaces a page chosen as a victim. If the chosen page is dirty, it is swapped out to disk to reflect the updates in disk.

Since in an OO/ORDBMS the page buffer and the object buffer are extensions of the database, we must recover the states of the two buffers as well when partially rolling back the database. An OO/ORDBMS, like an RDBMS, can rollback the state of the page buffer and the disk using logs. However, partially rolling back the state of the object buffer necessitates an additional mechanism. For a total rollback after a transaction failure, the object buffer state can be simply ignored since the transaction terminates at that point. In contrast, a partial rollback to a savepoint requires recovering the object state to that of the savepoint, since the transaction continues from that point after the partial recovery.

Updates of objects in the object buffer are not recorded in a log, nor reflected on disk, thus hindering a partial rollback. As a solution, one may consider applying the partial rollback scheme of the RDBMS, that is, using the snapshots of the object buffer recorded at savepoints. However, this scheme incurs the substantial space and time overhead of saving the snapshots at every savepoint. To address the problem, we propose other schemes that recover the database including the object buffer in partial rollback of an OO/ORDBMS.

4. PARTIAL ROLLBACK SCHEMES OF OO/ORDBMSS

In this section, we propose four partial rollback schemes for OO/ORDBMSs: the page buffer-based scheme (PB), object buffer-based scheme (OB), dual buffer-based scheme with the soft log (DB-SL), and dual buffer-based scheme with shadows (DB-SO). We explain the concept, procedures, and advantages and disadvantages of each scheme.

4.1 Single Buffer-Based Partial Rollback Scheme (SB)

As mentioned in Introduction, the single buffer-based scheme uses either the page buffer or object buffer, but not both, for managing recovery data for partial rollback. Depending on which buffer is the target of recovery, we further classify it into the page buffer-based and the object buffer-based schemes. We explain these two schemes here.

4.1.1 *Page Buffer-Based Partial Rollback Scheme (PB)*

PB flushes updated objects from the object buffer to the page buffer at savepoints, and recovers the state of only the page buffer at a partial rollback. Disk-resident logs are used to recover the page buffer state. We call these logs *hard logs* to distinguish them from *soft logs* used in other schemes (OB and DB-SL).

At a savepoint: PB performs operations extended from those of the RDBMSs (described in Section 2) with the additional step of flushing the updated(i.e., dirty) objects in the object buffer to the page buffer.

During a partial rollback: PB (1) restores the objects in the page buffer using the hard log, and (2) deletes all objects in the object buffer because they might have been updated after the savepoint.

PB is easily implementable due to its reliance on hard logs only, i.e., without any other recovery data. However, frequent object buffer flushes incur the overhead of moving objects from the object buffer to the page buffer, and also from the page buffer to disk, leading to an increasing number of disk I/Os. We call this overhead the *forced-flush overhead.*

The forced-flush overhead may cause considerable performance degradation if savepoints are set frequently. In addition, the deletion of buffered objects reduces the hit ratio of the object buffer, causing more objects to be swapped in after a partial rollback. The performance degradation is more noticeable in the client-server environment where the cost of object transfer is relatively high.

4.1.2 *Object Buffer-Based Partial Rollback Scheme (OB)*

OB is meant to remove the potential problems of PB – the forced-flush overhead and the low hit ratio of the object buffer. It records in the soft log all updates occurring in the object buffer, and recovers the state of the object buffer directly using the log. In order to undo the updates of the objects swapped out prior to the partial rollback point, it swaps the objects back into the object buffer before applying the soft log records. It also records in the hard log all updates in the page buffer caused by the objects swapped out from the object buffer, but the hard log is used for a total, not partial, rollback. While hard logs are stored in a stable disk, soft logs are stored in main memory because they are no longer needed once a partial rollback or transaction execution is completed. Furthermore, soft log records keep undo information only, which suffices for performing partial rollbacks.

OB is different from the partial rollback scheme of RDBMSs primarily in two aspects: it uses the *soft log* recorded for the object buffer, and swapping occurs between the object buffer and the page buffer.

At a savepoint: OB performs the same operations as in RDBMSs (described in Section 2) except that is uses the soft log instead of the hard log.

During a partial rollback: OB (1) swaps in the objects swapped out prior to the partial rollback point, (2) undoes all updates of objects in the object buffer using the soft log, and (3) delete the used soft log records. Note that any swap-out's occuring due to the swap-in's in Step 1 are recorded in the hard log as normal updates.

