

# ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method

C. Mohan  
Hamid Pirahesh

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA  
mohan@ibm.com, pirahesh@ibm.com

**Abstract** This paper presents a method, called ARIES-RRH (*Algorithm for Recovery and Isolation Exploiting Semantics with Restricted Repeating of History*), which is a modified version of the ARIES transaction recovery and concurrency control method implemented to varying degrees in Starburst, QuickSilver, the OS/2<sup>1</sup> Extended Edition Database Manager, DB2<sup>1</sup> V2, Workstation Data Save Facility/VM and the Gamma data base machine. ARIES redoes, during restart after a system failure, *all* updates which had been logged to stable storage but whose effects on the data base pages had not yet been reflected in nonvolatile storage before the failure. This *repeating history* paradigm of ARIES includes redoing the updates of even those transactions that are to be rolled back later in the undo pass of restart. The latter may lead to some wasted work being done. It was pointed out in the ARIES paper that repeating history was required to support fine-granularity (e.g., record) locking. This paper further analyzes this paradigm and proposes more efficient handling of redos, especially when the smallest granularity of locking is *not* less than a page, by combining the paradigm of *selective redo* from DB2 V1. Even with fine-granularity locking, it is not always the case that all the unapplied but logged changes need to be redone. ARIES-RRH, which incorporates these changes, still retains all the good properties of ARIES - avoiding undo of undos, single pass media recovery, nested top actions, etc. In this paper, we also explain the fundamentals behind why DB2 V1's selective redo works, in spite of failures during restart recovery.

## 1. Introduction

In transaction processing systems, the problem of performing recovery from transaction, system and media failures efficiently, especially in conjunction with support for fine-granularity locking and semantically-rich modes of locking [BaRa87], had been an open problem for a long time. While concurrency control has received widespread attention in the theoretical and the practical-minded research community, recovery methods and storage management techniques have not received enough attention, even though they are also crucial for the correct and efficient functioning of a transaction processing system. This is especially unfortunate considering the fact that these three topics are highly intertwined if high concurrency, efficient recovery, and flexible storage man-

agement for varying length objects are desired (see the lengthy discussions in [MHLPS89] which illustrate these points). Even when recovery is considered, most often it is only normal transaction rollback, rather than the more complicated crash recovery problem, that is considered. The ARIES transaction recovery and concurrency control method introduced in [MHLPS89] was the first comprehensive solution to this problem. ARIES has been implemented, to varying degrees, in the IBM products OS/2 Extended Edition Database Manager<sup>1</sup> [ChMy88], Workstation Data Save Facility/VM and DB2<sup>1</sup> V2R1, in the IBM Research prototypes Starburst [SCFLM86] and QuickSilver [HMSC88], and in the University of Wisconsin's Gamma data base machine [DGSBH90].

In this paper, we present a method, called **ARIES-RRH** (*Algorithm for Recovery and Isolation Exploiting Semantics with Restricted Repeating of History*), which is an enhanced version of ARIES. ARIES redoes, during restart after a system failure, *all* updates which had been logged to stable storage but whose effects on the data base pages had not yet been reflected in nonvolatile storage before the failure. This **repeating history** paradigm of ARIES includes redoing the updates of even those transactions that are to be rolled back later in the undo pass of restart. The latter feature of ARIES may lead to some wasted work being done which delays the completion of restart processing and the initiation of processing of new transactions. It was pointed out in [MHLPS89] that repeating history was *required* to support fine-granularity (i.e., smaller than page-granularity) locking. This paper further analyzes this paradigm and proposes more efficient handling of redos, especially when the smallest granularity of locking is not smaller than a page. ARIES-RRH combines the paradigm of *selective redo* from DB2 V1 with the repeating history paradigm of ARIES. Even with fine-granularity locking, it is not always the case that all the unapplied but logged changes need to be redone. ARIES-RRH, which incorporates these changes, still retains all the good properties of ARIES like avoiding undo of undos, single pass media recovery, nested top actions, and so on. In this paper, we also explain the fundamentals behind why DB2 V1's selective redo works, in spite of failures during restart.

The rest of the paper is organized as follows. For the benefit of those readers that may not be completely

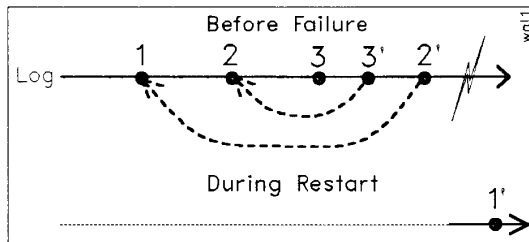
<sup>1</sup> AS/400, DB2, IBM and OS/2 are trademarks of the International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. DEC, VAX DBMS, VAX, VAXcluster and Rdb/VMS are trademarks of Digital Equipment Corp. Informix is a registered trademark of Informix Software, Inc.

familiar with some of the recovery concepts, we introduce some of the basic logging and recovery ideas in appendix A. If the meaning of some term or phrase used in the rest of the paper is not clear, the reader should refer to the appendix. In section 2, we present an overview of ARIES. In section 3, we discuss the benefits of selective redo and what problems we face in achieving it. In section 4, we consider the effect of coarse-granularity (e.g., page) locking. Fine-granularity locking is taken into account in section 5. We summarize in section 6.

## 2. ARIES

The aim of this section is to provide a brief overview of ARIES. The reader is referred to [MHLPS89] for the details about ARIES and to [RoMo89] for the description of ARIES/NT, which is the extension of ARIES to the nested transactions model. Index management algorithms are described in [Moha90a, MoLe89], while the algorithm for a linear hashing based storage method is presented in [Moha90b].

