In: Proc. of the 7th Int. Conf. on Database and Expert Systems Applications (DEXA'96), Zurich, Switzerland, Sept. 1996. pp. 116-129.

Employing Object-Based LSNs in a Recovery Strategy

Fernando de Ferreira Rezende and Thomas Baier
Department of Computer Science - University of Kaiserslautern
P.O.Box 3049 - 67653 Kaiserslautern - Germany
Phone: +49 (0631) 205 3274 - Fax: +49 (0631) 205 3558
E-Mail: rezende@informatik.uni-kl.de

Abstract. *We present a recovery strategy supporting partial rollbacks of transactions and object-granularity locking, called WALORS (**WAL**-based and **O**bject-oriented **R**ecovery **S**trategy). As the name suggests, WALORS uses the principle of Write-Ahead Logging (WAL). In contrast to other strategies, WALORS stores a Log Sequence Number (LSN) in every object of the database (DB) to correlate the state of the object with logged updates to the object. Due to this feature, WALORS does not need to employ a repeating history (redo all) paradigm. Instead, it supports selective undo as well as selective redo passes. To avoid the problems of supporting fine-granularity locking in the context of WAL, WALORS employs special control structures which enable it not having to write compensation log records (CLRs) from CLRs and guaranteeing its idempotence even in the face of repeated failures or of nested rollbacks. WALORS does operation logging of all updates, including the ones performed during rollbacks, works with fuzzy checkpoints, supports media recovery, and is flexible enough w.r.t. the kinds of buffer management policies being implemented.*

1 Introduction

As times go by, computers are getting always faster and cheaper. According to an analysis made by Gray [10], by the end of this decade we can expect clusters of computers with thousands of processors giving a tera-op processing rate, terabytes of RAM storage, many terabytes of disc storage, and terabits-per-second of communications bandwidth (the era of *4T machines*). Even nowadays, the architectures of DBs are already very powerful, almost matching some of these features (e.g., AT&T Teradata, Tandem, Oracle, IBM's SP2, Informix, etc.). Among others, one of the most innovative aspects of such architectures is the very large memory employed by them, some reaching 50 GB of storage capacity (e.g., AlphaServer systems of Digital). In turn, this enlargement of memory is normally exploited in two ways. On one hand, the block size as well as the page size are significantly enlarged. On the other hand, the common memory space for the DB is also enlarged, so that it is possible to have a huge number of such large blocks in the memory at the same time. As a consequence, both measures increase the hit rate of the DB cache, and decrease then the time-consuming accesses to the secondary storage. Perhaps, one of the greatest challenges presently is to build the associated software to support such super-machines' environments.

Traditionally, recovery strategies following WAL and doing operation logging employ a unique LSN per page. This is usually so, no matter whether a locking granule finer than a page is supported. However, if we think of such large pages (e.g., 32 KB), it turns out that one LSN per page is insufficient for precisely tracking what is really going on inside a page. This is a fact because such large pages may store hundreds of objects. However, an operation affecting any of these objects must be generalized by such recovery strategies as being an operation performed to a page, simply because a page's LSN disregards the page contents. This generalization leads to a lack of information, with hard consequences to the restart processing in such recovery strategies.

In this paper, we present a recovery strategy, called WALORS, focusing on such aspects like very large page sizes. When we started designing WALORS, our main goal was to cope with the above problem, and thus to more precisely track the state of the objects inside a page, rather than the state of the page as a whole. In order to achieve this goal, we have then employed one LSN per object. On this basis, we can be very precise when analyzing the log records, and track a log record to the specific object that it gives respect to, and not to the set of objects inside a page. The main consequences of this design decision will be discussed in the scope of this paper.

This paper is organized as follows. In Sect. 2, we discuss the assignment of LSNs both per page and per object. Thereafter, we present the data structures of WALORS (Sect. 3). Then, we discuss the main tasks of WALORS during normal processing (Sect. 4), and at system restart (Sect. 5). Finally, we conclude the paper in Sect. 6.

2 Page- and Object-Based LSNs

LSNs are a very efficient means for precisely tracking the state of a data item w.r.t. logged actions relating to that data item.[1] They work as follows. Firstly, to each update operation recorded in the log file as a log record, an LSN is assigned to it. LSNs are monotonically increased, the last log record contains the highest LSN. Secondly, each data item carries an LSN, which describes the logical point of time in which the data item was most recently updated. Hence, by means of LSNs, each update operation recorded in the log file carries a kind of time stamp. Furthermore, by affixing this time stamp to each data item every time it is updated, the recovery component can very efficiently decide at restart time whether redo or undo actions are to be taken to the data item. This decision is taken by just comparing LSNs:

- **Redo** is necessary when **data item's LSN < log record's LSN**; and
- **Undo** is necessary when **data item's LSN >= log record's LSN**.

LSNs are also used by the buffer manager to realize the WAL protocol. The two basic rules of the WAL protocol are [8, 9]:

(1) Before overwriting an object to non-volatile storage with uncommitted updates, the log records relative to such updates should firstly be forced to the non-volatile log space. (Undo information is gathered.)

(2) Before committing an update to an object, the log records relative to such an update should be firstly forced to the log on non-volatile storage. (Redo information is gathered.)[2]

Among other aspects, the many recovery strategies nowadays in the market may be differentiated by means of the passes they execute when restarting the system after a failure, and in the way they store and handle the LSNs. ARIES [17], for example, performs an analysis pass followed by a redo all pass (the *repeating history* paradigm), and a selective undo pass. In contrast to that, we designed WALORS aiming at supporting the *selective redo* paradigm, i.e., just the updates of *non-loser transactions* that did not get propagated into the DB before the crash are redone. Fig. 1 compares the actions taken by ARIES and WALORS during restart processing after a system crash.

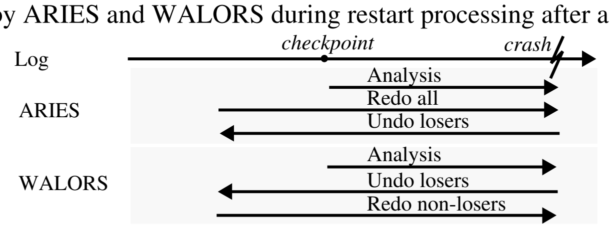

Fig. 1. Restart processing strategies of WALORS and ARIES.