OB indeed eliminates the aforementioned overhead of PB, and shows good performance provided that the object buffer is large enough to keep most objects that will be undone during partial rollbacks. However, since undoing updates on swapped-out objects requires swapping them back into the object buffer, other objects may have to be swapped out if main memory is not large enough. This incurs more disk I/Os. In addition, OB uses increasingly large main memory to store the soft log as more updates are performed within a transaction. We call this overhead the *memory overhead.*

4.2 Dual Buffer-Based Partial Rollback Scheme (DB)

The performance degradation of single buffer-based schemes occur because updated objects stored in one buffer are transferred to the other buffer at the time of either setting a savepoint (in PB) or performing a partial rollback (in OB). DB deals with this problem by managing and recovering the two buffers separately. This scheme utilizes hard logs for recovering updated objects in the page buffer and recovery data (soft logs or shadows) for recovering those in the object buffer.

In DB, objects are classified into three types based on their update and swapping activities between the savepoint and the partial rollback point as shown in Figure 1. In the figure, no swapping is allowed during a time interval marked with a thick line, and swapping is required during an interval marked with the recycling sign. Swapping activities during a time interval marked with a thin line are irrelevant to the type of the objects.

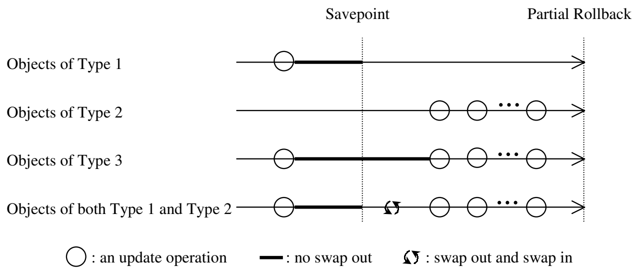


Figure 1: Object Types.

The recovery method differs for each type. Objects of Type 1 are those updated only before the savepoint and therefore need no recovery during a partial rollback. There is a caveat for the page buffer, however. In DB, updated objects are not flushed at a savepoint, but may be swapped out to the page buffer after the savepoint due to object buffer replacement. We call such objects *delayed objects.* The updates on those objects, which occurred before the savepoint, should not be undone. To embody this idea, when the swap-out of an updated object is recorded as an update in the hard log, the associated HardSaveLSN of the savepoint is stored together with the log record. Then, during a partial rollback, a log record whose HardSaveLSN is identical to that of the given savepoint is ignored.

Objects of Type 2 are those updated only after the savepoint. Since swapping before the update has no effect to either buffer, the recovery mechanism is determined by swapping after the update. There are three cases. First, if no swapping occurs at all, we simply discard the updated objects in the object buffer because the updates have not been reflected in the page buffer yet. Second, if only swap-out occurs, we recover the objects in the page buffer using the hard log. Note that the swapped-out objects do not exist in the object buffer anymore. Third, if both swap-out and swap-in occur, we recover the objects in the page buffer using the hard log and discard the objects in the object buffer. This recovery mechanism is essentially identical to that of PB.

Objects of Type 3 are those updated both before and after the savepoint and not swapped out until the point of the second update. They are recovered in the object buffer using the recovery data in the same manner as in OB. Note that the first and second updates occurring in the object buffer are recorded as no more than one update in the hard log because no swapping occurs between the two updates. For this reason, using the hard log restores the objects in the page buffer to either the state before the first update or after the second update, but never to the state at the savepoint. However, it does not cause a problem as long as the objects in the object buffer are restored correctly. The restored objects are eventually swapped out to replace the incorrect objects in the page buffer. In our work, we choose to rollback the page buffer to the state prior to the first update to simplify the entire recovery mechanism and to maintain the consistency with other recovery schemes.

The recovery mechanism becomes intricate when we consider the swapping activities of objects at different points in time after the savepoint. There are two cases. If objects updated before the savepoint are swapped out and back in before the second update, they are handled as objects of both Type 1 and Type 2 – Type 1 before the second update, and additionally Type 2 after the second update. Note that if the objects are swapped after the second update (without having been swapped before), they are handled as objects of Type 3. Any subsequent swapping does not affect the type.