ARIES keeps track of the changes made to the data base by using a log and it implements the write-ahead logging (WAL) protocol. In addition to logging, on a per-affected-page basis, update activities performed during forward processing of transactions, ARIES also logs, typically using compensation log records (CLRs), updates performed during partial or total rollbacks of transactions during both normal and restart processing. In ARIES, CLRs have the property that they are redo-only log records. By appropriate *chaining* of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart recovery or of nested rollbacks. This is to be contrasted with what happens in IMS [PeSt83], which may undo the same nonCLR multiple times, and in AS/400<sup>1</sup> [CICo89], DB2 and NonStop SQL<sup>1</sup> [Tand87], which, in addition to undoing the same nonCLR multiple times, may also undo CLRs one or more times (see [MHLPS89] for examples). These have caused severe problems in real-life customer situations.



<sup>1</sup> I' is the CLR for I. Only UndoNxtLSN chain is shown (I' has a NULL pointer). PrevLSN chain should be obvious.

Figure 1: ARIES Recovery Scenario - Log Records of a Single Transaction

When the undo of a log record causes a CLR to be written, the CLR is made to point, via the **UndoNxtLSN** field of the CLR, to the *predecessor* of the log record being undone. The latter information is readily available since every log record, including a CLR, contains a pointer (**PrevLSN**) to the most recent preceding log record written by the same transaction. For example, in Figure 1, CLR 1' has a NULL UndoNxtLSN value because log record 1 has a NULL PrevLSN, which means that the transaction has been completely rolled back. Thus, during rollback, the UndoNxtLSN field of the *most recently written CLR* keeps track of the progress of rollback. It tells the system from where to continue the rollback of the transaction, if a system failure were to interrupt the completion of the rollback or if a nested rollback were to be performed. It lets the system bypass those log records that had already been undone.

Since CLRs are available to describe what actions are actually performed during the undo of an original action, the undo action need not be, in terms of which page(s) is affected, the exact inverse of the action that is being compensated (i.e., logical undo is made possible). This allows very high concurrency to be supported. For example, in a B<sup>+</sup>-tree, a key inserted on page 10 by one transaction may be moved to page 20 by another transaction *before* the key insertion is committed, as we permit in ARIES/IM [MoLe89] (see [Moha90b] for the description of ARIES/LHS which also exploits this feature). Now, if the first transaction were to roll back, then the key will be located on page 20 by retraversing the tree and deleted from there. A CLR will be written to describe the key deletion on page 20. This enables page-oriented redo which is very efficient.

ARIES uses a log sequence number (LSN) on every data base page to track the page's state. Every time a page is updated and a log record is written, the LSN of the log record is placed in the **page\_LSN** field of the updated page. This tagging of the page with the LSN allows ARIES to precisely track, for restart- and media-recovery purposes, the state of the page with respect to logged updates for that page. It allows ARIES to support novel lock modes, using which, before an update performed on a record by one transaction is committed, another transaction may be permitted to modify the same data for specified operations (e.g., increment/decrement type operations [BaRa87]).

Periodically during normal processing, ARIES takes checkpoints. The checkpoint log records identify the transactions that are active, their states, and the addresses of their most recently written log records, and also the modified data (*dirty* data) that is in the buffer pool. During restart recovery, ARIES first scans the log from the last checkpoint to the end of the log. During this **analysis pass**, information about dirty data and transactions that were in progress at the time of the checkpoint is brought up to date as of the end of the log. The analysis pass using the dirty data information determines the starting point (**RedoLSN**) for the log scan of the redo pass. The analysis pass also determines the list of transactions to be rolled back in the undo pass. Then, during

the redo pass, ARIES repeats history, with respect to those updates logged on stable storage, but whose effects on the data base pages did not get reflected on disk before the system failure. This is done for the updates of ALL transactions, including the updates of those transactions that had neither committed nor reached the in-doubt state of two-phase commit by the time of the crash (i.e., even the missing updates of the so-called loser transactions are redone). This essentially reestablishes the state of the data base as of the time of the failure. A log record's update is redone if the affected page's page\_LSN is less than the log record's LSN.

The next pass is the undo pass during which all loser transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. This is done by continually taking the maximum of the LSNs of the next log record to be processed for each of the yet-to-be-completely-undone loser transactions, until no loser transaction remains to be undone. Unlike during the redo pass, during the undo pass (as well as during normal undo), performing undos is not a conditional operation. That is, ARIES does not compare the page\_LSN of the affected page to the LSN of the log record to decide whether or not to undo the update. Once a log record is processed for a transaction, the next record to process for that transaction is determined by looking at the PrevLSN or the UndoNxtLSN field of the record, depending on whether it is a nonCLR or a CLR, respectively.

