Recovery on the Basis of Objects

Fernando de Ferreira Rezende

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 working on the basis of objects, and 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 recovery 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 also media recovery, and is flexible enough with respect to the kinds of cache management policies being implemented.*

Keywords: *Transactions, database, recovery, fault tolerance, reliability.*

1. Introduction

As times go by, computers are getting always faster and cheaper. According to an analysis made by Gray [9], 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 the unit of transfer between secondary and primary memories, as well as the page size are significantly enlarged (e.g., from 2 to 32 KB). On the other hand, the common memory space for the DB (i.e., the DB cache) 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 at the present time is to build the associated software to support such super-machines' environments.

Traditionally, recovery strategies following the WAL protocol and doing operation logging employ a unique LSN per page in order to track the page's state with respect to logged actions to the page. This is usually so, no matter whether a locking granule finer than a page is supported. Notwithstanding, if we think of such large pages (e.g., 32 KB), it turns out that an 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, one operation affecting any of these (hundreds of) objects must be generalized by such recovery strategies as being an operation performed to a page, simply because a page's LSN disregards the contents of the page. 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 an LSN per object. Hence, each object carries its own LSN. 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. Before discussing the main consequences of this design decision, we introduce some more general features of WALORS:

- WALORS was designed to cope with *transaction*, *system*, and *media failures*. Additionally, WALORS also supports the concept of savepoints and rollbacks to savepoints, i.e., *partial rollbacks*, which is known to be crucial for handling, in a user-friendly fashion, integrity constraint violations [11], application program failures and deadlocks [17], problems arising from using obsolete cached information [16], etc.
- On behalf of a better performance during normal processing, since failures should be considered as exceptions, we have decided to design WALORS to do *operation logging at the object level.* Operation logging allows to do logical logging and supports semantically rich lock modes (e.g., based on the commutativity of operations [6, 26, 1, 28, 5, 22, 3, 13]). (Notwithstanding, we have strived for maintaining WALORS as simple as possible and hence avoiding complex, and thus error-prone, algorithms.)
- As the acronym itself betrays, we use the *write-ahead log (WAL) protocol* [8, 12] in WALORS.
- WALORS supports fine-granularity locking, i.e., object-level locking.
- WALORS does selective redo (i.e., it avoids repeating the history to gain a unique DB representation after a crash) as well as selective undo passes. Additionally, WALORS dynamically builds at restart time a *forward chain*, in which all log records to be redone take part, and by means of which a very efficient redo processing is possible.
- When designing WALORS, we aimed at being flexible enough and not making any kind of restrictive assumptions about the cache management policies being employed. In the scope of this paper, we assume that undo as well as redo actions are necessary during system restart, i.e., that the challenging *steal/no-force* policy, in the terminology of [15], is in effect.

This paper is organized as follows. In Sect. 2, we discuss the assignment of LSNs both per page and per object, and point out their main consequences to a recovery strategy supporting fine-granularity locking in the context of WAL. Thereafter, we present the data structures used in WALORS (Sect. 3). Then, we discuss the main tasks of WALORS during normal processing (Sect. 4), and at system restart (Sect. 5). Finally, in Sect. 6 we briefly discuss media recovery, and in Sect. 7 we conclude the paper.

2. Page- and Object-Based LSNs

The objective of this subsection is to discuss page-based LSNs and the problems that they introduce in supporting fine-granularity locking in the context of WAL. Further on, we try to motivate the use of object-based LSNs due to potential performance gains that can be achieved by their means.

Log Sequence Numbers (LSNs) are a very efficient means for precisely tracking the state of an entity with respect to logged actions relating to that entity. (We use the term *entity* to denote both a page or an object in general.) 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 entity carries an LSN, which describes the logical point of time in which the entity 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 entity 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 entity. This decision is taken by just comparing LSNs:

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

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

(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.)

Hence, before the cache manager flushes an entity from the cache into the non-volatile DB, it ensures that all log entries up to and including the one whose LSN equals the entity's LSN have been appended to the log file. 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 [11, 27, 19, 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 *(Algorithm for Recovery and Isolation Exploiting Semantics)* [19], 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, and in the next subsections we discuss why ARIES must repeat the history and how WALORS achieves a selective redo.

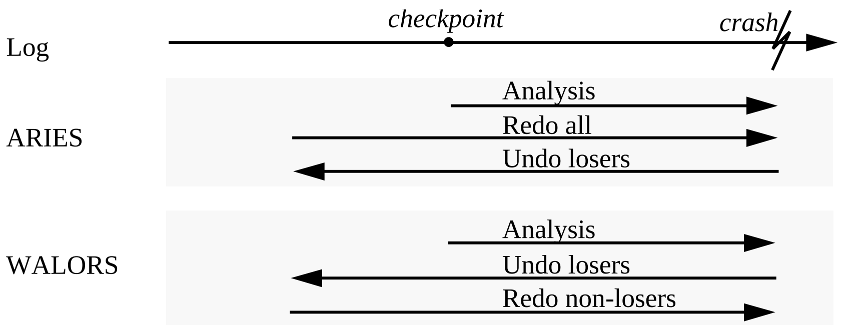

Figure 1: Restart processing strategies of WALORS and ARIES.

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. (Of course, only the accesses to different records in the page need to be considered, i.e., update accesses which are the ones of interest for the recovery component.) Since just one LSN per page is available, the use of LSNs to recognize which actions must be taken during restart processing may become problematic. We illustrate the problem by means of Figs. 2 and 3. In Fig. 2, we have two transactions, T1 and T2. Whereas T1 accesses record 1 (R1), T2 accesses R2 in the same page. As can be seen, each update operation either of T1 or T2 causes the LSN of the page to be accordingly changed. The problem here 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 failure.

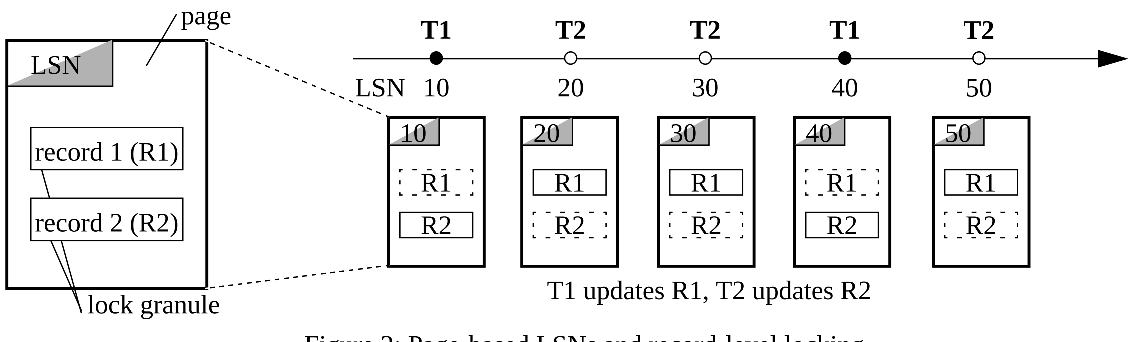

Figure 2: Page-based LSNs and record-level locking.