In order to implement the recovery mechanisms described above, we use a data structure called the *savepoint updated object list (SUOL)*, a list of updated objects in the object buffer at a given savepoint. Using the SUOL, the type of an object is identified as follows: (1) objects that are in the SUOL of a savepoint but not updated in the object buffer yet are classified as Type 1; (2) objects that are not in the SUOL and updated after the savepoint are classified as Type 2; (3) objects that are in the SUOL and updated after the savepoint without being swapped out are classified as Type 3; (4) objects that are in the SUOL but swapped out before being updated are classified as Type 1, and additionally Type 2 after the update.

Depending on the recovery data maintained in the object buffer, DB is classified into dual buffer-based partial rollback using soft log (DB-SL) and dual buffer-based partial rollback using shadows (DB-SO). We explain these two schemes in detail in the following subsections.

4.2.1 *Dual Buffer-Based Partial Rollback Scheme Using Soft Log (DB-SL)*

For a given savepoint, DB-SL recovers the objects in the SUOL using the soft log; for those not in the SUOL, it deletes them from the object buffer if they exist in the object buffer. It restores the objects using the hard log if they have been swapped out to the page buffer. When an object contained in an SUOL is updated (thus becoming Type 3), the update is recorded in the soft log.

At a savepoint: DB-SL performs operations for handling the SUOL and the soft log. More specifically, it (1) records the SUOL of the savepoint, the current transaction execution state, and the SaveLSNs's of hard and soft logs of the transaction (denoted by *HardSaveLSN* and *SoftSaveLSN*) in main memory, and (2) returns a savepoint identifier.

During a partial rollback: DB-SL (1) deletes from the object buffer all objects not in the SUOL of the savepoint, (2) for all objects in the SUOL, undoes the updates in the object buffer using the soft log up to the SoftSaveLSN of the savepoint(objects being undone are swapped in if they are not currently in the object buffer), (3) undoes the updates in the page buffer using the hard log up to the HardSaveLSN of the savepoint (excluding delayed objects), and (4) restores the saved transaction execution state.

Among the steps of partial rollback steps in DB-SL, we see that objects of Type 1 are recovered in steps 3 and 4, those of Type 2 are in steps 1, 3, and 4, and those of Type 3 are in steps 2, 3, and 4.

As in OB, DB-SL incurs the overhead of using the soft log. However, compared with OB, using the hard log in parallel reduces the overhead significantly for two reasons. First, since DB-SL creates soft log records only for the objects in SUOL (i.e., of Type 3[1]), the memory overhead is smaller than that of OB for objects of Type 2 or of both Type 1 and Type 2. Second, since the number of updates undone using the soft log is smaller in DB-SL, the overhead of swapping in the objects that are not in the object buffer is also smaller than that of OB.

4.2.2 *Dual Buffer-Based Partial Rollback Scheme Using Shadows (DB-SO)*

DB-SO uses shadows instead of the soft log to avoid object swapping required in DB-SL. A shadow is a copy of an object made at a point in time. When an object in an SUOL is updated for the first time after the savepoint, the shadow of the object is created in main memory. The savepoint and partial rollback operations are the same as in DB-SL except using shadows instead of the soft log.

At a savepoint: DB-SO (1) records the SUOL of the savepoint, the current transaction execution state, and the HardSaveLSN of the savepoint in main memory, and (2) returns a savepoint identifier.

During a partial rollback: DB-SO (1) deletes from the object buffer the objects not in the SUOL of the savepoint, (2) replaces all objects in the SUOL (or create them if they are not in the object buffer) by their shadows, (3) undoes the updates in the page buffer using the hard log up to the HardSaveLSN of the savepoint, and (4) restores the saved transaction execution state.

[1]Note that the SUOL objects of Type 1 are not updated after the savepoint. Thus, no soft log records are created for them.

DB-SO has three advantages over DB-SL. First, DB-SO does not need object swapping during a partial rollback. It can recover the state of the object buffer simply by copying the shadows directly into the object buffer even for the objects not in the object buffer. Second, since DB-SO generates shadows only at the first update of the objects in SUOLs, whereas DB-SL logs all updates of the objects, it incurs much smaller memory overhead than DB-SL. Third, while the memory overhead of DB-SL varies unpredictably depending on the number of updates on the objects in the SUOL of the savepoint, that of DB-SO is no more than the number of objects in all SUOLs, offering a predictable upper bound.