2.1 Page-Based LSNs and Record-Level Locking

Due to these features, it is possible that many transactions access the same page to work on records into it. Since just one LSN per page is available, the use of LSNs to recognize which actions must be taken during restart processing may become

1. In this paper, we use the term *data item* to denote both a page or an object in general.
2. On following these two rules, WAL systems allow that an updated object may be propagated back to the same non-volatile storage location from where it was read (i.e., an *update-in-place* policy is supported). Most of the commercially available systems employ WAL, since the update-in-place policy is considered to be better than the *shadow page* technique [7, 17, 1].

problematic. The problem is that no confident information may be obtained about which specific record in the page was the last time updated, since the page's LSN is always changed independently of the records inside it. Due to this deficit on information, special care must be taken when restarting the system after a crash.[3] Essentially, the undo pass can no longer identify precisely which updates should be undone or not. Of course, unnecessarily undoing an operation may still be harmless in physical logging under certain conditions, but with operation logging severe inconsistencies may appear if an operation is undone more than once.[4] ARIES copes with this problem by means of the *repeating history* paradigm [17], i.e., in the redo pass it redoes all updates not yet propagated into the DB. By doing so, it guarantees that all updates of both losers and non-losers are reflected in the DB after the redo pass has completed, and hence the following undo pass does not get into problems with the LSNs' comparison. (In fact, the undo pass does not even need to compare them.) This paradigm does avoid this problem, but on the other hand a higher processing overhead during restart is the price for it.[5]

2.2 Object-Based LSNs and Object-Level Locking

WALORS works with objects as the finest lock granule. For its effective realization, each object of the DB must be designed with an LSN. Putting an LSN into each object makes possible the realization of selective redo as well as selective undo passes. Due to the locking concept [3, 6, 15], it is impossible that more than one transaction update an object at the same time, and so its LSN is only changed sequentially by the transactions, accordingly to their serialization order imposed by the lock protocol.

Fig. 2 illustrates the two essential cases that may happen when two transactions update the same object. In Fig. 2 a), the update operation (LSN 30) of the loser T2 was propagated into the DB. This operation will be rolled back, no matter whether firstly the redo or the undo pass is performed. If we perform undo firstly, then the update 30 will be rolled back and a corresponding CLR will be written, which will push the object's LSN forwards. This does not have consequences to the following redo pass, since no actions will be taken anyway. In Fig. 2 b), it is necessary to redo the update 20 of the non-loser T1, because the object was propagated at time 10. On firstly redoing this operation, the object's LSN will be set to 20. However, this value is smaller than the LSNs recorded for the update operations performed by the loser T2, and thus no undo action will be taken, what is correct because both operations (30 and 40) were never propagated into the DB. In summary, it does not matter in which specific order the undo and redo passes are realized, the LSN comparison will not be compromised in any cases.

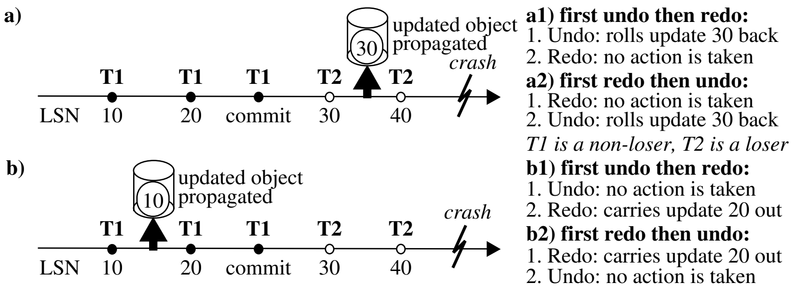

Fig. 2. Selective redo in the context of WAL and object-based LSNs.

3. The reader is referred to [17] for a detailed discussion about this problem.
4. Operation logging allows to do logical logging and supports semantically rich lock modes (e.g., based on the commutativity of operations [5, 23, 24, 4, 20, 2, 13]).
5. Mohan and Pirahesh tried to optimize the redo pass in ARIES-RRH (*Restricted Repeating of History*) [19]. They did obtain a selective redo pass for ARIES, but at the costs of a coarser lock granule: Instead of a record, a page is the finest lock unit for update operations.

2.3 Considering the Trade-Off between Page-Based and Object-Based LSNs

Essentially, the main differences between these two approaches are:

- *Page-based LSNs and fine-granularity locking (e.g., record-level)*: Each page carries an LSN; at system restart, a number of logged actions must be redone for just being undone later.
- *Object-based LSNs and fine-granularity locking (e.g., object-level)*: Each object carries an LSN; no logged actions are unnecessarily redone or undone at restart time.

Hence, storing an LSN per object usually requires more memory than per page.[6] On the other hand, an LSN per object leads to better processing times at restart processing than per page, since logged actions are neither redone nor undone unnecessarily. In order to draw a trade-off between these approaches, we need the average size of objects and page, to the end of knowing how many objects fit into a page, and hence how much more memory is needed for storing the LSNs in the objects. In addition, we need to know how many operations are in average logged during normal processing in a time interval, how many of them are unnecessarily redone and later undone at system restart, and how long such an operation takes, in order to know how much processing time is saved. Finally, aspects like performance of the recovery algorithms, the efficiency of the different strategies for the LSNs' comparisons, the employed data structures, etc. must be also taken into account. We are currently studying all these aspects for drawing a trade-off between both approaches. Further on, we are also considering the influences of very large buffers on the buffer management policies (*force/no-force, steal/no-steal*), on checkpointing schemes, on lock granules, and the complexity of these aspects for recovery. Notwithstanding, we believe that with the page sizes getting larger and larger, the number of logged actions to a page gets accordingly increased, so that it makes necessary to be more precise when logging them. At last, we further believe that nowadays a better response time is more desired than economies of memory resources.

3 Data Structures of WALORS

3.1 Log Record

The log file is composed of sequentially ordered log records (Table 1).

Table 1. Log record.