Fig. 3 shows the problem that can happen in such a situation during restart processing. (In particular, this illustration mimics the one used by Mohan et al. [19] in their discussions about this problem.) The two transactions work on the same page. T1 reached its commit point before the crash time (it is therefore a *non-loser*) and T2 was still active (thus, a *loser*). According to the transactions' properties of atomicity and durability, the effects in the DB of the loser transactions must be undone, whereas the ones of non-loser transactions must survive. Fig. 3a) describes a problem-free scenario. At restart time, the update 30 (recognizable by LSN 30) of T1 will be carried out in the redo pass, and the update 20 of T2 will be rolled back in the following undo pass. These actions are achieved by comparing the page's LSN and the log record's LSN as described before, and they correspond in this situation to exactly what is expected to be done. However, if we change this scenario a little bit, so that the page is propagated into the DB at time 10 (i.e., LSN 10), we get a problem with the LSN comparison in the undo pass (Fig. 3b). Firstly, in the redo pass the update 30 of T1 will be carried out. As a result of this operation, we have that the page's LSN is pushed forward, and worst, beyond the value established by the loser T2. Hence, the following undo pass will try to roll back the update 20 of T2, although it was never propagated and thus should not at all be undone. In summary, the undo pass can no longer identify precisely which updates should be undone or not. As noticed by Mohan et al. [19], the suggestion of Bernstein et al. [2] to reverse the order of the redo and undo passes in order to solve this problem does not work. If this is done, the problem appears in the first situation, i.e., Fig. 3a) represents a problem scenario whereas Fig. 3b) a problem-free one. 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.

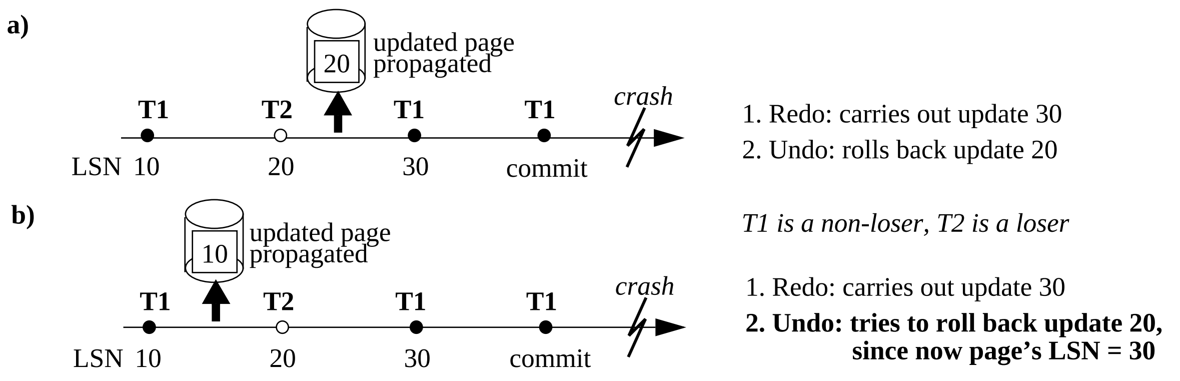
a)
20
updated page propagated
T1
T2
T1
T1
crash
LSN 10
20
30
commit
1. Redo: carries out update 30
2. Undo: rolls back update 20
b)
10
updated page propagated
T1
T2
T1
T1
crash
LSN 10
20
30
commit
T1 is a non-loser, T2 is a loser
1. Redo: carries out update 30
2. Undo: tries to roll back update 20, since now page's LSN = 30

Figure 3: Problem-free (a) and problem (b) scenarios with selective redo in the context of WAL and page-based LSNs.

ARIES copes with this problem by means of the *repeating history* paradigm [19], 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 difficulties with the LSNs' comparison. In fact, the undo pass does not even need to compare the LSNs. Using Fig. 3b) to exemplify, in the redo pass ARIES carries out not only update 30 of the non-loser, but also the update 20 of the loser. Therefore, the following undo pass rolls back the update 20 of the loser, what is right in this case, since it has been redone previously. This paradigm does avoid the above mentioned problem, but, on the other hand, a higher processing overhead during restart is the price for it. Mohan and Pirahesh tried to optimize the redo pass in ARIES-RRH (*Restricted Repeating of History*) [21]. 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.

2.2 Object-Based LSNs and Object-Level Locking

WALORS works with objects as the finest lock granule, which are seen as logical units. For the effective realization of WALORS, each object of the DB must be designed with an LSN. This may be done by means of simply adding an encapsulated attribute to the objects (the *inheritance mechanism* of object-oriented systems can be used to facilitate this task). Putting an LSN into each object makes possible the realization of selective redo as well as selective undo passes. Due to the locking concept [4, 10, 17], 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. 4). Therefore, the above discussed problems cannot happen in WALORS.

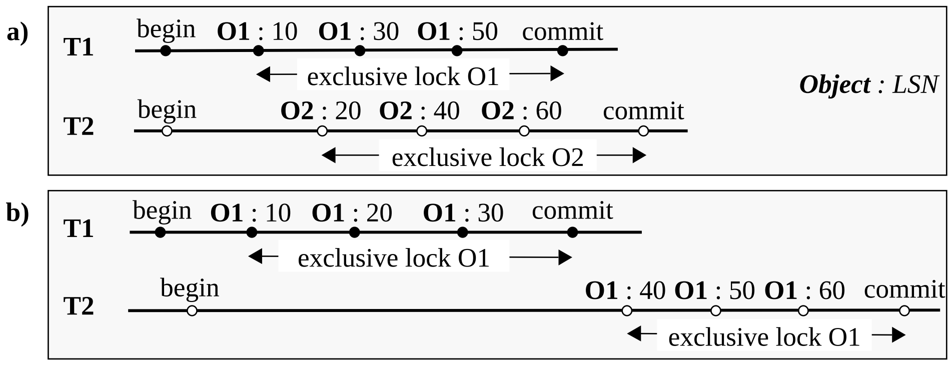

Figure 4: Object-level locking and LSNs' assignment.

Fig. 5 illustrates the two essential cases that may happen when two transactions update the same object. In Fig. 5a), 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 *Compensation Log Record* (CLR) will be written, which will push the object's LSN forward. This does not have consequences to the following redo pass, since no actions will be taken anyway. (CLRs build a fundamental concept in supporting operation logging, since by means of them one can precisely track the state of an entity with respect to, firstly, how the entity has been affected during the rollback process, and secondly, how much of the process has been effectively accomplished thus far (see Sect. 4.2).) In Fig. 5b), 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.