There are times when we would like some changes of a transaction to be committed irrespective of whether later on the transaction as a whole commits or not. We do need the atomicity property for these changes themselves. A few of the many situations where this is very useful are: for performing page splits and page deletes in indexes [MoLe89] and for relocating records in a hash-based storage method [Moha90b]. ARIES supports this via the concept of nested top actions. The desired effect is accomplished by writing a dummy CLR at the end of the nested top action (see Figure 2, where records 3, 4 and 5 constitute the nested top action). The dummy CLR has as its UndoNxtLSN the LSN of the most recent log record written by the current transaction just before it started the nested top action. Thus, the dummy CLR lets ARIES bypass the log records of the nested top action if the transaction were to be rolled back after the completion of the nested top action. ARIES's repeating history feature ensures that the nested top action's changes would be redone, if necessary, after a system

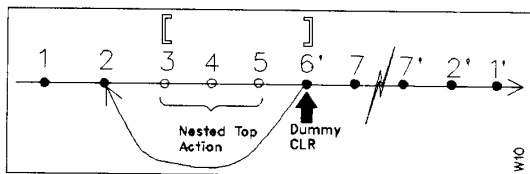


Figure 2: Using ARIES, a Completed Nested Top Action's Effects Persist In Spite of Transaction Rollback

failure even though they may be changes performed by a loser transaction. If a system failure were to occur before the dummy CLR is written, then the incomplete nested top action will be undone since the nested top action's log records are written as undo-redo (as opposed to redo-only) log records. This provides the desired atomicity property for the nested top action itself.

ARIES supports selective and deferred restart, fuzzy image copies (archive dumps), media recovery, and high concurrency lock modes (e.g., increment/decrement), which exploit the semantics of the operations and which require the ability to do operation logging. It is flexible with respect to the kinds of buffer management policies (e.g., steal, no-force, etc. [HaRe83]) that can be implemented and the characteristics of the stored data. Efficient storage management can be done for varying length objects. Page-oriented redos and logical undos are permitted. Opportunities also exist for exploiting parallelism during restart recovery.

### 3. Selective Redo: Problems and Benefits

As shown in Figure 3, in the recovery methods of System R [GMBLL81] and DB2 V1 [Crus84], only the missing updates of terminated and in-doubt transactions (the nonloser transactions) are redone during the redo pass. We call this the selective redo paradigm. In [MHLPS89], we show why this paradigm could lead to problems when fine-granularity (i.e., smaller than page-granularity) locking is to be supported with WAL. Figure 4 and Figure 5 illustrate the problem. In the first scenario, the page on disk already has the effect of the loser transaction's (T2's) update (20) and so selective redo does not cause a problem when conditional undo is performed. A problem comes in the second scenario because the disk ver-

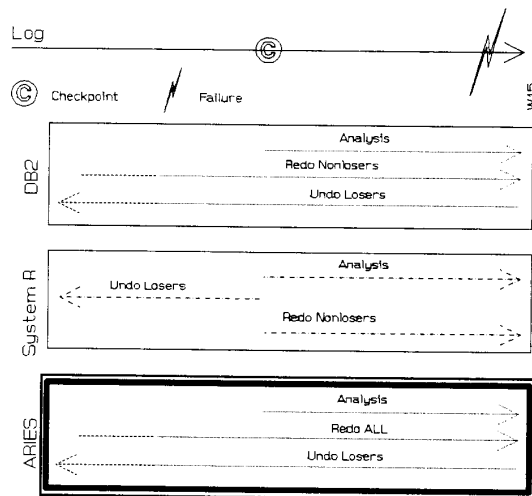
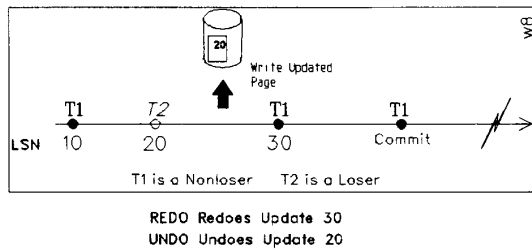


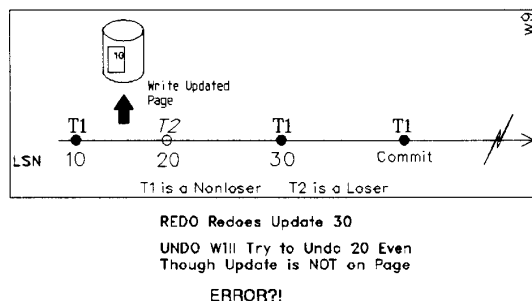
Figure 3: Restart Processing in Different Methods



**Figure 4: Selective Redo with WAL - Problem-Free Scenario**

sion of the page does not have T2's update and selective redo results in that update *not* being redone but a subsequent one (30 by T1) being redone. The latter results in page\_LSN becoming 30. Once such a situation is allowed to come about, the true state of the page becomes *ambiguous*. That is, the interpretation of the LSN (30) of the page can no longer be that *definitely* all updates that have been logged, for that page, up to that LSN are reflected in the current version of the page. Undoing an update when it is not currently reflected in the page would be correct only if physical logging is done (e.g., as in IMS). With logical/operation logging (e.g., decrement/increment information) it will lead to destruction of the integrity of the data. Note that the depicted scenarios are possible only if the locking granularity is smaller than a page.

Since ARIES was designed to support fine-granularity locking, the solution adopted to avoid such ambiguous situations was not to perform selective redo but to follow the repeating history paradigm. In System R, even with fine-granularity locking, selective redo does not cause problems because in that system the shadow-page technique is used for recovery. In System R, no LSNs exist on pages and whether or not a nonloser (respectively, loser) transaction's log record is redone (undone) is decided solely based on whether or not the record succeeded (preceded) the most recent checkpoint before the system failure (see [MHLPS89] for a detailed expo-



**Figure 5: Selective Redo with WAL - Problem Scenario**

sition). In DB2 V1, problems are avoided since page locking is the smallest granularity of locking. The depicted scenarios are impossible with page locking. That is, an uncommitted update (by T2) cannot precede a committed update (by T1). Since DB2 V1 does not repeat history, *conditional undo* is performed in that system. That is, during the *restart undo* of a transaction, a CLR or a nonCLR may be encountered and it may be found that the corresponding data base page's LSN is *less than* the LSN of the log record. In this case, there is nothing to undo, but *still the CLR for that record is written*. We explain later the fundamental reasons behind why the latter is necessary.