5. PERFORMANCE EVALUATION OF PARTIAL ROLLBACK SCHEMES

5.1 Model of Experiments

Using simulations, we have evaluated the performance of the proposed schemes. In this section, we describe the performance criteria, system model, and workload model used in our experiment. The models are based on the one used by Franklin et al. [2] and have been modified to fit our purpose.

Performance Criteria We use the following three criteria:

1. *nSwaps*: the number of objects swapped between the object buffer and the page buffer
2. *nDiskIOs*: the number of disk I/Os for swapping pages in the page buffer and accessing the hard log
3. *MemoryOverhead*: the size of the main memory space for storing recovery data such as soft log or shadows

In order to distinguish between different schemes, we associate a performance criterion and a partial rollback scheme by appending the latter to the former as a subscript, e.g., $nSwaps_{\mathrm{PB}}$ for "*nSwaps* of PB."

System model Figure 2 shows the reference system model. It consists of a diskless client workstation and a server machine with a disk, connected over the network. The client comprises an object buffer manager, a data/recovery manager, a transaction source, and a resource manager. The object buffer manager uses the LRU object replacement policy. The data/recovery manager maintains either the soft log or shadow depending on the partial rollback scheme, and performs partial rollback and commit operations in the object buffer. The transaction source initiates transactions according to the workload model explained later. The resource manager provides CPU service and access to the network. The server is modeled similarly to the client, but with the following differences. First, the resource manager manages disks as well as CPU. Second, the page buffer manager manages pages instead of objects. Third, the data/recovery manager maintains a hard log instead of a soft log.

Table 1 shows the system parameters used in the experiments. We set the size of a page to 4 Kbytes, and the size of an object to 200 bytes, and assume the number of objects in a page is 20. Because the soft log record contains undo information only, we set its size identical to the size of an object. In addition, as the hard log record contains both undo and redo information, we set its size to double the size of an object. The size of a log buffer is set to one page.

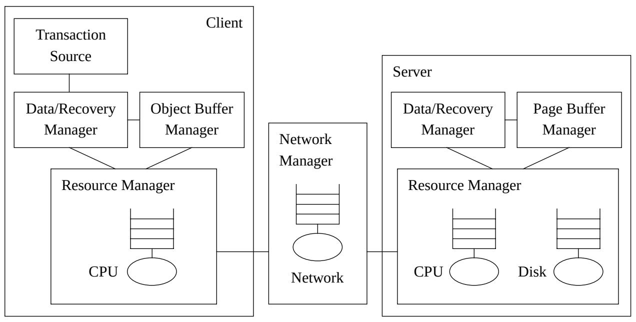


Figure 2: System Model.

The size of database is set to 600 Mbytes, containing about 3 million objects. The page buffer size is set to 128 pages (512 Kbytes)[2].

Table 1: System Parameters.

Size of a page	4096 bytes
Size of an object	200 bytes
Size of a hard log record	Size of an object × 2
Size of a soft log record	Size of an object
Size of a log buffer	One page
Size of database	600 MB (more than 3,000,000 objects)
Size of the page buffer (N_{PB})	128 pages (512 KB)
Size of the object buffer (N_{OB})	320, 640, ... , 5440

Workload model Figure 3 shows the access pattern of a transaction, where m_1 denotes the number of operations from the transaction start to the first savepoint, m_2 from the first savepoint to the partial rollback point, and m_3 from the partial rollback point to the commit point. The number of operations between two consecutive savepoints is evenly set to m_2/N_S. We perform partial rollbacks always to the first savepoint, which facilitates analyzing the overhead of savepoint operations by keeping the number of undone operations constant.

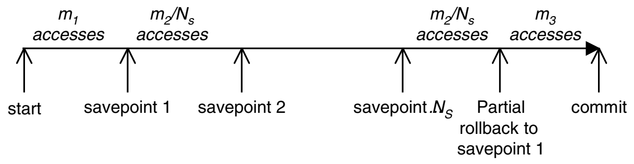


Figure 3: An Access Pattern of a Transaction.