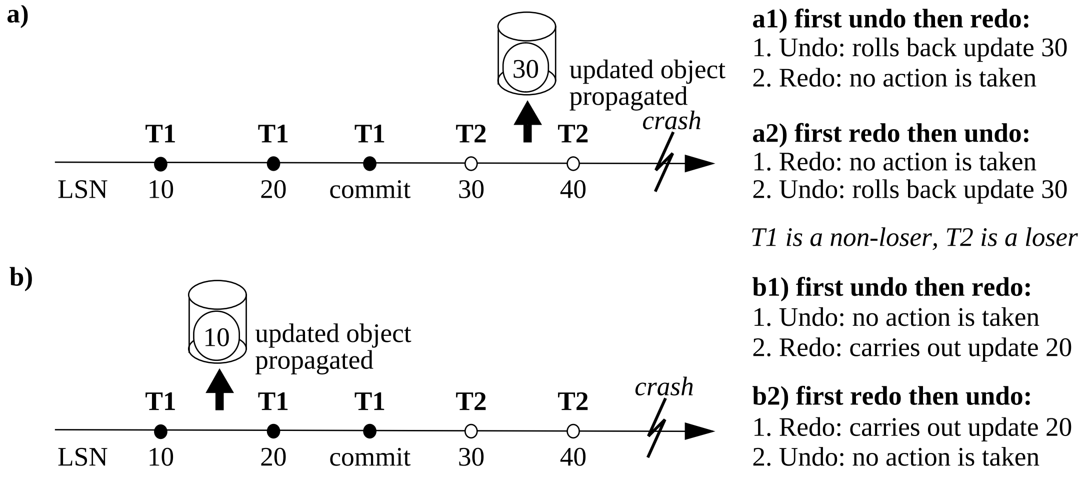
a)
30
updated object propagated
T1
T1
T1
T2
T2
crash
LSN
10
20
commit
30
40
a1) first undo then redo:
1. Undo: rolls back update 30
2. Redo: no action is taken
a2) first redo then undo:
1. Redo: no action is taken
2. Undo: rolls back update 30
T1 is a non-loser, T2 is a loser
b)
10
updated object propagated
T1
T1
T1
T2
T2
crash
LSN
10
20
commit
30
40
b1) first undo then redo:
1. Undo: no action is taken
2. Redo: carries out update 20
b2) first redo then undo:
1. Redo: carries out update 20
2. Undo: no action is taken

Figure 5: Selective redo in the context of WAL and object-based LSNs.

2.3 Considering the Trade-Off between Page-Based LSNs 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 unnecessarily, 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 (primary and secondary) memory than per page. (Supposing an object is smaller than a page, of course. For ease of exposition, it is assumed in this paper that an object fits into a single page.) 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. Additionally, 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 in both approaches, 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 in our studies the influences of very large caches on the cache 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 the operations. 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. Fig. 6 presents the structure of the log records, and in the following we discuss each one of its fields.

LSN	Type	TRID	Obj_ID	Last_LSN	RedoNext_LSN	UndoNext_LSN	Length	Data

Figure 6: Log record.

- LSN (*Log Sequence Number*)

  The LSN uniquely identifies a log record. As said, it increases monotonically. In fact, 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 in the log records.

- Type

  Identifier of the log record's type. There are many types of log records, which are listed in the following:

  - *Begin*

    It describes the begin of a transaction, and therefore contains neither undo nor redo information.

  - *Update*

    It is used to record the update operations. It contains undo and redo information, thus. (We must make a difference between the kinds of update operations, and provide special *update* log records for insert and delete update operations (see Sect. 5.2.3).)

  - CLR (*Compensation Log Record*)

    The CLR contains information about rolled back update operations. It contains just redo information (see Sect. 4.2). (As for update operations, we employ special CLRs when rolling back insert and delete operations (see Sect. 5.2.3).)

  - *Prepare*

    On writing a *prepare* log record, the first phase of the *2-Phase Commit Protocol* is concluded (see Sect. 4.5). It contains information about the locks granted to the committing transaction.

- *Savepoint*

  It establishes a savepoint for the transaction.

- *Commit*

  It signals the successful termination of the transaction.

- *Abort*

  It indicates that the transaction could not be concluded successfully.

- TRID (*TRansaction IDentifier*)

  Unique identifier of the transaction which wrote the log record.

- Obj_ID (*Object IDentifier*)

  Identifier of the object upon which the update operation was performed. It is used in *update* and CLR log records.

- Last_LSN

  LSN of the last log record written by the same transaction. In case of *begin* log records, it contains a null pointer.

- 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. (As can be noticed, we need two separate backward chains in order to correctly process both transaction (partial) rollbacks (UndoNext_LSN field) as well as system restarts (Last_LSN field) (see Sects. 4.3 and 5.2).)

- Length

  It contains the length of the complete log record.

- Data

  It contains the undo and/or redo information describing the performed operation.

3.2 Transaction Table

The transaction table (Fig. 7) is used for the management of the transactions. It is necessary for controlling the total and partial rollbacks of transactions. During normal processing, it is continuously updated. In addition, it is written to non-volatile storage (log file) as part of the checkpoint operations, and is considered during system restart in order to establish the losers and non-losers transactions.

TRID	State	UndoNext_LSN	Last_LSN
⋮	⋮	⋮	⋮

Figure 7: Transaction table (Trans_Tab).

- TRID (*TRansaction IDentifier*)

  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 (Fig. 8) 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. There is an entry in the table for each dirty object in the volatile storage. Every time an object is flushed from the cache and propagated into the DB, its corresponding entry is removed from this table. This table is written to non-volatile storage as part of the checkpoint operations, and is mainly used during system restart for establishing the recovery boundaries.

Obj_ID	Dirty_LSN
⋮	⋮

Figure 8: Dirty object table (Dirty_Object).

- Obj_ID (*Object IDentifier*)

  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 fetched into the volatile storage.

3.4 Object Structure

As said, the recovery manager requires that each object of the DB must possess an LSN field (Fig. 9). This LSN is the address in the log file of the last update operation performed in this object. Every time an object is updated, the log record's LSN must be recorded in the corresponding object.

Obj_ID	LSN
Object data	

Figure 9: Object structure.

4. Normal Processing

In this section, we describe WALORS during normal processing. At this time, its main activities are the collection of log information and the treatment of transaction failures.

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. 10). Of course, the first log record of a transaction has a null pointer in this field.

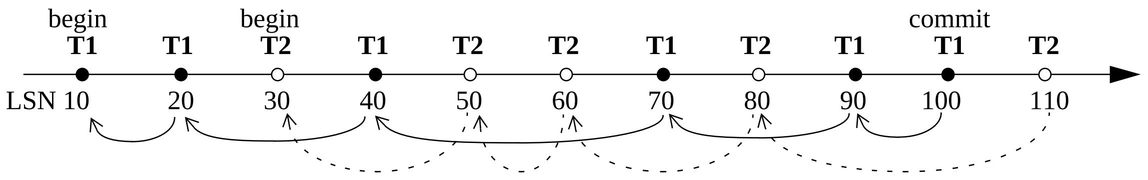

Figure 10: Backward chain of log records.

4.2 Compensation Log Records

Before explaining the treatment of transaction failures, we discuss a bit about the CLRs. 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. In particular, this guarantees that the LSNs of the objects always increase, even when operations are undone during a rollback/undo process. Thus, 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, and thus undo information is irrelevant. As already shortly discussed (see Fig. 6 and Sect. 3.1), a CLR contains information both in the UndoNext_LSN and in the RedoNext_LSN fields of the log record. The UndoNext_LSN field receives the Last_LSN pointer of the log record being undone, i.e., it is a pointer to the previous update operation performed by the same transaction. On the other hand, the RedoNext_LSN field receives the LSN of the log record this CLR is compensating for (the reason for that will become clear in Sect. 5.2.2). Fig. 11a) shows the chains of log records and CLRs that are built by rolling back the updates 40 and 30 of transaction T1. In turn, Fig. 11b) illustrates the pointers (fields of the log records) by means of which such chains are obtained. In the next subsections, the exact employment of CLRs will become clear.