LSN	The LSN uniquely identifies a log record.[a] It increases monotonically.
Type	Identifier of the log record's type. There are many types: *begin*, *update*, *CLR*, *prepare*, *savepoint*, *commit*, and *abort*.
TRID	Unique identifier of the transaction which wrote the log record.
Obj_ID	Object upon which the update was performed (used in *update* and CLR log records).
Last_LSN	LSN of the last log record written by the same transaction.
RedoNext_LSN	This pointer has two different main intentions. On one hand, it is used to make a very efficient selective redo possible. For this purpose, it is constructed dynamically during restart processing, and points to the next log record to be worked out by the redo pass (see Sect. 5.2.1). On the other hand, it is present in CLRs and is used for guaranteeing the idempotence of the undo pass. In this context, it is updated during normal processing, and points to the log record that this CLR is compensating for (see Sect. 5.2.2).
UndoNext_LSN	It is present only in CLRs, and points to the next log record of the transaction that must be coped with during the rollback process.[b]
Length	It contains the length of the complete log record.
Data	It contains the undo and/or redo information describing the performed operation.

a. LSNs need not be explicitly noted in the log record in centralized DBs, they can actually be the address of the log record in the log file. However, in client/server DBs, they do need to be explicitly stored.
b. We need two separate backward chains in order to correctly process both transaction rollbacks (*UndoNext_LSN* field) as well as system restarts (*Last_LSN* field) (see Sect. 4.3 and Sect. 5.2).

6. Supposing an object is smaller than a page, of course. For ease of exposition, it is assumed in this paper that an object fits in a single page.

3.2 Transaction Table

The transaction table (Table 2) is used for the transaction management. It is necessary for controlling the total and partial rollbacks of transactions.

Table 2. Transaction table (*Trans_Tab*).

TRID	Unique identifier of the transaction.
State	It identifies the current state of a transaction. It can be *active* or *in-doubt*. The status *committed* or *aborted* of transactions do not need to be recorded in the table, since, as soon as a transaction terminates, its corresponding entry is removed from the table.
UndoNext_LSN	It is the address of the next log record of this transaction to be rolled back. It is used during an abort or a partial rollback of the transaction.
Last_LSN	LSN of the last log record written by the transaction. It is used for back chaining the log records of a transaction.

3.3 Dirty Object Table

This table (Table 3) manages the so-called dirty objects, i.e., objects that have been updated in the volatile storage, and which updates were not propagated to the non-volatile version of the same objects. Every time an object is flushed from the buffer and propagated into the DB, its corresponding entry is removed from this table.

Table 3. Dirty object table (*Dirty_Object*).

Obj_ID	It contains the unique identifier of an object.
Dirty_LSN	This LSN describes the logical point of time at which the object was firstly updated after being checked into the volatile storage.

4 Normal Processing

4.1 Update Operations

On performing an update operation and consequently generating a log record, the object's LSN receives the log record's LSN. The log records of a transaction are chained together by backward pointers. This backward chain is realized by means of the *Last_LSN* field of the log records, and the values are assigned accordingly to the *Last_LSN* field of the transaction table. The *Last_LSN* field of a log record points exactly to the previous log record written by this same transaction (Fig. 3). Of course, the first log record (*begin*) of a transaction has a null pointer in this field.

4.2 Compensation Log Records

CLRs have been around for a long time and they are implemented in most WAL systems. The main reason of their existence is to enable the recovery manager to precisely track the process of undoing operations by logging the actions performed during rollbacks. Additionally, they are fundamental for media recovery and for guaranteeing the idempotence of the whole recovery process (see Sect. 5.2.2). A CLR is written for every update operation recorded in the log which needs to be undone for any reason. On undoing an operation, the object's LSN receives the LSN of the corresponding CLR that was written. By means of CLRs, the recovery manager can know, namely through the LSNs' comparison, whether the respective undo operations were already propagated into the DB or not.

The construction of CLRs is similar to the one of normal log records. They contain only redo information, simply because there is no need to undo them. As discussed (Table 1), a CLR contains information in both *UndoNext_LSN* and *RedoNext_LSN* fields. The *UndoNext_LSN* field receives the *Last_LSN* pointer of the log record being undone. In turn, the *RedoNext_LSN* field receives the LSN of the log record this CLR is compensating for (refer to Sect. 5.2.2). Fig. 3 shows the chains of log records and CLRs that are built by rolling back the updates 40 and 30 of transaction T1.

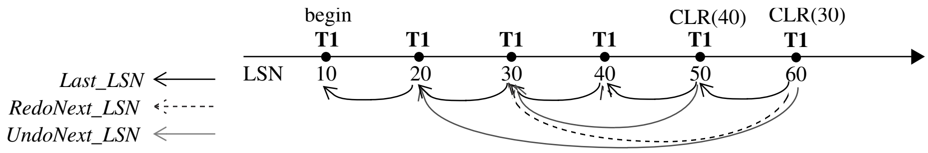

Fig. 3. Chains of log records after undoing updates 40 and 30 of T1.

4.3 Total Rollback of Transactions

When a transaction aborts, all its update operations must be rolled back. During the rollback process, the log records are brought into the log buffer (if they are not yet there), and the update operations of the transaction are rolled back in reverse chronological order. Fig. 4 illustrates the rollback of T2 (for the sake of simplicity, the *RedoNext_LSN* pointers are not shown). Firstly, the update 60 was undone and a CLR was written (LSN 80). The *Last_LSN* field of this CLR integrates it into the normal backward chain of log records of a transaction, it points thus to the update 60. In turn, its *UndoNext_LSN* field is made to point to the previous update operation, namely 50. This is the next to be undone. The total rollback of a transaction is finished when the *UndoNext_LSN* field of the last written CLR points to the *begin* log record of this transaction. At this time, an *abort* log record is written (LSN 110).

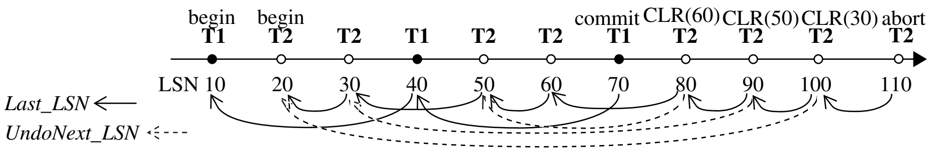

Fig. 4. Total rollback of transaction T2.