Table 2 shows the workload parameters used to control the system workload. Since the number of transactions in a single client system does not affect the performance in terms of our criteria[3], we assume a single transaction without loss of

[2]We used rather small sizes for the database and the buffer compared with what we would find on a modern computer. We did it for the sake of efficiency of the experiments. However, the experimental results are valid because they are determined by the ralative sizes of the database and buffer, not by the absolute sizes.

[3]In a multiple-client system, *nDiskIOs* will increase. Nevertheless, the trends will be similar.

Table 2: Workload Parameters.

Number of transactions	1
Number of savepoints (N_S)	1, 3, 5, 7, 9
Number of partial rollback	1
Ratio of update operations (R_W)	0.3
Number of object accesses per transaction (M)	10000 ($M = m_1 + m_2 + m_3$)
Number of accessed objects per transaction	3000, 10000
Access sequence of repeatedly accessed objects	Exponential distribution (reference mean: 685, update mean: 457)
Access pattern of a transaction	m_1 = 2000, m_2 = 5000, m_3 = 3000

generality. Savepoint operations can be performed as often as needed, but partial rollback operations are less frequent. Thus, we fix the number of partial rollback operations to be one. Because update operations are less frequent than reference operations in an ordinary transaction, we set the ratio of update operations R_W to 0.3.

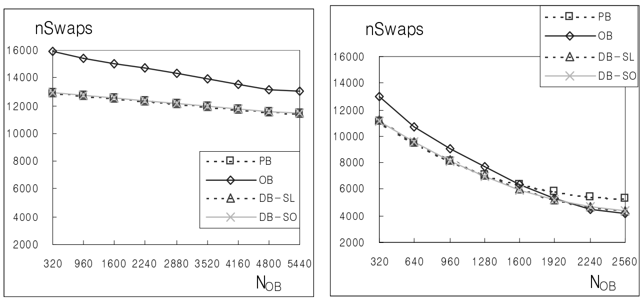
5.2 Experiments and Results

In this section we outline the simulation performed and evaluate the results.

Outline We use the following three dimensions: the *size of the object buffer*, the *frequency of savepoint operations*, and the *hit ratio of the object buffer.* First, we vary the *number of objects* maintained in the object buffer (N_{OB}) from 320 ($< m_2/N_S$) to 5,440 ($> m_2$) by an increment of 320 to the effect of increasing the object buffer size. Second, we vary the *number of savepoints* (N_S) from 1 to 9 by an increment of 2 to the effect of changing the frequency of savepoints. Third, we consider two cases of different object buffer hit ratios: case 1 *without* objects accessed repeatedly (i.e., hit ratio = 0) and case 2 *with* such objects (i.e., hit ratio > 0). For case 1, we set the number of accesses per transaction to 10,000 and the number of accessed objects to 10,000. For case 2, we set the former to 10,000 and the latter to 3,000. In a DBMS, accesses to objects are typically concentrated on a small number of objects[2]. We simulate this effect by using an exponential distribution of accesses to objects. That is, we generate a sequence of 10,000 array indices exponentially distributed in the range of 1 to 3,000. Here, the means are set to 685 for references and 457 for updates. Then, we use the 10,000 numbers as indices to the object array to obtain an object access sequence.

Given the three dimensions, in Experiment 1, we compare the performance of the four schemes for different values of N_{OB}, and in Experiment 2, for the different values of N_S. In Experiment 3, we compare the memory overhead for different values of N_{OB} and N_S. Each experiment is divided into the two cases: with or without objects accessed repeatedly.

Experiment 1 We observe how the performance varies with N_{OB} while fixing N_S to 1. Figure 4 shows the resulting *nSwaps* of the four schemes. As shown in Figure 4(a), without objects accessed repeatedly, *nSwaps* decreases slowly as N_{OB} increases. While PB and DB show almost identical *nSwaps*, OB shows greater *nSwaps* than the other schemes. It is because, while the other schemes just delete the objects in the object buffer during the partial rollback, OB swaps in the objects to be undone (if not in the object buffer), thus forcing other objects to be swapped out, and also swaps out the undone objects after the partial rollback as the transaction continues. Additionally, DB-SL and DB-SO show



(a) Without objects accessed repeatedly. (b) With objects accessed repeatedly.