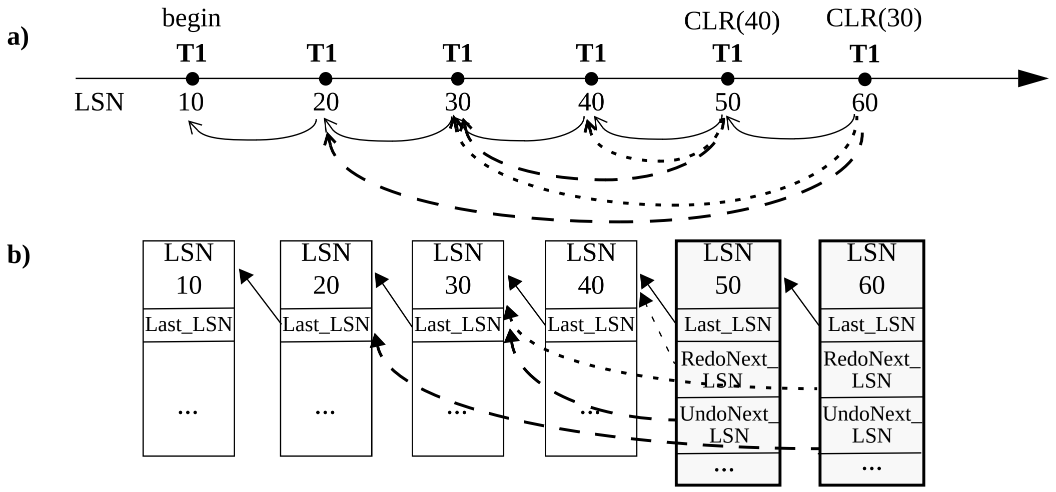

Figure 11: Chains of log records after undoing updates 40 and 30 of T1.

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. 12 illustrates step by step the rollback of a transaction, T2 (for the sake of simplicity, we do not show the RedoNext_LSN pointers in this illustration). In Fig. 12a), the update 60 was undone and a corresponding CLR was written (with LSN 80, which is the one to be newly recorded in the object's LSN). 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 (Fig. 12b). 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 in order to signal that the transaction could not be completed successfully (Fig. 12c).

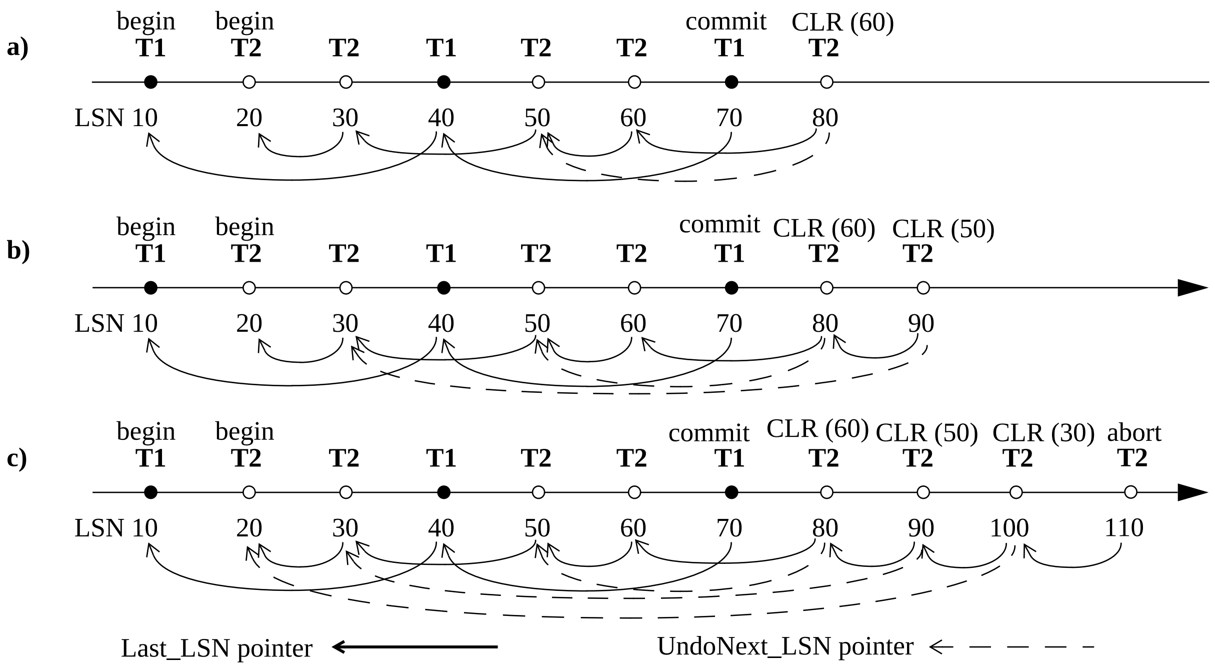

Figure 12: Total rollback of transaction T2.

4.4 Partial Rollbacks of Transactions

Complementing the total rollbacks, WALORS also supports partial rollbacks of transactions. 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 may be used by the transaction, by the system, or even by the user, and 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. Fig. 13 illustrates the partial rollback of a transaction, T1 (as before, the RedoNext_LSN pointers are not shown here). Fig. 13a) shows the log records for T1 with a savepoint. T1 was rolled back until this savepoint, and hence the updates 60 and 50 were undone and corresponding CLRs were written (Fig. 13b). The backward chain is realized in the same way as for total rollbacks, i.e., CLRs are naturally integrated into the backward chain of the log records of a transaction by means of the Last_LSN field, and their UndoNext_LSN fields receive the next operation to be undone. Whether the savepoint after the partial rollback process is still outstanding or not does not matter for the whole recovery process, it is more an implementation decision. After being partially rolled back, the transaction can make forward progress again and even commit (Fig. 13c).

During the rollback process of a transaction, after processing a non-CLR, the next log record is determined by means of its Last_LSN field. 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 field. Hence, already undone log records are skipped over through this pointer. However, as we will see in Sect. 5.2.2, 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.