4.4 Partial Rollbacks of Transactions

WALORS also supports partial rollbacks of transactions.[7] Transactions can establish *savepoints* at any point of time during their existence. Thereafter, they can request the rollback of all updates subsequent to the establishment of a still outstanding savepoint. This concept has as main purpose to limit the extent of transaction rollbacks. In principle, the procedure for partial rollbacks is almost identical to total rollbacks. The only difference is that the whole rollback process is not interrupted when the corresponding *begin* log record is found, but previously, when the desired *savepoint* log record is reached. The backward chain is realized in the same way as for total rollbacks. Whether the savepoint after the partial rollback process is still outstanding or not does not matter for the recovery process, it is more an implementation decision. After being partially rolled back, the transaction can make forward progress again and even commit.

During the rollback process of a transaction, after processing a non-CLR, the next log record is determined by means of its *Last_LSN*. Nevertheless, CLRs may be encountered when interpreting the log records in the reverse order as they were written. When a CLR is found, the next log record to be examined during the rollback process is determined by looking up to its *UndoNext_LSN*. Hence, already undone log records are skipped over through this pointer. However, this pointer may be followed only by the rollback processes of transactions during normal processing, because during restart processing after a system crash only the *Last_LSN* pointer can be used (Sect. 5.2.2).

4.5 Transaction Commits and Aborts

We assume that some *two-phase commit* (2PC) protocol [8] is in effect to handle

7. Partial rollbacks are crucial for user-friendly handling integrity constraint violations [7], application program failures and deadlocks [15], problems arising from using obsolete cached information [14], etc.

transaction commits. Hence, in the first phase of 2PC, the log records are written in the log file (as part of WAL). Whether or not the objects modified by the transaction are forced to the DB at this time depends on the buffer replacement policy being employed (*force*/*no-force*); it does not matter for WALORS. Thereafter, a list of the *update* locks acquired by the transaction is written in the log file. Finally, the *prepare* log record is synchronously written in the log file, and the first phase is completed (at this time, the transaction's *read* locks may be released). The transaction is said to be now in the *in-doubt* state. In the second phase of 2PC, a *commit* log record is written in the log file and all locks of the transaction are released. The transaction is thus *committed*.

The abort of a transaction is handled by rolling all its operations back (total rollback, Sect. 4.3), releasing all its locks, and finally synchronously writing an *abort* log record in the log file. The transaction is hence *aborted*.

4.6 Checkpoints

Checkpoints are useful to reduce the time of the system restart process. They are taken during normal processing and accordingly to different strategies (refer to [12]). We have chosen for WALORS the so-called *fuzzy checkpoints*. The main feature of fuzzy checkpoints is that modified objects need not be forced to disk at checkpoint time (*hot-spots* require special handling),[8] on the contrary, only status information is written in the log file. Thus, the costs for checkpointing are very low.

Fuzzy checkpoints are performed in two well-defined phases. At the beginning, a special log record is written in the log, *begin_checkpoint*, whose LSN is remembered. Thereafter, another special log record is built, *end_checkpoint*. Inside this record, WALORS includes a list of the modified objects currently in the buffer (*Dirty_Object* table) and a list of the *active* (and maybe *in-doubt*) transactions (*Trans_Tab* table). As soon as this log record is built, it is synchronously written in the log. Finally, the remembered *begin_checkpoint*'s LSN is written in a pre-defined, well-known place in stable storage. This is the starting point for system restart. At last, *begin_checkpoint* as well as *end_checkpoint* need not be forced to the log file at checkpoint time. Their migration to non-volatile storage may be normally governed by WAL. The only requirement is that the well-known place in stable storage which stores the *begin_checkpoint*'s LSN may only be updated when the *end_checkpoint* reaches the log file in stable storage.

5 System Restart

The system restart process brings the DB to a transaction consistent state after a system crash. In order to achieve this, WALORS uses the log records stored in the log file and works on them by means of different passes (refer to Fig. 1).

5.1 The Analysis Pass

This pass produces the necessary information for the next two passes, namely, a list of *active*/*in-doubt* transactions (*Trans_Tab*), a list of modified objects that probably were in the buffer in the moment of the crash (*Dirty_Object*), and the address of the last successfully written log record (*Last_Log*). It uses as input the LSN of the last complete checkpoint. At this time, the *Trans_Tab* and the *Dirty_Object* tables are initialized with the information contained in the checkpoint. From that on, the log records are read forwards until the end of the log file, and interpreted according to their types:[9]

- *Begin* - On finding a *begin* log record, a new entry is created in the *Trans_Tab* table. The transaction's state is set to *active*, and the *Last_LSN* field is set to the LSN of this log record.

8. *Hot-spots* are frequently accessed objects that may stay in the buffer for long periods of time. We assume that such *hot-spots* are handled by the buffer manager by means of some strategy [11, 18]. The main point is that they must be reasonably often propagated into stable storage to reduce the restart work.
9. *Savepoint* log records are not of interest for system restart. Hence, we disregard them from now on.

- *Update* or CLR - The *Last_LSN* field of the corresponding transaction in the *Trans_Tab* table is updated to this log record's LSN. If there is no entry in the *Dirty_Object* table for the object onto which this update was performed, a new entry for it is created. In this case, this object's *Dirty_LSN* is set to this log record's LSN, signalling when it was modified for the first time.
- *Prepare* - In the *Trans_Tab* table, the transaction's state is set to *in-doubt*, and the *Last_LSN* pointer of the corresponding transaction is updated to this log record's LSN.
- *Commit* or *Abort* - The corresponding entry for the transaction in the *Trans_Tab* table is deleted.

5.2 The Undo Pass

The undo pass has two main tasks. On one hand, it is responsible for rolling the loser transactions back, on the other hand, it prepares the subsequent redo pass. To the end of undoing the operations of losers in the exact reverse chronological order, the log file is read backwards, starting from the *Last_Log* address established by the previous analysis pass. In addition, the knowledge about the losers is taken from the *Trans_Tab* table produced by this same previous pass. On individually processing the log records, it must be first of all ascertained whether it pertains to a loser or to a non-loser. The following actions depend on this. Whereas the propagated updates of losers are rolled back, the non-propagated updates of non-losers are prepared for the subsequent redo pass, in that a *forward chain* is dynamically built.