During the design of ARIES, it was felt that repeating history was fundamental to guaranteeing proper recovery as long as fine-granularity locking was in effect for any part of the data base. Here, we show that, while repeating history of all logged updates is a *sufficient* condition to support fine-granularity locking with WAL, it is not a *necessary* condition. In this paper, we combine the paradigms of selective redo and repeating history so that only when necessary repeating history is performed for *some data*. Even for such data, not all of the logged changes have to be repeated.

The motivations behind the work reported were:

- The first motivation was a scientific one - to gain a much better understanding of the fundamental interactions between recovery and concurrency control, and to ascertain the minimum required redo processing, especially because ARIES has become very popular in terms of implementations (DB2, the OS/2 Extended Edition, Workstation Data Save Facility/VM, QuickSilver, the Gamma data base machine and Starburst) and interest in the research community [DGSBH90, Hsia90, KoLS90, Kuma90, Lome90, Rahm89, Spec89, SPSW90, Vura90, WHBM90, WHMZ90]. The research community still does not have a very good understanding of the subtle interactions between concurrency control and recovery. A partial evidence of this is the fact that even in a textbook devoted to concurrency control and recovery in data base systems [BeHG87], there is only 1.5 pages worth of discussion of fine-granularity locking with WAL and operation logging. Even there an incorrect algorithm is presented (undo pass precedes redo pass).
- The second motivation was to gain some performance benefits that are already present in the DB2 V1 method, especially when only page locking is being used with ARIES. Intuitively, it seems like a waste of resources to redo some work during restart recovery for *loser* transactions only to undo them a little later during the undo pass. The expected savings are the following:
  - Reduce the number of I/Os to data base pages during the redo pass by avoiding reading into the buffer pool those pages, in the dirty data list, for which log records written by only *loser* transactions are encountered.

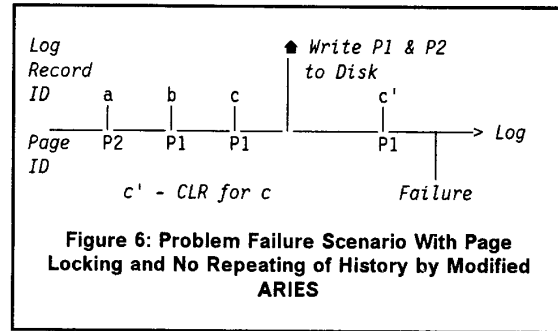
This will permit the redo pass to be completed faster than in ARIES and thereby permit new transaction processing activity to start earlier, if, during the redo pass, locks are going to be obtained to protect the uncommitted updates so that new transactions may be allowed into the system as soon as the redo pass finishes. Then, the undo of the loser transactions may be performed in parallel with the processing of new transactions. For example, in IBM's IMS/XRF [IBM87] and in Tandem's NonStop SQL, when *hot standbys* are defined, this is what is done when the primary system fails and the backup takes over. This improves the availability of data, especially if some long-running loser transactions have to be undone. The latter types of transactions are not at all uncommon.

- Reduce the number of pages dirtied by restart processing by not redoing some updates during the redo pass only to undo them later during the undo pass. This should reduce the number of write I/Os of dirty pages. It also saves the CPU pathlengths involved in redoing and undoing updates. These again should allow new transaction processing to start earlier, even if the earlier-mentioned features of IMS/XRF and Tandem's systems are not adopted.

The amount of unnecessary redo work that could be saved would depend on (1) the speed at which the buffer manager writes dirty pages to disk and (2) the duration of execution of individual update transactions. Of course, there is a trade-off with respect to the first point. The more quickly the buffer manager writes the pages, the lesser the amount of redo work to be performed even if repeating history were to be performed. But then, if there is any locality of reference amongst a set of pages across different transactions, then the quick writing of dirty pages would not allow us to amortize the cost of disk write of a page across multiple updates to the same page by different transactions or by a single long transaction. Frequent writing of hot spot pages would also cause concurrency problems if a page is not going to be allowed to be modified when it is being written to disk. Systems like DB2 delay doing writes also in order to accumulate multiple dirty pages for a single file so that the capability of the operating system to write multiple pages using a single *start I/O* command could be exploited to reduce the CPU and I/O overheads [TeGu84]. For reasons like these and also to reduce the lock holds times, systems like DB2 and NonStop SQL follow the no-force buffer management policy.