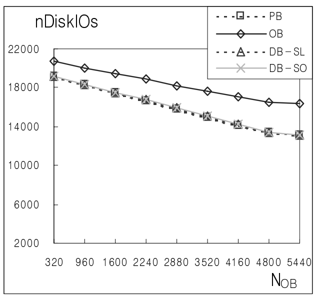
Figure 4: *nSwaps* when the Number of Savepoints N_S = 1.

the same *nSwaps* because no object is undone in the object buffer during a partial rollback.

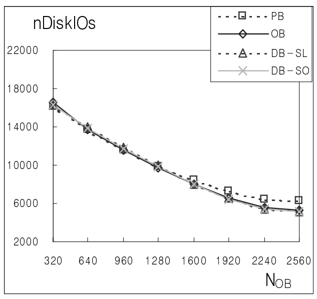
Figure 4(b) shows that DB outperforms the other schemes when there are objects accessed repeatedly. We see that $nSwaps_{\text{PB}}$ exceeds $nSwaps_{\text{DB}}$ when N_{OB} reaches 960, and exceeds $nSwaps_{\text{OB}}$ when N_{OB} reaches 1,600. It is because, while the object buffer hit ratios of OB and DB improves as N_{OB} increases, thus reducing the number of objects swapped, the forced flush in PB will increase the number of objects swapped out. *nSwaps* of DB-SL is slightly higher than that of DB-SO because DB-SL occasionally swaps in the objects to be undone during a partial rollback.

Figure 5 shows that *nDiskIOs* of the four schemes changes similarly to *nSwaps* as N_{OB} changes. In Figure 5(a), *nDiskIOs* of all schemes except OB decreases more rapidly than *nSwaps* as N_{OB} increases. It is because, as the number of swapped objects decreases, so do the number of disk I/Os and the number of objects undone in the page buffer during a partial rollback. *nDiskIOs* of OB does not decrease as much as the others because it does not undo the updates recorded in the hard log during a partial rollback. However, when there exist objects accessed repeatedly as in Figure 5(b), *nDiskIOs* of OB drops as much as those of the other schemes. It is because of the OB's sensitivity to the rate of object swapping during a partial rollback. Obviously, a higher object buffer hit ratio reduces the number of object swapping.

We find from the result of the experiment that the performance of all four schemes improves as the object buffer size increases. We also make other observations. When there exist objects accessed repeatedly, PB shows inferior performance to the others as N_{OB} increases because of the forced-flush overhead. This phenomenon becomes more noticeable

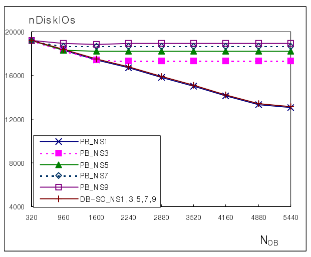


(a) Without objects accessed repeatedly.

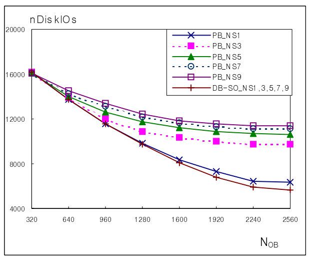


(b) With objects accessed repeatedly.

Figure 5: *nDiskIOs* when the Number of Savepoints $N_S = 1$.



(a) Without objects accessed repeatedly.



(b) With objects accessed repeatedly.

Figure 6: *nDiskIOs*$_{\text{PB}}$ and *nDiskIOs*$_{\text{DB-SO}}$ when the Number of Savepoints N_S = 1, 3, 5, 7, and 9.

as N_S increases, as we see in Experiment 2. When N_{OB} is quite large and there exist objects accessed repeatedly, OB shows performance comparable to DB due to the increase of object buffer hit ratio. DB consistently performs better than SB regardless of N_{OB} and the existence of repeatedly accessed objects.

Experiment 2 We observe the performance as N_S is changed. Due to the forced-flush overhead at savepoints, PB's performance is sensitive to N_S, while the others are not. Since DB-SO offers the best performance among partial rollback schemes, we focus on PB ans DB-SO in our experiment[4]. We shows the change of *nDiskIOs*$_{\text{PB}}$ and *nDiskIOs*$_{\text{DB-SO}}$ with respect to N_S and N_{OB} in Figure 6. The numbers appended to 'PB_NS' and 'DB-SO_NS' denote the number of savepoints. For example, 'PB_NS1' means the "nDiskIOs of PB in the case of one savepoint."