Fig. 5 shows a simplified example of this pass. It begins by processing the last log record of the log file (Fig. 5 a). This log record, LSN 110, pertains to the loser T2 and must therefore be rolled back, if it were already propagated into the DB (for the sake of simplicity, we give up the LSN comparison in this example and assume that all updates of the loser T2 must be rolled back and the ones of the non-loser T1 redone). The next log record, LSN 100, pertains to the non-loser T1, but it does not need to be considered because it contains no redo information. Contrarily, the next log record, LSN 90, does contain redo information (an *update*) and is therefore taken into account for the forward chain. At this time, the LSN of this log record is remembered as a variable, the *RedoNext_LSN variable*. Following, the log record 80 of T2 is undone and the 70 of T1 is reached. Thus, the *RedoNext_LSN* field of this log record receives the remembered *RedoNext_LSN variable*. After that, this variable is updated with the LSN of this new log record. By means of these proceedings, the losers are being undone (Fig. 5 b), and the updates of non-losers chained together (Fig. 5 c). Finally, after terminating the undo pass, an *anchor address* is passed on to the redo pass which efficiently, directly accesses the log records of non-losers to the end of redoing them.

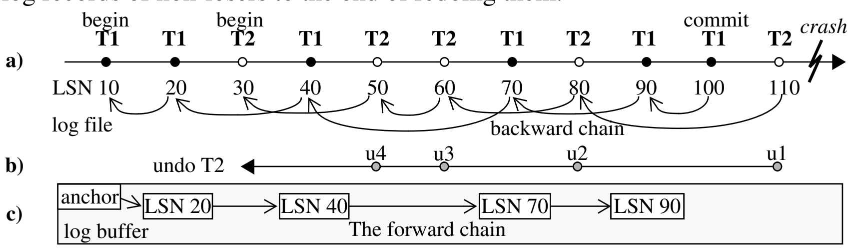

Fig. 5. A simplified undo pass after a system crash.

5.2.1 The Forward Chain

The undo pass prepares at run time a forward chain in the log buffer for the redo pass (Fig. 5 c). The log records of non-losers which were not yet propagated into the DB are fetched up into the log buffer and forward chained. Contrarily to the backward chains, which are transaction-specific and built at normal processing time, this forward chain spreads to log records of different non-losers and is built during restart processing.

The forward chain's main purpose is to speed up the redo pass. As will become clear in the next section, the undo pass is appropriate for the construction of this chain because all log records must be investigated during restart processing anyway, i.e., no log record may be skipped over by using the *UndoNext_LSN* pointer of CLRs, as in the (partial or total) rollback process. Since all log records are analyzed, the log records' LSNs of non-losers are also checked against the LSNs of the corresponding objects. If the update operation were not yet propagated into the DB, its log record must be considered for redo purposes, and therefore it is appended in the forward chain. To realize such a chain, the *RedoNext_LSN* field of the log record is used (see Table 1, this is the first purpose of this field, the second is discussed in the next section). Only log records containing redo information are of interest here, i.e., *update* and CLR log records. Notice that CLRs may only appear as log records of a non-loser, when this transaction was partially rolled back before reaching its commit point.[10]

5.2.2 Guaranteeing Idempotence in the Undo Pass

Idempotence is a substantial property of a recovery strategy. It guarantees that the same, consistent DB state is reached after a system restart, even in the face of repeated failures during restart processing. An operation is known as being idempotent if doing it several times is equivalent to doing it once. As discussed previously, logical logging is usually not idempotent, its idempotence is met by employing LSNs in the log records. Another important means for achieving idempotence is the use of CLRs, they supply fundamental information which is used during system restart. On the basis of CLRs and their LSNs, the undo process can determine whether the corresponding undo operations were already propagated into the DB or not. In WALORS, guaranteeing idempotence when handling losers in the undo pass is not trivial. On investigating their log records, the following cases are to be differentiated:

1. The described update was not yet propagated into the DB.
2. The update was propagated into the DB, there is still no corresponding CLR though.
3. The update was propagated into the DB, there is already a CLR. The CLR's undo operation was not yet propagated into the DB, i.e., the update was in fact not yet rolled back.
4. The update and its corresponding CLR were propagated into the DB, i.e., the update was actually rolled back.

Fig. 6 illustrates examples of these different cases. In this scenario, T1 was in process of being (partially or totally) rolled back, when the system has then crashed. The update 30 on object O1 was not propagated into the DB before the system crash (case 1). The update 40 on O2 was propagated into the DB, but there is no CLR for it, i.e., the previous rollback process, before the crash, had taken no action w.r.t. this update (case 2). The updates 60 and 70 on objects O4 and O5, respectively, were considered by the previous rollback process, there are CLRs for them thus, but the undo actions described by their CLRs were not propagated into the DB, and hence the effect is as if the operations were in fact not really undone (case 3). Finally, the update 50 on O3 was taken into account by the previous rollback process, and both it and its CLR were propagated into the DB, and therefore the operation was really already undone (case 4).

The processing of these different cases is done as follows. In the 1st case, no undo measure needs to be taken, since the update was not propagated into the DB. In the 2nd case, the update must be rolled back and a corresponding CLR written. If a CLR already exists, which was not yet propagated though (3rd case), then the update is rolled back, and the affected object receives the LSN of the already written CLR. If the update and

10. Since this forward chain is realized in the log buffer, appropriate measures must be taken in order to not extrapolating the log buffer's capacity. In general, this can be achieved by correctly adjusting the checkpointing interval time. In addition to that, in our implementation of WALORS we have taken another precaution: On approaching the limits of the buffer capacity, we discard enough log records in the tail of the chain and retain only their LSNs. As soon as the redo pass is initiated, we start an asynchronous process to check into the buffer the discarded log records by means of their retained addresses (LSNs).

its CLR were already propagated (4th case), then no more action is necessary. This is the correct way to handle the log records of losers during undo. However, there may appear some problems if we use the simplified proceedings described so far for processing the undo pass. In the following, we illustrate and discuss some problematic situations and present our complete undo pass and some necessary data structures for well coping with these problems.