#### 4. Coarse-Granularity Locking

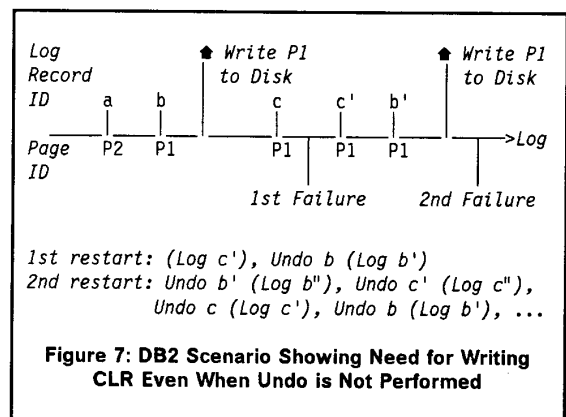
Suppose the granularity of locking is a page or something bigger than that. Let us call this *coarse-granularity locking*. This means that at any time a page could have the uncommitted updates of at most one transaction. Since recovery works properly in DB2, which supports only coarse-granularity locking, even with selective redo, the interesting question is could ARIES be modified to perform selective redo for those objects on which only



coarse-granularity locking is in effect, even though there may be other objects for which fine-granularity locking is in effect. On the surface this may appear to be an acceptable thing to do given that selective redo works in DB2. The only modifications to ARIES that would appear to be needed are:

1. An identification in each log record of whether fine-granularity or coarse-granularity locking was in effect when that log record's update was performed.
2. During the redo pass, if a log record written by a *loser transaction* is encountered and it indicates that coarse-granularity locking was in effect, then do *not* check the affected page to see whether the update needs to be redone.
3. During the undo pass, if a log record to be undone is encountered, then undo it only if the affected page's *page\_LSN* is *greater than or equal to* the LSN of the log record. It is this type of conditional undo that DB2 does during restart recovery.

It turns out that the above changes alone are not sufficient. The problem comes when a system failure occurs while a transaction is rolling back and some of its CLRs had already been written to stable storage, but not all of the effects of the CLRs had been reflected in the



nonvolatile storage versions of the data base pages. Let us consider the scenario depicted in Figure 6, where coarse-granularity locking is assumed to be in effect. In the figure, a CLR for a record is indicated with the ID of the original record followed by a prime ('). Here, pages P1 and P2 were written to disk after the updates of log records a, b and c had been performed on them. We assume that all the log records were written by a single transaction which was still in flight when the system failure happens. Even though, before the system failure, log record c had been undone, resulting in the CLR c' being written, the effect of c' was *not* reflected in the disk version of page P1. For this scenario, with the above proposed changes, modified ARIES would not redo any actions during the redo pass, but will take the following actions during the undo pass:

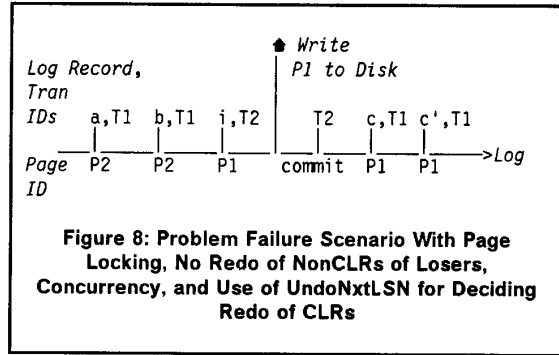
undo b (log b'), undo a (log a')

Note that c would not even be encountered during the undo pass due to the fact that the UndoNxtLSN pointer in c' will be pointing to b. Since the update of c was written to disk before the failure, if c' is not redone, then the system would not be ensuring the atomicity property of transactions and that is not acceptable. The reader may be curious as to how DB2's selective redo can avoid this problem. The explanation has to do with the fact that DB2 does not treat CLR's in any special way and so during the undo pass all log records will be processed and based on the page\_LSN the undo will or will not be actually performed. In any case, *CLR's will be written for all encountered records*. In the above scenario, DB2 will take the following actions during the undo pass:

(log c''), undo c (log c'), undo b (log b'), undo a (log a')

Note that c' is *not* undone since its change was not written to nonvolatile storage before the failure, but a CLR (c'') is written for it. The example in Figure 7 illustrates the reason for DB2 having to write a CLR even when an undo is not actually performed. In this scenario, during restart recovery after the first failure, even though c did not require an undo, c' was written. Then, b was undone, b' was written, and P1's LSN was set to the LSN of b' (which will be higher than the LSN of c). After these events happen, let P1 be written to disk and let a second failure happen. Then, during restart recovery after the second failure, b' would be undone. Even though c and c' were never on the page, because the LSN of the page was made higher by the initial undo of b and the subsequent undo of b', DB2 would undo c' and c also. If c' had not been written during the restart after the 1st failure just because c did not require an undo, then, during restart recovery after the 2nd failure, only c would be undone. The latter would be wrong since the update of c is not part of the page. By writing c' and letting it be undone, DB2 in effect puts in the update of c on P1 and lets it be undone later when c is undone.

With ARIES, the above problem does not occur because (1) CLR's are never undone and (2) each nonCLR is undone only once as it is encountered only once during rollback processing (due to the UndoNxtLSN pointer in CLR's). The number of failures during restart does not



affect the amount of logging or undo work performed by ARIES. With ARIES, during the restart after the 2nd failure, since the UndoNxtLSN field of b' would point to a, it would do the following: undo a (log a'). Hence, the only reason ARIES would have to write CLR's even when updates do not have to be actually undone is to make media recovery simpler. If no look-ahead with complex processing is employed during media recovery, then c would be redone as soon as that log record is encountered. Since there would be no c' later, the effects of c would persist, thereby violating the transaction atomicity property. This is the reason c' should be written even though during restart recovery the effects of c did not have to be actually undone on the page.