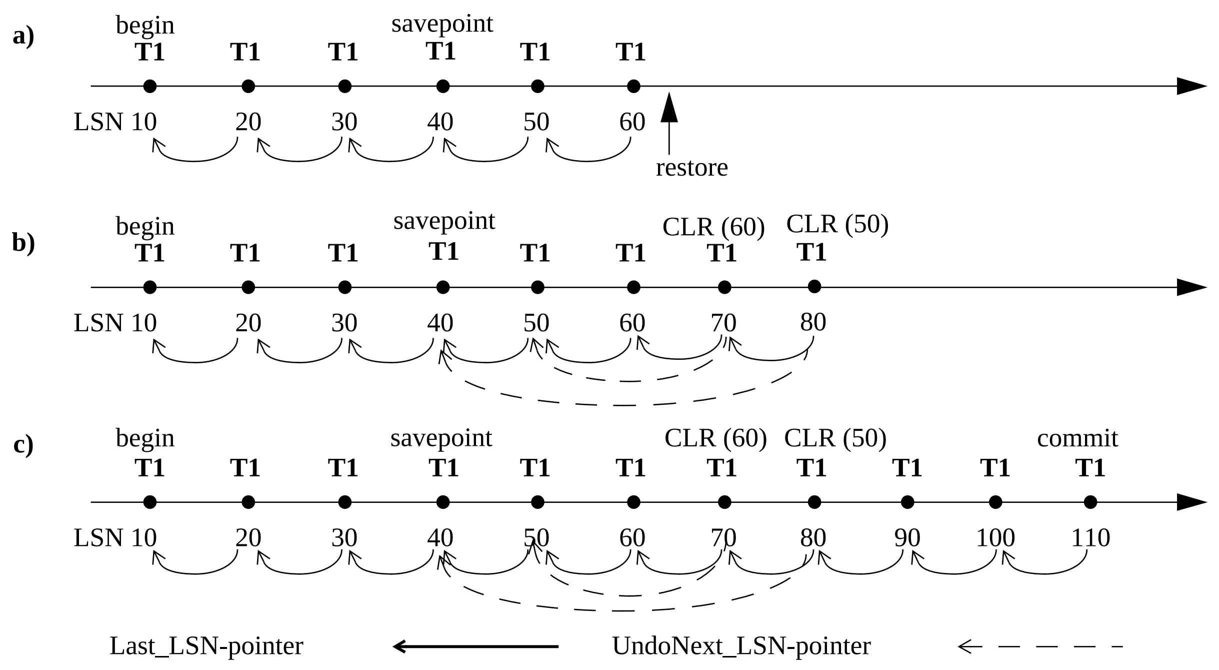

Figure 13: Partial rollback of transaction T1.

4.5 Transaction Commits and Aborts

As required by the atomicity property of transactions, a transaction must either be completely executed (commit) or none of its effects may be reflected in the DB (abort). We assume that some *two-phase commit* (2PC) protocol [8] is in effect to handle transaction commits. Hence, in the first phase of 2PC, the log records are written in the log file (as part of the WAL protocol). Whether or not the objects modified by the transaction are forced to the DB at this time depends on the cache 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 back all its operations (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. (Note that the transactions that may be aborted are the ones in the *active* or *in-doubt* states.)

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 [15]). We have chosen for WALORS the so-called *fuzzy checkpoints*. The main feature of fuzzy checkpoints is that modified objects need not to be forced to disk at checkpoint time, on the contrary, only status information is written in the log file. Thus, the costs for checkpointing are very low. However, on employing fuzzy checkpoints, *hot-spots* require special handling. *Hot-spots* are frequently accessed objects, so that they may stay in the cache for long periods of time. We assume that such *hot-spots* are handled by the cache manager by means of some particular strategy [7, 14, 20]. The main point is that they must be reasonably often propagated into stable storage in order to reduce the restart work.

Fuzzy checkpoints are performed in two well-defined phases. At the beginning, a special log record is written in the log, the *begin_checkpoint* log record, whose LSN is remembered. Thereafter, another special log record is built, the *end_checkpoint* log record. Inside this record, WALORS includes a list of the modified objects currently in the cache (the Dirty_Object table) and a list of the *active* (and maybe *in-doubt*) transactions (the Trans_Tab table). As soon as this log record is built, it is synchronously written in the log. Finally, the remembered *begin_checkpoint* log record's

LSN is written in a pre-defined, well-known place in stable storage. This is the starting point for system restart. At last, the *begin_checkpoint* as well as the *end_checkpoint* log records need not be forced to the log file at checkpoint time. Their migration to non-volatile storage may be normally governed by the WAL protocol. The only requirement is that the well-known place in stable storage which stores the *begin_checkpoint* log record's LSN may only be updated when the *end_checkpoint* log record has reached the log file in stable storage.

5. System Restart

After a system crash, the DB usually is in an unanticipated state. The system restart process brings the DB to a transaction consistent state, guaranteeing the atomicity and durability properties of transactions. In order to achieve this, WALORS uses the log records stored in the log file and works on them by means of different passes, namely an *analysis*, an *undo*, and a *redo pass* (refer to Fig. 1). The goal is to ensure that the updates of loser transactions are undone (guaranteeing atomicity) and that the ones of non-losers are redone (providing durability).

The first pass of this process, the analysis pass, uses as input the *begin_checkpoint* log record's LSN written in a well-known (to it) place in stable storage as part of the last checkpoint operation (see Sect. 4.6). Starting with this address, the log records are interpreted forward and the Dirty_Object as well as the Trans_Tab tables (included in the corresponding *end_checkpoint* log record) are appropriately brought up-to-date. On finishing the log file's analysis, all loser and non-loser transactions are determined. It begins then the undo pass, which uses both tables established during the previous pass for undoing the propagated update operations of loser transactions in the reverse order. The actions taken during this pass are recorded in the log file by means of corresponding CLRs. Lastly, the redo pass is started, which redoes all not yet propagated updates of non-loser transactions. As a result, the DB finally is again in a transaction consistent state. In the following, we explain in details each one of these passes.

5.1 The Analysis Pass

As already said, the system restart begins with an analysis pass. This pass produces the necessary information for the next two passes, namely a list of the *active*/*in-doubt* transactions (Trans_Tab table), a list of the modified objects that probably were in the cache in the moment of the crash (Dirty_Object table), and the address of the last successfully, in its entirety written log record. 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 point on, the log records are read forward until the end of the log file, and interpreted according to their types. (*Savepoint* log records are not of interest for the system restart process. Hence, we disregard them from now on.)

- *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.

- *Update* or CLR

  The Last_LSN pointer 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 operation was performed, a new entry for this object is created in this table. In this case, this object's Dirty_LSN is set to this log record's LSN, signaling 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.

On reaching the end of the log file, the analysis pass is concluded and its results are available to the next passes:

- Trans_Tab Table

After processing the analysis pass, this table contains all transactions that were not terminated at the time of the system crash. These are the loser transactions, whose propagated update operations must be rolled back in the undo pass, therefore.

- Dirty_Object Table

  This table contains information about all objects which probably were in the cache at the crash time. For these objects, there may be operations to be undone or redone, what is determined accordingly to the LSN comparison and to the transactions' status (losers or non-losers) which accessed them.

- Last_Log

  It is the address of the last successfully written log record of the log file. The undo pass begins its work in this point.

5.2 The Undo Pass

The undo pass has two main tasks to take care of. On one hand, it is responsible for rolling back the loser transactions, on the other hand, it prepares the subsequent redo pass. In order to undo the operations of loser transactions in the exact reverse chronological order, the log file is read backward, starting from the Last_Log address established by the previous analysis pass. In addition, the knowledge about the loser transactions 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 the log record pertains to a loser or to a non-loser transaction. The following actions depends on this. Whereas the propagated updates of loser transactions are rolled back, the non-propagated updates of non-loser transactions are prepared for the subsequent redo pass, in that a *forward chain* is built.