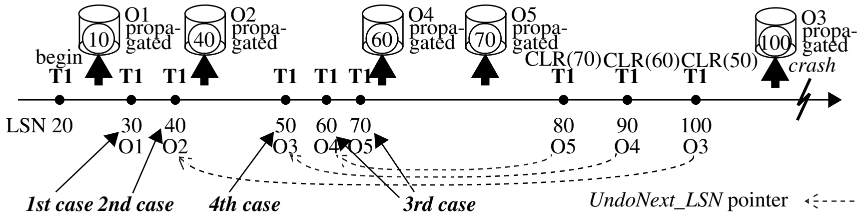

Fig. 6. Different cases when handling log records of loser transactions in the undo pass.

Problem by Following the CLRs' UndoNext_LSN Pointers

As shown in Fig. 6, not all undo operations of a loser, which was involved in a rollback process before the system crash, may have come into effect in the DB. This scenario makes very clear the difference between rolling a transaction back during normal processing and rolling it back during system restart. Contrarily to normal processing, the CLR's *UndoNext_LSN* pointer shall not be followed during restart processing. In Fig. 6, one can notice that although the CLR(50) was already propagated into the DB on O3, its *UndoNext_LSN* pointer may not be used because both updates 60 and 70 on O4 and O5, respectively, were not yet rolled back, i.e., albeit there are CLRs for them, the CLRs' undo actions have not affected the DB before the crash. Consequently, the undo pass must investigate each log record of a transaction. In the example (Fig. 6), the next log record to be worked out by the undo pass is the CLR(60). On analyzing it, one notices that the corresponding undo operation (undo from 60) was not yet propagated (log record's LSN(90) is greater than object's LSN(60)). Nevertheless, one may not at all immediately carry this operation out, since the results would be a wrong order in the realization of the undo pass (in this case, the update 60 would be rolled back before the update 70). As said, the update operations must be always rolled back in the exact reverse order of their original execution.[11]

The solution to this problem shall fulfil the following conditions:

(1) The undo operations must be rolled back in the right (reverse) order; and additionally
(2) There shall not be generated CLRs for CLRs.

In order to reach the above goals, we have designed WALORS to manage two transaction-specific LIFO (*last in first out*) chains during system restart: A *Non_Undo* chain; and a *CLR_Exists* chain. These chains are dynamically created for a transaction during the undo pass as soon as the first CLR for it is found in the process of reading the log file backwards. On finding a CLR, both chains are initialized for the corresponding transaction (if they do not exist yet), and an entry is created in one of both as follows:

- If the undo operation described by the CLR were already propagated, then an entry in the *Non_Undo* chain is created containing the LSN of the corresponding update operation this CLR has compensated for.[12]
- If the CLR's undo operation were not yet propagated, then an entry in the *CLR_Exists* chain is created, which contains two LSNs: The LSN of this CLR, and the one of the update operation this CLR compensates for (i.e., CLR's LSN and *RedoNext_LSN* fields).

When reading an *update* log record, it is decided by means of both chains, whether the *update* log record must be rolled back or not, and whether a CLR is to be written or

11. In general, that is true at least for non-commutative update operations performed on the same object.
12. This LSN is the value stored in the *RedoNext_LSN* field of this CLR during normal processing (Sect. 4.2).

not. At this time, there are three cases to be considered:

1. There is an entry in the *Non_Undo* chain for this *update* log record, then no more undo action is taken, since it was already rolled back.
2. There is an entry in the *CLR_Exists* chain for this update operation, then the LSN comparison is executed to decide whether an undo operation is necessary or not. However, the modified object does not receive the LSN of this *update* log record, instead the one of the already written CLR. Thus, no CLR is generated anymore.
3. There is neither an entry in the *Non_Undo* chain nor in the *CLR_Exists* chain for the update operation, then the LSN comparison takes place, an undo operation is executed if the update operation were already propagated and must therefore be rolled back, and a CLR is generated.

Let us apply the above proceedings to the problematic scenario illustrated in Fig. 6. Table 4 shows the steps taken during the undo pass and the employment of both chains.

Table 4. Employing the *Non_Undo* and *CLR_Exists* chains.

Step	LSN	Log Record	*Non_Undo* Chain	*CLR_Exists* Chain	Actions
1st	100	CLR(50)	->nil	->nil	- initialize both chains - entry in: *Non_Undo*
2nd	90	CLR(60)	(LSN 50)->nil	->nil	- entry in: *CLR_Exists*
3rd	80	CLR(70)	(LSN 50)->nil	(LSN 60, 90)->nil	- entry in: *CLR_Exists*
4th	70	*update*	(LSN 50)->nil	(LSN 70, 80)->(LSN 60, 90)->nil	- undo from 70 - new object LSN: 80 - entry out: *CLR_Exists*
5th	60	*update*	(LSN 50)->nil	(LSN 60, 90)->nil	- undo from 60 - new object LSN: 90 - entry out: *CLR_Exists*
6th	50	*update*	(LSN 50)->nil	->nil	- no undo necessary - entry out: *Non_Undo*
7th	40	*update*	->nil	->nil	- undo from 40 - write CLR(40): LSN 110 - new object LSN: 110
8th	30	*update*	->nil	->nil	- no undo necessary - write CLR(30): LSN 120 - new object LSN: 120
9th	20	*begin*	->nil	->nil	- end of T1's undo - release both chains

At the beginning (1st step), CLR(50) is read, what causes both chains to be initialized, and its LSN (100) is compared to the object's LSN (100). In this case, this undo operation was already propagated, and hence an entry in the *Non_Undo* chain is created. Following (2nd step), CLR(60) is read, and the LSN comparison executed. This undo operation was not yet propagated (90 < 60), but there is already a CLR for it, thus an entry in the *CLR_Exists* chain is created. Thereafter (3rd step), the same proceedings are taken for the CLR(70), causing the generation of another entry in the *CLR_Exists* chain. Now, an *update* (LSN 70) is read (4th step). Both chains are then checked, and it is found out that an entry for this update exists in the *CLR_Exists* chain. Hence, this update is rolled back, the object receives the LSN of the already written CLR (i.e., LSN 80), this entry is deleted from the *CLR_Exists* chain, and no CLR needs to be generated. Thenceforth (5th step), similar actions are taken for the next *update* (LSN 60). On reading the next log record, an *update* (LSN 50, 6th step), it is found out that there is an entry in the *Non_Undo* chain for this *update*, what means that it was already rolled back, and therefore no undo action is necessary anymore, just this entry must be removed from the *Non_Undo* chain. Following, the next log record (LSN 40) is again an *update* (7th step). Since both chains are now empty, this operation is handled as usual, i.e., the LSNs' comparison takes place, and, if necessary, the operation is undone, a CLR for it is generated, and the object's LSN receives the new CLR's LSN. Further, by means of the LSN comparison for the next log record (an *update*, LSN 30) it is found out that this operation has in fact never taken place (8th step).[13] Finally, the *begin* for T1 is found