Now, referring back to the scenario of Figure 6, the problem is how to fix modified ARIES, call it **ARIES-RRH** (*ARIES with Restricted Repeating of History*), so that it will realize on encountering c' during the redo pass that it needs to be redone. On first thought, one might guess that we could exploit the UndoNxtLSN value and say that a CLR should be redone if the page\_LSN has a value **greater than** the UndoNxtLSN value in that CLR. While this would work in the above scenario, in the general case it will not. Consider the changes depicted in Figure 8 to the scenario of Figure 7. In this case, when c' is encountered during the redo pass, the LSN on P1 would be found to be higher than b (UndoNxtLSN value in c'), but that does not necessarily mean that the update of c is part of the page. In this scenario, note that P1's LSN is higher only because of the update i performed by transaction T2. Given just the UndoNxtLSN value in c' and the fact that the page's LSN is greater than the former, we cannot tell whether the situation is like the one shown above or is one in which the page was written to disk after the update logged in c was performed. What is really needed to resolve this ambiguity is to know precisely what the LSN of c is. If the page LSN is **greater than or equal to** the LSN of c (and *less than* the LSN of c'), then c' would have to be redone. That is, a CLR's update must be redone only if (1) the *original* update which the CLR compensated is already reflected in the page, *and* (2) the CLR's update is not yet reflected in the page. Hence, in ARIES-RRH, we modify the logging done during undo processing so that the CLR also con-

tains the LSN of the record whose undo caused the CLR to be written. We call this the *UndoneLSN*.

Now, we can state the rules to be followed by ARIES-RRH during redo processing:

- Treat nonCLRs and CLRs written when fine-granularity locking was in effect the same way ARIES treats them. That is, redo such a log record if the page\_LSN is *less than* the LSN of the log record.
- If a nonCLR or CLR written by a terminated or in-doubt transaction when coarse-granularity locking was in effect is encountered, then redo its change if the page\_LSN is *less than* the LSN of the log record.
- Ignore a nonCLR written by a loser transaction when coarse-granularity locking was in effect.
- If a CLR written by a loser transaction when coarse-granularity locking was in effect is encountered, then redo its change if (1) the page\_LSN is *less than* the LSN of the CLR, AND (2) the page\_LSN is *greater than or equal to* the UndoneLSN found in that CLR.

The rules to be followed by ARIES-RRH during restart undo processing are:

- If a CLR is encountered, use its UndoNxtLSN value to determine the next log record to process. If the UndoNxtLSN value is NIL, then the transaction has been completely undone.
- If a nonCLR is encountered, then examine the page affected by the log record. If the page\_LSN is *equal to or greater than* the log record's LSN, then undo the change; otherwise, there is no undo work to be performed. In either case, write a CLR that describes the undo action that was or would have been performed.

The reason for writing the CLR even when the undo is not actually performed has to do with making media recovery simpler. Since media recovery will be done in a single pass of the log, all log records whose updates are missing would be redone. Therefore, if a nonCLR's update is redone during media recovery for a transaction which rolled back and a corresponding CLR had not been written during some earlier restart undo processing which completed the rollback of that transaction, then, at the end of media recovery, that update's effects would persist. This would be a violation of the transaction atomicity property. As explained in [MHLPS89], the powerful property of media recovery in ARIES is that there is no need to pay attention to the transaction state change log records during such processing.

## 5. Fine-Granularity Locking

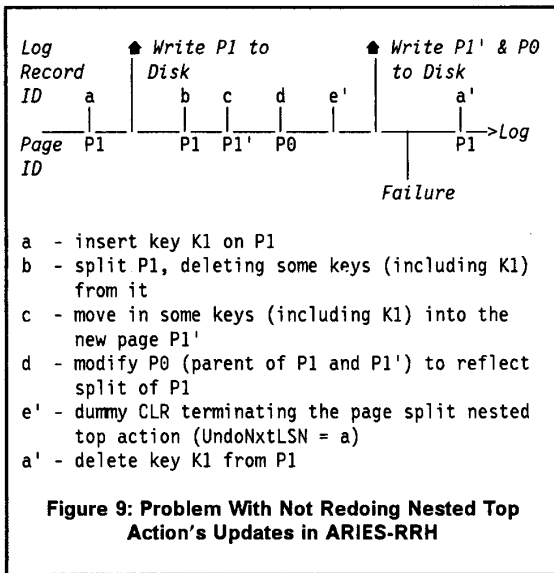
Even for the data for which fine-granularity (e.g., record) locking is being done, ARIES-RRH can do better than what ARIES does. The more generalized version of the previous section's changes would be to require that, for

*each page* for which fine-granularity was in effect, repeating of history be performed *only until* the update of the *most recent* log record written for that page by a *nonloser* transaction is redone. All subsequent *nonCLRs* written by loser transactions for that page need not be redone, while any CLRs that are encountered are to be treated the same way as they were in the case of coarse-granularity locking. Note that, with these changes, the problem depicted in Figure 5 will not occur.

The major problem with adding the enhancements of this section is that during the redo pass being able to determine what is the last update to a particular page by a nonloser transaction requires looking ahead in the log or at least keeping an upper bound, *across all pages*, of the last log record written by such transactions. The upper bound may be easily computed during the analysis pass. Unfortunately, this value is bound to be at least as high as the LSN of the last checkpoint and if there is even one short transaction that terminated *after* the checkpoint, then the upper bound will be even higher.

Another possibility is to use the dirty pages list that is available at the end of the analysis pass. This list is generated based on the list logged at the time of a checkpoint as further modified during the analysis pass, as described in [MHLPS89]. The log may then be analyzed in a new pass in between the analysis and redo passes by starting from the restart redo point. The cost of this extra log pass will be very small since all the accessed log records would have any way been needed during the redo pass. During this new pass, based on the information about the list of loser and in-doubt transactions which will be available at the end of the analysis pass, for each of the pages in the list, ARIES-RRH can keep track of the log record with the *highest LSN* which was written by a *nonloser* transaction. At the end of this pass, for each page in the list, there will be two LSNs: the *starting LSN* from which log records may have to be redone for that page (conditionally, based on the actual page\_LSN in the disk version of the page) and the *stopping LSN* beyond which no *nonCLRs* have to be redone for that page.