Before passing on to the detailed explanation of the undo pass, we show in Fig. 14 a simplified example of this pass. The undo pass begins by processing the last log record of the log file (Fig. 14a). This log record, LSN 110, pertains to the loser transaction T2 and must therefore be rolled back, if it were already propagated into the DB (for the sake of simplicity in this example, we give up the LSN comparison 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 (it is an *update* log record) 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 this proceeding, the loser transactions are being undone (Fig. 14b), and the updates of non-loser transactions chained together (Fig. 14c). 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-loser transactions to the end of redoing them.

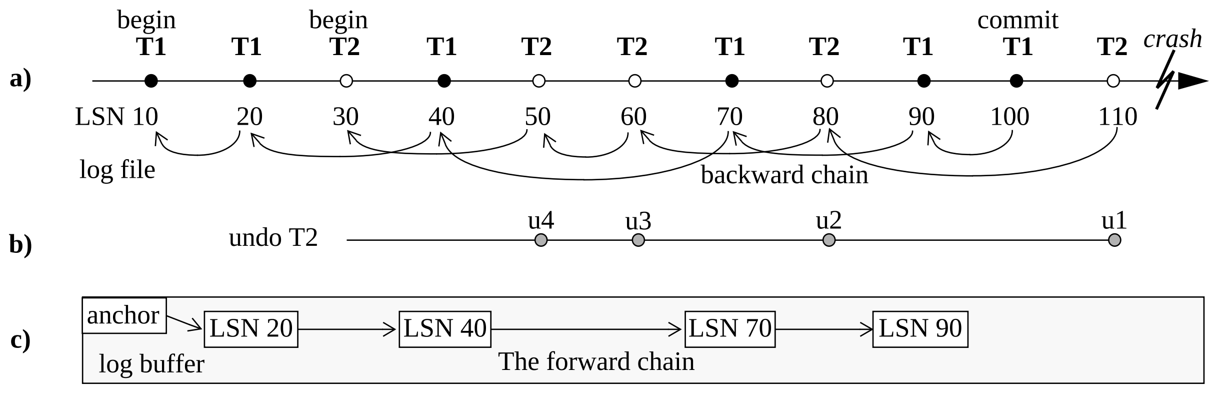

Figure 14: A simplified undo pass after a system crash.

5.2.1 The Forward Chain

As briefly explained, the undo pass prepares at run time a forward chain in the log buffer for the redo pass. The log records of non-loser transactions which were not yet propagated into the DB (i.e., the ones that require to be redone) are fetched 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-loser transactions and is built during restart processing.

The main purpose of the forward chain 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-loser transactions 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 Fig. 6 and Sect. 3.1, 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 transaction, when this transaction was partially rolled back before reaching its commit point.

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 fetch into the buffer the discarded log records by means of their retained addresses (LSNs).

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, operation 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 loser transactions 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 corresponding 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. 15 illustrates examples of these different cases. In this scenario, the transaction T1 was in process of being (partially or totally, it does not matter) 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 with respect to 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 first case, no undo measure needs to be taken, since the update was not propagated into the DB. In the second case, the update must be rolled back and a corresponding

CLR written. If a CLR already exists, which was not yet propagated though (third case), then the update is rolled back, and the affected object receives the LSN of the already written CLR. If the update and its CLR were already propagated (fourth case), then no more action is necessary. This is the correct way to handle the log records of loser transactions 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.

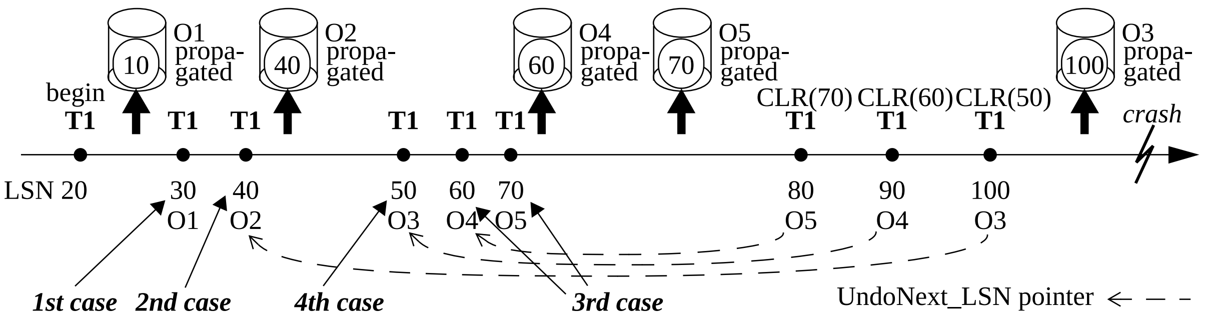

Figure 15: Different cases when handling log records of loser transactions in the undo pass.

Following the UndoNext_LSN Pointers of CLRs during System Restart

As previously illustrated in Fig. 15, not all undo operations of a loser transaction, 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 back a transaction during normal processing and rolling it back during system restart. Contrarily to the proceeding in normal processing, the CLR's UndoNext_LSN pointer shall not be followed during restart processing. In the scenario of Fig. 15, 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 of WALORS must investigate each log record of a transaction. In the example (Fig. 15), 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 update 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 out this operation, 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. (In general, that is true at least for non-commutative update operations performed on the same object.)

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. Generating CLRs for CLRs themselves may become very cumbersome for the recovery strategy, since in the worst case the number of log records written during repeated restart failures grows exponentially.

In order to reach the above mentioned 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 backward. 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. (This LSN is the value stored in the RedoNext_LSN field of this CLR during normal processing, see Sect. 4.2.)

- 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., both 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 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.

   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.

   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 described proceedings to the problematic scenario previously illustrated in Fig. 15. Table 1 shows the steps taken during the undo pass and the employment of both chains.

Table 1: 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
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* log record (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* log record (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* log record (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). Finally, the *begin* log record for T1 is found (LSN 20, 9th step), what means that the undo process of this transaction has successfully terminated, and both (empty) chains are then released.

Hence, by means of the Non_Undo and CLR_Exists chains, the undo operations are performed in the exact reverse order, and additionally, no CLRs are generated for CLRs. Further on, it should be noticed that such chains are created dynamically during the undo pass only for the loser transactions which were already in process of being rolled back before the system has crashed.

Non-Propagated Update Operations

Besides the above explained problem, there still is a situation WALORS must cope with when interpreting the log records during the undo process. We introduce this problem by means of an example (Fig. 16). Fig. 16a) illustrates the log records of a transaction, T1, which has made updates on an object, O1. However, not all the log records of this transaction were propagated into the DB before the system has crashed (e.g., LSN 40). When rolling back this transaction at system restart, the log record 40 is read, and it becomes clear that this update does not need to be undone by means of the LSN comparison. Since no undo action was taken, one could think of generating no CLR for this step. We show in the following that this may lead to problems. Proceeding with the example in Fig. 16b), the next log record to be read is 30. This log record was propagated into the DB, hence it is undone and a corresponding CLR is generated (LSN 50). After that, suppose the object O1 is again propagated into the DB and the system crashes once more. When starting again the undo process (Fig. 16c), the CLR(30) is read and the necessary measures for handling CLRs are taken (refer to the previous subsection). Thereafter, the update 40 is read and unfortunately the LSN comparison fails here. Since O1 was propagated again into the DB before the crash, it possesses now a new LSN, 50, which is greater than the one of log record 40. Therefore, the undo pass will try to undo update 40, although it was in fact never propagated into the DB.