In Figure 6(a) and (b), when $N_S = 1$, the performances of PB and DB-SO are almost identical. However, PB's performance decreases as N_S increase, since updated objects in the object buffer are flushed frequently as savepoints are set frequently. In contrast, DB-SO's performance does not changes as N_S changes. As a result, DB-SO outperforms PB when savepoints are set frequently.

As shown in Figure 6(a), when no object is accessed repeatedly, *nDiskIOs* of PB does not decrease despite the increase of object buffer size if the number of savepoints is more than one. For example, when $N_S = 5$, *nDiskIOs* remains almost constant at 18,000 for N_{OB} greater than 960. We can explain this phenomenon as follows. PB deletes the objects in the object buffer during a partial rollback, which decreases the number of object swaps, as well as the number of disk I/Os for swapping pages and writing hard log records. The number of deleted objects is calculated as $R_W \cdot \min(m_2/N_S, N_{\text{OB}})$, where the symbols are defined in Tables 1 and 2. We observe that, as savepoints are set frequently, updated objects in the object buffer are flushed frequently. As a result, the number of deleted objects are fixed to m_2/N_S, and therefore *nDiskIOs* remains the same even if the object buffer size increases. Furthermore, when there exist objects accessed repeatedly, given more than one savepoint, the number of flushed objects increases even though the object buffer size increases. Thus, as shown in Figure 6(b), it causes more disk I/Os, reducing the rate of decrease of *nDiskIOs*$_{\text{PB}}$. Consequently, for a larger value of N_S, the performance of PB improves less as N_{OB} increases. We omit the result of *nSwaps*$_{\text{PB}}$ here because the trend of *nSwaps* is similar to that of *nDiskIOs*.

[4]We also have performed experiments for the other schemes. The performances of the other schemes are not sensitive to N_S either. Due to lack of space, we omit the details here.

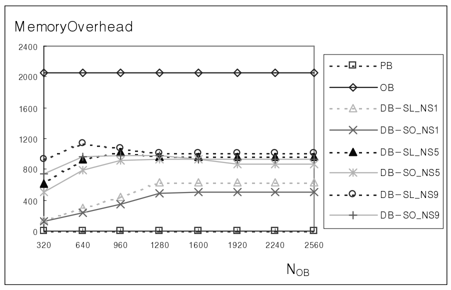


Figure 7: Memory Overhead when there Exist Repeatedly Accessed Objects, and N_S = 1, 5, and 9.

Experiment 3 We observe the memory overhead of the four schemes. When there exists no object accessed repeatedly, OB incurs very large overhead to maintain the soft log while the other schemes do not at all. PB by its nature does not maintain any separate recovery data of the object buffer. For DB, without objects accessed repeatedly, no more update occurs for the objects contained in SUOL, and therefore, no recovery data is created.

When there exist objects accessed repeatedly, the memory overhead of the four schemes changes with N_{OB} and N_S as shown in Figure 7. The horizontal axis shows the object buffer size, and the vertical axis the memory overhead represented as either the number of shadows (in DB-SL) or the number of soft log records (in DB-SO). For both DB-SL and DB-SO, the number of savepoints is appended to 'DB-SL_NS' and 'DB-SO_NS,' respectively.

As expected, the memory overhead of PB is always zero. OB's memory overhead is equal to the number of updates performed before the partial rollback, and is irrelevant to N_S and N_{OB}. In contrast, as shown in Figure 7, the memory overhead of DB-SL and DB-SO increases with N_S, but is still far smaller than that of OB. It is because, although the number of objects in SUOLs increases as N_S increases, the number is still very small compared with the number of all objects updated until the partial rollback. The memory overhead of DB-SL is slightly more than that of DB-SO as expected. It is because DB-SO scheme records the recovery data only for the first update of the object after the savepoint, while DB-SL records the recovery data whenever the object in the SUOLs is updated.

Summary In PB, the number of force-flushed objects increases as savepoints are set more frequently. This overhead causes performance degradation, and also dwarfs the performance gain obtained from increasing the object buffer size.

The performance of OB is not affected by the number of savepoints. It is, however, influenced by the size and the hit ratio of the object buffer. The smaller the object buffer size may be, the more object swaps occur during a partial rollback (in order to undo the updates of objects not in the object buffer), resulting in poorer performance. By contrast, as the object buffer grows in size, the swapping overhead shrinks rapidly. In addition, as more objects are accessed repeatedly, more object buffer hits occur, which result in less frequent object swaps and therefore better performance. Lastly, OB incurs an inherent memory overhead for maintaining the soft log.