A third possibility is to take advantage of the *force* policy [HaRe83] that is implemented in systems like IBM's IMS [PeSt83], DEC's VAX DBMS<sup>1</sup> and VAX Rdb/VMS<sup>1</sup> [ReSW89], and MCC's ORION [KGBW90]. This policy results in all pages modified by a transaction being written to nonvolatile storage at the time the transaction goes in-doubt and/or at the time of termination of the transaction. With such a policy in use, there is no need to perform any redo activities for any pages of the data base. Unlike the systems listed that follow this policy, ARIES-RRH will permit semantically rich lock modes and allow multiple transactions' uncommitted changes to be present in the same field since (1) it uses LSNs in the nonvolatile storage versions of pages, and (2) it does operation logging as opposed to doing before-image and after-image logging always. ARIES and ARIES-RRH will undo a change only if the change is definitely known to be present in the page. Idempotence is not assumed



because of physical logging and locking as many systems do.

Nested top actions which are used to perform operations like index page splits and page deletes in ARIES/IM [MoLe89] and record relocations in linear hashing in ARIES/LHS [Moha90b] require special handling. A nested top action's updates have to be redone if the *dummy CLR* is present, even if the transaction performing those updates is a loser transaction. Hence, for the purposes of the optimizations of this section, we can flag all log records written as part of a nested top action as ones requiring to be redone always. Of course, the redos will be performed only as long as the affected pages don't already have those updates. The scenario in Figure 9 illustrates the problem with not redoing a nested top action's updates for an affected page just because after the nested top action that page had no updates which were performed by a nonloser transaction.

In this example, after inserting K1 on P1, the transaction attempts to insert another key on P1 which causes a split of P1 to be performed. After the split is completed, but even before the transaction inserts the new key, the system crashes. If the update to P1 in log record *b* were not redone, then the key K1 will be present on P1 as well as P1' (the new page of the split). Then, the undo of log record *a* will remove K1 only from P1. That is, page-oriented undo of *a* would succeed. K1 would still be present in P1', which would be a violation of the transaction atomicity property. If only the update of log record *b* had been redone, then that would have resulted in K1 being eliminated from P1. After that, the undo of log record *a* would have been performed logically (i.e.,

by traversing the tree) and K1 would have been deleted from P1' as well.

Note that we could get into structural inconsistency problems also, if we do not redo the nested top action related log records. ARIES-RRH functions correctly even if transitions, for a given object, from fine-granularity locking to coarse-granularity locking and vice versa are permitted to happen online.

## 6. Summary

We presented a method, called ARIES-RRH, which is an enhanced version of the ARIES transaction recovery and concurrency control method that has been implemented in a number of systems. The enhancements related to the amount of redo of updates that needs to be performed at the time of system restart in order to bring the data base to a consistent state. They should result in a reduction in the number of I/Os and in the amount of CPU processing during the redo and undo passes of restart. This should improve the availability of the system by allowing processing of new transactions to begin earlier than with ARIES. ARIES-RRH requires that, for each page, *all* updates logged at least up to the point of the most recent committed<sup>2</sup> or *in-doubt* update be redone, if those updates are not already present in the page as determined by comparing the LSN of the page with the LSNs of the log records. In addition, when CLR's are encountered, they would have to be redone, if the LSN of the affected page is *greater than or equal to* the UndoLSN of the CLR and is *less than* the LSN of the CLR. We firmly believe that, with the flexibility offered via operation logging and the support for semantically-rich modes of locking by ARIES and ARIES-RRH, this is the best that can be done. Of course, if only physical logging and locking are supported (as in IMS), the intervening updates of loser transactions need not be redone. ARIES-RRH does not compromise on the ARIES method's properties of never undoing a CLR's updates and never undoing the same nonCLR's updates more than once. We also explained the fundamental reasons behind why certain existing recovery algorithms work correctly in the face of failures during restart recovery or during media recovery. We believe that our work has led to a better understanding of the fundamental interactions between concurrency control and recovery methods.

## 7. References

- BaRa87** Badrinath, B.R., Ramamritham, K. *Semantics-Based Concurrency Control: Beyond Commutativity*, Proc. 3rd IEEE International Conference on Data Engineering, February 1987.
- BeHG87** Bernstein, P., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- ChMy88** Chang, P.Y., Myre, W.W. *OS/2 EE Database Manager Overview and Technical Highlights*, IBM Systems Journal, Vol. 27, No. 2, 1988.

<sup>2</sup> Conceptually, we treat a transaction which terminated after aborting or rolling back completely as a transaction which performed a partial rollback to its beginning and then committed.