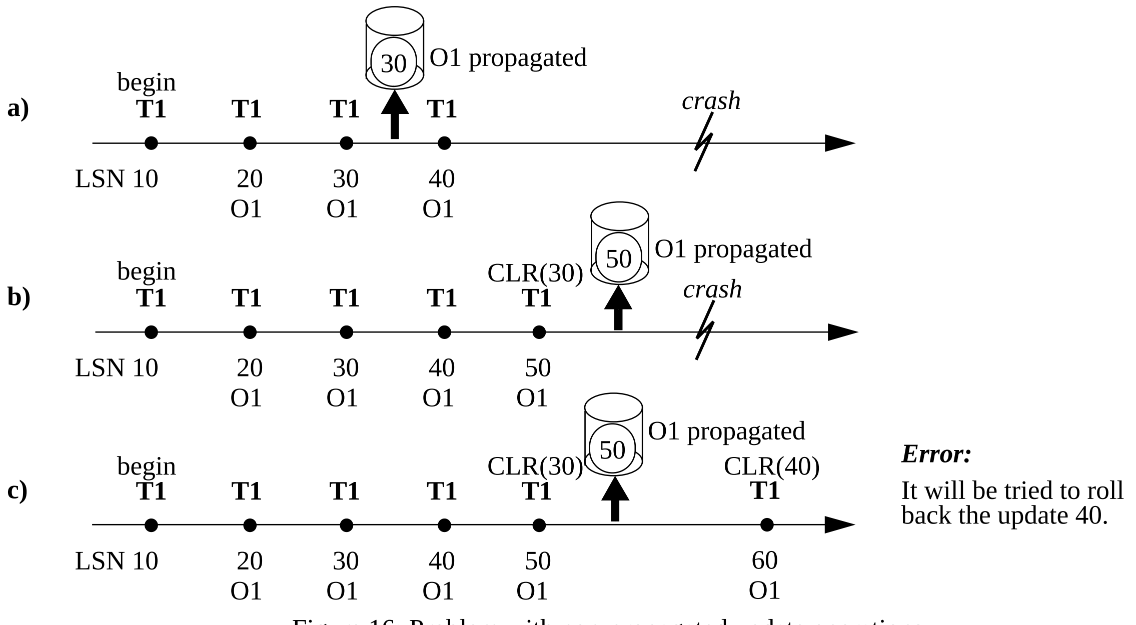

Figure 16: Problem with non-propagated update operations.

This problem with non-propagated update operations is in fact not new, and is intrinsic to recovery strategies employing CLRs. One of the easiest way to solve this problem is to generate, during the undo pass, CLRs even for the non-propagated update operations. Applying this measure to the example of Fig. 16, a CLR for the update 40 is generated before the CLR(30). When restarting again the system after the new crash, the undo pass reads the CLR(40) and, by means of the proceedings presented in the previous subsection, it inserts an entry for this CLR in the Non_Undo chain. Hence, due to this entry, when analyzing the update 40 it will not be tried to undo it. This is the simple way we cope with this problem in WALORS, i.e., generating CLRs even for non-propagated update operations.

5.2.3 Handling Object Inserts and Deletes

Probably the most cumbersome aspect of employing an LSN per object instead of per page gives respect to the manipulation 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. The log records are:

• *Update_Delete_Obj*; and

• *Update_Insert_Obj*.

These log records 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 back this error message, 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 transaction, 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* log record was not propagated.

- *CLR_Insert_Obj*:

The undo operation for the *Update_Insert_Obj* log record 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 Processing the Log Records of Loser Transactions: Summary

In the following, we summarize the processing of all types of log records that may be found for a loser transaction during the undo pass:

- *Begin*

  On reaching the *begin* log record of a loser transaction, its corresponding entry in the Trans_Tab table is deleted. The rollback process of this transaction is concluded.

- *Update*, *Update_Delete_Obj*, and *Update_Insert_Obj*

  On finding any of these log records for a loser transaction, it is first of all checked whether there exists a corresponding entry in the Non_Undo chain for this log record. If there is such an entry, this log record does not need to be rolled back, just its corresponding entry is removed from the Non_Undo chain. If there is not an entry in this chain, the LSN comparison is performed to decide whether the log record were already propagated or not. If necessary, an appropriate undo action is taken. When undoing the log record, it is decided whether a CLR must be generated by checking if there exists an entry for this update operation in the CLR_Exists chain. If there is not such an entry, a CLR is generated and the object receives this CLR's LSN. If an entry exists, no new CLR is generated and the object receives the LSN of the already generated CLR.

- CLR, *CLR_Delete_Obj*, and *CLR_Insert_Obj*

  On analyzing any of these CLRs, it is firstly checked whether this operation were already propagated by means of the LSN comparison. If so, a corresponding entry is created in the Non_Undo chain. If not, an appropriate entry is created in the CLR_Exists chain.

5.2.5 Processing the Log Records of Non-Loser Transactions: Summary

The goal here is to prepare all log records necessary for the redo pass. Just *update* and CLR log records need to be considered, since redo information is contained only in those. The proceeding is the same for any of them:

- *Update*, *Update_Delete_Obj*, *Update_Insert_Obj*, CLR, *CLR_Delete_Obj*, and *CLR_Insert_Obj*

  The LSN of the log record is compared with the LSN of the corresponding object. If, by means of this comparison, it is realized that the operation was already propagated into the DB, this log record is no longer considered, i.e., no more action is taken. On the other hand, if it were not yet propagated, this log record receives in its RedoNext_LSN field a pointer to the last log record of the forward chain (the remembered RedoNext_LSN variable).

In particular, it is interesting to notice here that one can make the processing of log records of non-loser transactions even more efficient, especially for CLRs. When a CLR of a non-loser transaction 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 so far, both the *update* log record 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 transaction 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 here.

5.2.6 Undo Pass Boundary

The log file is processed during the undo pass at least until the last successfully written checkpoint is reached. Each particular restart process has its specific undo pass boundary. On the one hand, all loser transactions must be rolled back. On the other hand, all log records relevant for the redo of non-loser transactions must be investigated. The rollback process is concluded when no more entries for loser transactions 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. As seen, this table contains information about the objects which were in the cache at system crash. 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 fetched into the cache. 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 follows:

- *min_LSN* := MIN (*min_losers_LSN*, *min_Dirty_LSN*)

  where: *min_losers_LSN* is the smallest LSN of the loser transactions; and

  min_Dirty_LSN is the smallest LSN between the Dirty_LSN fields of the Dirty_Object table.

Finally, during the undo pass all log records are processed until the one with the LSN *min_LSN* is reached. On reaching this log record, 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-loser transactions, in order to guarantee their durability. As explained, most work of this pass is already performed during the undo pass. At this point, all relevant log records were already fetched into the log buffer, and were chained by means of pointers in the right redo order. Hence, the redo pass simply receives the first pointer of this forward chain from the undo pass (illustrated as *anchor* in Fig. 14), 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, writing 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 issue a checkpoint operation.