13. Notice that CLRs must be generated even for the non-propagated update operations.

(LSN 20, 9th step), what means that the undo process of this transaction has successfully terminated, and both (empty) chains are then released.

Thus, by means of the *Non_Undo* and *CLR_Exists* chains, the undo operations are performed in the exact reverse order, and further, no CLRs are generated for CLRs. At last, it should be noticed that such chains are created during the undo pass only for the losers which were already in process of being rolled back before the system has crashed.

5.2.3 Handling Objects' Inserts and Deletes

Probably the most cumbersome aspect of employing an LSN per object instead of per page gives respect to the handling by the recovery strategy of both operations: *Delete an object*, and *insert an object*. When analyzing a log record, how can one surely know whether these operations were propagated into the DB? The problem stays in the LSN comparison. For example, after deletion, the object no longer possesses an LSN, in fact the object itself does not exist anymore, and thus the comparison between object's LSN and log record's LSN is hard to be performed. In WALORS, we have solved this problem through the introduction of two special *update* log records, on the basis of which an extended LSN comparison is performed: *Update_Delete_Obj*, and *Update_Insert_Obj*. These are written during normal processing like usual *update* log records. However, they are differently handled during the undo pass at system restart. In particular, a special measure must be taken when the following error message appears: *'The object does not exist.'* After requesting the LSN of an object for the LSN comparison of any of these special *update* log records, and receiving this error message back, then it is as follows decided whether the corresponding update operation were propagated into the DB or not:

- *Update_Delete_Obj*: The delete operation was propagated.
- *Update_Insert_Obj*: The insert operation was not propagated.

Hence, depending on the specific situation, appropriate undo/redo measures can be taken. For example, on receiving this error message (*'The object does not exist.'*) when interpreting an *Update_Delete_Obj* log record during the undo of a loser, it means that this update operation was in fact propagated into the DB, and therefore the appropriate undo action (i.e., insert the object) must be taken.

As previously discussed, for each undo action taken during the undo pass, a corresponding CLR must be always generated. However, due to the same reasons as for the special update operations, we must employ special CLRs in the undo pass for both insert and delete operations, in order to avoid problems with the LSN comparison. On undoing an *Update_Delete_Obj* log record, a special CLR, named *CLR_Delete_Obj*, is generated. On the other hand, a special CLR, *CLR_Insert_Obj*, is always generated when undoing an *Update_Insert_Obj* log record. As before, both special CLRs are differently handled during the undo pass. When the LSN comparison of any of both cannot be performed, because the corresponding object does not exist, it is decided as follows whether such a CLR were propagated or not:

- *CLR_Delete_Obj*: The undo operation for the *Update_Delete_Obj* was not propagated.
- *CLR_Insert_Obj*: The undo operation for the *Update_Insert_Obj* was propagated.

Finally, all these special *update* as well as CLR log records are handled and generated during normal processing as usual *update* and CLR log records. The above described special measures are taken only when the LSN comparison fails and the error message *'The object does not exist.'* is received.

5.2.4 Undo Pass' Boundary

The log file is processed during the undo pass at least until the last successfully written checkpoint is reached. On one hand, all losers must be rolled back. On the other hand, all log records relevant for the redo of non-losers must be investigated. The undo process is concluded when no more entries for losers exist in the *Trans_Tab* table. In order to establish the log record, until which the undo pass must investigate for preparing the redo pass, the *Dirty_Object* table is used. This table contains for each object a *Dirty_LSN* field, which describes the logical point of time where the object was updated for the first time since it was checked into the buffer. The smallest *Dirty_LSN* field of this table gives the LSN of the log record until which the undo pass must investigate. In summary, the undo pass' boundary can be determined as:

- *min_LSN* := MIN (*min_losers_LSN*, *min_Dirty_LSN*)
  where: *min_losers_LSN* is the smallest LSN of the losers; and *min_Dirty_LSN* is the smallest LSN among the *Dirty_LSN* fields of the *Dirty_Object* table.

Finally, during the undo pass all log records are processed until the LSN *min_LSN* is reached. On reaching it, the undo pass is concluded and the redo pass is started.

5.3 The Redo Pass

The redo pass is the simplest of them, and is processed based on the results of both previous passes. The goal is to redo the non-propagated updates of non-losers. As explained, most work of this pass is already performed during the undo pass. At this point, all relevant log records were already fetched up into the log buffer, and were forward chained by means of pointers in the right redo order.[14] Hence, the redo pass simply receives the first pointer of this forward chain (illustrated as *anchor* in Fig. 5), and processes all its log records. In particular, the redo pass can be very efficiently performed, since accesses to the log file are avoided in principle. A new system crash after the redo pass has started implies that the whole system restart process must be re-initialized. Nevertheless, this may be optimized in that special checkpoints may be taken after each of the passes has been concluded. For example, taking a special checkpoint after the undo pass containing all elements (LSNs) of the forward chain would make a new system restart more efficient. On successfully finishing the redo pass, the DB is again in a transaction consistent state. At this time, it makes sense to take a checkpoint.

6 Conclusions

In this paper, we have presented the properties and the functionality of WALORS. WALORS supports the total and partial rollbacks of transactions, system as well as media failures, and object-granularity locking. As the name suggests, WALORS employs WAL, allowing thus an update-in-place policy to be implemented. With respect to logging, WALORS does operation logging at the object level, a fundamental feature to support semantically rich lock modes. Differently from usual recovery strategies employing WAL, WALORS uses an LSN in each object of the DB. This feature enables it to perform selective undo as well as selective redo passes at system restart.