The performance of DB is not affected by the number of savepoints. It shows relatively good performance compared with other schemes regardless of the object buffer size and the existence of repeatedly accessed objects. Besides, its memory overhead is negligible compared with that of OB. DB incurs more memory overhead than PB. Nonetheless, DB's performance is regarded better than PB's because it can accommodate more frequent savepoints. Between the two schemes of DB, DB-SO shows better performance than DB-SL.

6. CONCLUSION

We have proposed a thorough study of partial rollback schemes of OO/ORDBMSs with a dual buffer. We have proposed four partial rollback schemes for OO/ORDBMSs, which are page buffer-based, object buffer-based, dual buffer-based using soft log, and dual buffer-based using shadows. Then, we have explained the concepts and procedures of each scheme, and discussed the pros and cons of them individually. In addition, we have described the experiments conducted for performance evaluation of the four schemes and the results.

We summarize the performance of the four proposed schemes as follows. The page buffer-based scheme is the easiest to implement because it requires the minimum extension of the existing scheme of RDBMSs. However, it suffers from performance degradation due to forced-flush overhead if savepoints are set frequently. The object buffer-based scheme shows good performance when the object buffer size is large enough. However, it has to sustain memory overhead to maintain the soft log for recovery. The dual buffer-based schemes are harder to implement than the previous two. However, they always show relatively good performance regardless of the number of savepoints and the size of the object buffer. Their memory overhead is much smaller than that of the object buffer-based scheme. The dual buffer-based scheme using shadows shows better performance than the one using soft logs. These observations suggest that in general the dual buffer-based partial rollback scheme using shadows performs the best.

The partial rollback mechanism is essential to any system supporting interactive users. Thus, we believe the proposed partial rollback schemes and their evaluation results are very helpful to the design and implementation of OO/ORDBMSs using a dual buffer.

7. ACKNOWLEDGMENTS

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

8. REFERENCES

[1] Bernstein, P. A., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[2] Franklin, M. J., Carey, M. J., and Livny, M., "Transactional Client-Server Cache Consistency: Alternatives and Performance," *ACM Trans. on Database Systems*, Vol. 22, No. 3, pp. 315-363, Sept. 1997.

[3] Gray, J. et al., "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, Vol. 13, No. 2, pp. 223-242, June 1981.

[4] Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[5] Kim, W., *Introduction to Object-Oriented Databases*, Computer System Series, The MIT Press, 1st ed., 1990.

[6] Loomis, M. E. S., *Object Databases: The Essentials*, Addison-Wesley, 1995.

[7] Mohan, C. et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollback Using Write Ahead Logging," *ACM Trans. on Database Systems*, Vol. 17, No. 1, pp. 94-162, Mar. 1992.

[8] Subramanian, M., and Krishnamurthy, V., "Performance Challenges in Object Relational DBMSs," *IEEE Data Engineering Bulletin*, Vol. 22, No. 2, pp. 27-31, June 1999.

[9] Oracle Corp., Oracle Call Interface Programmer's Guide Release 8.0, 1997.

[10] UniSQL, C Application Programming Interface Reference Manual, 1996.

[11] Park, C. M., Carey, M. J., and Dessloch, S., "MAJOR: A Java Language Binding for Object-Relational Databases," *Proc. 8th Int'l Conf. Workshop on Persistent Object Systems*, Tiburon, California, Aug. 1998.

[12] Kim, S. H., Jung, M. S., Park, J. H., Park, Y. C., "A Design and Implementation of Savepoints and Partial Rollbacks Considering Transaction Isolation Levels of SQL2," *Proc. Sixth Int'l conf. on Database Systems for Advanced Applications*, pp. 303-312, Apr. 1999.

[13] Fussell, D. S., Kedem, Z. M., Silberschatz, A., "Deadlock Removal Using Partial Rollback in Database Systems," *Proc. Int'l Conference on Management of Data*, pp. 65-73, Apr. 1981.

[14] Özsu, M. T., Dayal, U., and Valduriez, P., *Distributed Object Management*, Morgan Kaufmann, 1994.