6. Media Recovery

As soon as some part of the non-volatile storage is damaged, media recovery measures are necessary for re-establishing the DB consistency. We consider here an object as the smallest granule for media recovery. In this section, the granule of media recovery is referred to as *the entity to be recovered*. In order to support media recovery, the generation of archive copies of the DB as well as of the log file is fundamental. Archive copies of the log file may be easily generated by closing the current log file and opening another one. This process may be started at any time, e.g., at reaching a predetermined size, by determination of the DB administrator, from time to time, etc. On the other hand, there are several techniques for the generation of the DB archive copy, and all of them have some impact on recovery [12]. For the sake of efficiency, we assume that a *fuzzy copy* of the DB is made, i.e., the DB may be dumped with modifications by concurrent transactions. The main point to be coped with when allowing the DB to be accessed at dump time is that the archive copy may contain updates of uncommitted transactions.

When creating the archive copy of the DB, the address of the last complete checkpoint is remembered along with the data. Such an *archive checkpoint* is later used as starting point for media recovery. In order to recover from a media failure, the archive as well as the current log file are accessed, and the log records are step by step applied to the archive copy of the DB. In the following, we briefly describe the main steps for media recovery:

(1) Loading the archive copy of the entity to be recovered.

    The first step is to load from the archive copy of the DB the entity to be recovered.

(2) Evaluation of the *archive checkpoint*.

    The archive checkpoint is necessary to establish the starting point in the archive log file from which the redo measures must start. After the establishment of the archive checkpoint, the Dirty_Object table is created in accordance to what is to be recovered, and the minimum Dirty_LSN of the table is ascertained. This is necessary because there may be update operations of committed transactions which were not propagated into the DB at the time the DB archive copy was generated.

(3) Redoing the operations on the entity to be recovered.

Firstly, the log records (relating to the entity being recovered) of the archive log file are redone from the established starting point on. Thereafter, the log records of the current log file are considered for the same redo purposes. In particular, just the log records containing redo information are considered (i.e., all kinds of *update* and CLR log records).

(4) Undoing possible loser transactions.

On reaching the end of the log file after redoing the operations, it may happen that the DB is in a transaction inconsistent state. If there are any active transactions then those that had made changes to the entity being recovered are undone in a process similar to the undo pass of system restart previously explained.

In summary, media recovery is a simple task given the features of WALORS. The main points are the generation of the archive copies of DB and log files and to remember the last complete checkpoint.

7. Conclusions

In this paper, we have presented the properties and the functionality of WALORS, a WAL-based and object-oriented recovery strategy. 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 the principle of write-ahead logging, 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 flexible enough with respect to the kinds of cache 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 probably were in the cache at system crash. The undo pass rolls back the propagated updates of loser transactions, and builds a forward chain for the following redo pass. At this time, special chains are considered in order to roll back the operations in the exact reverse order, and to avoid CLRs from 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-loser transactions by means of the forward chain built during the previous pass. In particular, this chain allows for a very efficient redo pass, since all relevant log records are fetched into the buffer and chained together in the right redo order. Finally, insert and delete update 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, media recovery is a simple task given all the features of WALORS.

In particular, we have already extended WALORS, coming up with two major versions of it, to work in a client/server environment, and to support nested transactions. Additionally, we have implemented this extended WALORS with all its functionality using as a running example KRISYS [18], a knowledge base management system (KBMS) prototype. In this context, WALORS follows the nested transaction model defined in [24], and supports the enhanced lock modes for KBMSs presented in [23, 25]. Due to space limitations, we have not considered these extensions in this paper.

Acknowledgments: *I would like to thank Prof. Theo Härder for his proposals concerning the structure and contents of this paper. In addition, he shared with me his great knowledge on recovery during several enlightening discussions on the matter. Thanks are also due to Thomas Baier, whose teamwork was fundamental for the realization of WALORS.*

References

[1] B. R. Badrinath and K. Ramamritham. Synchronizing Transactions on Objects. *IEEE Transactions on Computers*, 37(5):541-547, 1988.

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

[3] P. K. Chrysanthis, S. Raghuram and K. Ramamritham. Extracting Concurrency from Objects: A Methodology. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Denver, USA, pages 108-117, 1991.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database Systems. *Communications of the ACM*, 19(11):624-633, 1976.

[5] A. A. Farrag and M. T. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503-525, 1989.

[6] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186-213, 1983.

[7] D. Gawlick. Processing Hot-Spots in High Performance Systems. In *Proc. of the IEEE Computer Society Int. Conference*, San Franscisco, USA, pages 249-251, 1985.

[8] J. N. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, LNCS 60, Springer Verlag, Germany, 1978.

[9] J. N. Gray. Super-Servers: Commodity Computer Clusters Pose a Software Challenge. In *Proc. of the GI-Conf. on Database Systems for Office, Engineering and Science Environments*, Dresden, Germany, pages 30-47, 1995.

[10] J. N. Gray, R. Lorie, F. Putzolu and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Proc. of the IFIP Working Conf. on Modelling in Data Base Management Systems*, Freudenstadt, Germany, pages 365-394, 1976.

[11] J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu and I. L. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223-242, 1981.

[12] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, USA, 1993.

[13] T. Hadzilacos and V. Hadzilacos. Transaction Synchronization in Object Bases. *Journal of Computer and Systems Sciences*, 43(1):2-24, 1991.

[14] T. Härder. Handling Hot-Spot Data in DB-Sharing Systems. *Information Systems*, 13(2):155-166, 1988.

[15] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287-317, 1983.

[16] B. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms and R. A. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems*, 2(1):24-38, 1984.

[17] P. C. Lockemann and J. W. Schmidt (Eds.). *Databases Handbook* (in German). Springer Verlag, Germany, 1987.

[18] N. M. Mattos. *An Approach to Knowledge Base Management*. LNAI 513, Springer Verlag, Germany, 1991.

[19] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwarz. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. IBM Research Report RJ6649, IBM Almaden Research Center, USA, 1989.

[20] C. Mohan, I. Narang and S. Silen. *Solutions to Hot-Spot Problems in a Shared Disks Transaction Environment*. IBM Research Report RJ8281, IBM Almaden Research Center, USA, 1991.

[21] C. Mohan and H. Pirahesh. ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method. In *Proc. of the Int. Conf. on Data Engineering*, Kobe, Japan, pages 718-727, 1991.

[22] T. C. Rakow, J. Gu and E. J. Neuhold. Serializability in Object-Oriented Database Systems. In *Proc. of the Int. Conf. on Data Engineering*, Los Angeles, USA, pages 112-120, 1990.

[23] F. F. Rezende and T. Härder. A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In *Proc. of the Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA, pages 112-121, 1994.

[24] F. F. Rezende and T. Härder. Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs. In *Proc. of the Int. Conf. on Database and Expert Systems Applications*, London, U.K., pages 604-613, 1995.

[25] F. F. Rezende and T. Härder. Multiple Granularity Locks for the KBMS Environment. In J. Fong and B. Siu (Eds.), *Multimedia, Knowledge-Based & Object-Oriented Databases*, Springer Verlag, Singapore, pages 126-148, 1996.

[26] P. M. Schwarz and A. Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2(3):223-250, 1984.

[27] I. Traiger. Virtual Memory Management for Data Base Systems. *ACM Operating Systems Review*, 16(4):26-48, 1982.

[28] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488-1505, 1988.