During normal processing, the update operations of transactions are recorded as log records in the log file. Additionally, at this time WALORS supports the total rollbacks of transactions. Further on, WALORS allows the establishment of savepoints at any time during transaction execution, and performs rollbacks to outstanding savepoints as required. When undoing the operations of transactions, WALORS always generates CLRs, on the basis of which its idempotence is guaranteed. Finally, WALORS takes fuzzy checkpoints, due to their inherent feature of very low checkpointing costs, and is

14. One can make the processing of non-losers' log records during the undo pass even more efficient, especially for CLRs. When a CLR of a non-loser is found, it means that a partial rollback has taken place, and hence some previous update operations of this transaction were undone during normal processing. In the way we described the undo pass, both the *update* and its corresponding CLR will be appended to the forward chain for the redo pass. However, redoing both produces the same effect as doing nothing. In our implementation of WALORS, we have used the *CLR_Exists* chain and similar procedures for its management in order to avoid such situations of redoing an update operation of a non-loser for just undoing it later by means of processing its CLR. Due to space limitations, we did not enter in the details of this refinement.

flexible enough w.r.t. the kinds of buffer management policies being implemented.

System restart is handled in WALORS by means of an analysis, an undo, and a redo pass. The analysis pass ascertains the uncommitted transactions and the objects that were in the buffer at system crash. The undo pass rolls the propagated updates of losers back, and builds a forward chain for the following redo pass. At this time, special chains are considered in order to roll the operations back in the exact reverse order, and to avoid CLRs for CLRs. In addition, CLRs are generated for any undo operation, also for the non-propagated update operations, so that the LSN comparison may never fail even in the face of repeated failures. On finishing the undo pass, the redo pass is started, which redoes all non-propagated updates of non-losers by means of the forward chain. In particular, this chain allows for a very efficient redo pass, since all relevant log records are checked into the buffer and chained together in the right redo order. Finally, insert and delete operations are handled by WALORS by means of special *update* and CLR log records, guaranteeing the LSN comparison even when an object is no longer to be found in the DB. At last, we have not discussed media recovery here due to space limitations. However, media recovery is a simple task given all the features of WALORS.

In particular, we have extended WALORS, coming up with two major versions of it, to work in client/server environments, and to support nested transactions. In addition, we have implemented this extended WALORS with all its functionality using as a running example KRISYS [16]. In this context, WALORS supports the nested transaction model we have presented at DEXA'95 [22], and the enhanced lock modes of [21]. Due to space limitations, we have not considered these extensions here.

References

[1] Bernstein, P.A., Hadzilacos, V., Goodman, N.: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, USA, 1987.
[2] Chrysanthis, P.K., Raghuram, S., Ramamritham, K.: Extracting Concurrency from Objects: A Methodology. In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Denver, USA, May 1991.
[3] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database Systems. *Communications of the ACM* **19**(11), Nov. 1976. pp. 624-633.
[4] Farrag, A.A., Ozsu, M.T.: Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM TODS* **14**(4), Dec. 1989.
[5] Garcia-Molina, H.: Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM TODS* **8** (2), Jun. 1983.
[6] Gray, J., Lorie, R., Putzolu, F., Traiger, I.L.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. In: *Proc. IFIP Conf. on Modelling in DBMS*, Freudenstadt, Germany, Jan. 1976.
[7] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, Price, T., Putzolu, F., Traiger, I.: The Recovery Manager of the System R Database Manager. *ACM Computing Surveys* **13**(2), June 1981.
[8] Gray, J.: Notes on Database Operating Systems. In: *Operating Systems: An Advanced Course*, Springer, Berlin, Germany, 1978.
[9] Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, USA, 1993.
[10] Gray, J.: Super-Servers: Commodity Computer Clusters Pose a Software Challenge. In: *Proc. GI-Conf. on Database Systems for Office, Engineering and Science Environments*, Dresden, Germany, Mar. 1995.
[11] Härder, T.: Handling Hot-Spot Data in DB-Sharing Systems. *Information Systems* **13**(2), 1988.
[12] Härder, T., Reuter, A.: Principles of Transaction-Oriented DB Recovery. *ACM CS* **15**(4), 1983.
[13] Hadzilacos, T., Hadzilacos, V.: Transaction Synchronization in Object Bases. *Journal of Computer & Systems Sciences* **43**(1), 1991.
[14] Lindsay, B.G., Haas, L.M., Mohan, C., Wilms, P.F., Yost, R.A.: Computation and Communication in R*: A Distributed Database Manager. *ACM TOCS* **2** (1), Feb. 1984. pp. 24-38.
[15] Lockemann, P. C., Schmidt, J. W. (Eds.): *Databases Handbook* (in German). Springer, Germany, 1987.
[16] Mattos, N.M.: *An Approach to Knowledge Base Management*. LNCS 513, Springer, Germany, 1991.
[17] Mohan, C. et al.: *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. IBM ARC Research Report RJ6649, San Jose, USA, 1989.
[18] Mohan, C., Narang, I., Silen, S.: Solutions to Hot-Spot Problems in a Shared Disks Transaction Environment. In: *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, Sept. 1991.
[19] Mohan, C., Pirahesh, H.: ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method. In: *Proc. 7th Int. Conf. on Data Engineering*, Kobe, Japan, April 1991.
[20] Rakow, T.C., Gu, J., Neuhold, E.J.: Serializability in Object-Oriented Database Systems. In: *Proc. 6th Int. Conf. on Data Engineering*, Los Angeles, USA, Feb. 1990. pp. 112-120.
[21] Rezende, F.F., Härder, T.: A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In: *Proc. 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA, Nov. 1994.
[22] Rezende, F.F., Härder, T.: Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs. In: *Proc. 6th DEXA*, London, UK, Sept. 1995. pp. 604-613.
[23] Schwarz, P.M., Spector, A.Z.: Synchronizing Shared Abstract Types. *ACM TOCS* **2**(3), Aug. 1984.
[24] Weihl, W.: Commutativity-Based Concurrency Control for ADT. *IEEE TOC* **37**(12), 1988.