- CICo89** Clark, B.E., Corrigan, M.J. *Application System/400 Performance Characteristics*, **IBM Systems Journal**, Vol. 28, No. 3, 1989.
- Crus84** Crus, R. *Data Recovery in IBM Database 2*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- Curt88** Curtis, R. *Informix-Turbo*, **Proc. IEEE Comcon Spring '88**, February-March 1988.
- DGSBH90** DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I., Rasmussen, R. *The Gamma Database Machine Project*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.
- GMBLL81** Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. *The Recovery Manager of the System R Database Manager*, **ACM Computing Surveys**, Vol. 13, No. 2, June 1981.
- Gray78** Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems - An Advanced Course**, R. Bayer, R. Graham, and G. Seegmuller (Eds.), Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.
- HaRe83** Haerder, T., Reuter, A. *Principles of Transaction Oriented Database Recovery - A Taxonomy*, **Computing Surveys**, Vol. 15, No. 4, December 1983.
- HMSC88** Haskin, R., Malachi, Y., Sawdon, W., Chan, G. *Recovery Management in QuickSilver*, **ACM Transactions on Computer Systems**, Vol. 6, No. 1, p82-108, 1988.
- Hsia90** Hsiao, H. *Performance and Availability in Database Machines with Replicated Data*, **PhD Thesis**, Computer Sciences Technical Report #963, University of Wisconsin at Madison, August 1990.
- IBM87** *IMS/VS Extended Recovery Facility (XRF): Technical Reference, Document Number GG24-3153*, IBM, April 1987.
- KGBW90** Kim, W., Garza, J.F., Ballou, N., Woelk, D. *Architecture of the ORION Next-Generation Database System*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.
- KoLS90** Korh, H., Levy, E., Silberschatz, A. *Compensating Transactions: A New Recovery Paradigm*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.
- Kuma90** Kumar, A. *A Crash Recovery Algorithm Based on Multiple Logs that Exploits Parallelism*, **Proc. 2nd IEEE Symposium on Parallel and Distributed Processing**, Dallas, December 1990.
- Lome90** Lomet, D. *Recovery for Shared Disk Systems Using Multiple Redo Logs*, **Technical Report CRL 90/4**, DEC Cambridge Research Laboratory, October 1990.
- MHLPS89** Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, To Appear in **ACM Transactions on Database Systems**. Also Available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989; Revised November 1990.
- Moha90a** Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.
- Moha90b** Mohan, C. *ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators*, **IBM Research Report**, IBM Almaden Research Center, December 1990.
- MoLe89** Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.
- MoLI83** Mohan, C., Lindsay, B. *Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions*, **Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing**, Montreal, Canada, August 1983. Also available as **IBM Research Report RJ3881**, IBM San Jose Research Laboratory, June 1983.
- MoNP90** Mohan, C., Narang, I., Palmer, J. *A Case Study of Problems in Migrating to Distributed Computing: Page Recovery Using Multiple Logs in the Shared Disks Environment*, **IBM Research Report RJ7343**, IBM Almaden Research Center, March 1990.
- PeSt83** Peterson, R.J., Strickland, J.P. *Log Write-Ahead Protocols and IMS/VS Logging*, **Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems**, Atlanta, March 1983.
- Rahm89** Rahm, E. *Recovery Concepts for Data Sharing Systems*, **Technical Report 14/89**, University of Kaiserslautern, October 1989.
- ReSW89** Rengarajan, T.K., Spiro, P., Wright, W. *High Availability Mechanisms of VAX DBMS Software*, **Digital Technical Journal**, No. 8, February 1989.
- RoMo89** Rothermel, K., Mohan, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989. A longer version of this paper is available as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.
- SCFLM86** Schwarz, P., Chang, W., Freytag, J., Lohman, G., McPherson, J., Mohan, C., Pirahesh, H. *Extensibility in the Starburst Database System*, **Proc. Workshop on Object-Oriented Data Base Systems**, Asilomar, September 1986. Also Available as IBM Research Report RJ5311, IBM Almaden Research Center, September 1986.
- Spec89** Spector, A. *Modular Architectures for Distributed and Database Systems*, **Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems**, Philadelphia, March 1989.
- SPSW90** Schek, H., Paul, H.-B., Scholl, M., Weikum, G. *The DASDBS Project: Objectives, Experiences, and Future Prospects*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.
- Tand87** The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987.
- TeGu84** Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- Vura90** Vural, S. *A Simulation Study for the Performance Analysis of the ARIES Transaction Recovery Method*, **M. Sc. Thesis**, Middle East Technical University, February 1990.
- WHBM90** Weikum, G., Hasse, C., Broessler, P., Muth, P. *Multi-Level Recovery*, **Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems**, Nashville, April 1990.
- WHMZ90** Weikum, G., Hasse, C., Moenkeberg, A., Zabback, P. *The COMFORT Project: A Comfortable Way to Better Performance*, **Research Report No. 137**, ETH Zurich, July 1990.

## Appendix A. Logging and Recovery

To meet transaction and data recovery guarantees, the transaction processing system records in a *log* the progress of a transaction, and its actions which cause changes to recoverable data objects. The log becomes the source for ensuring either that the transaction's committed actions are reflected in the data base in spite of

various types of failures, or that its uncommitted actions are undone (i.e., rolled back). The log can be thought of logically as an ever growing sequential file.

Each log record is assigned, by the log manager, a unique **log sequence number (LSN)** at the time the record is written to the log. The LSNs are assigned in ascending sequence. Typically, they are the logical addresses of the corresponding log records. At times, version numbers or timestamps are also used as LSNs [MoNP90]. On finishing the logging of an update to a page, in many systems (e.g., in DB2 [Crus84], Starburst [SCFLM86], QuickSilver [HMSC88], the OS/2 Extended Edition Database Manager [ChMy88] and NonStop SQL<sup>1</sup> [Tand87]), the LSN of the log record corresponding to the *latest* update to the page is placed in a field (**page\_LSN**) in the page header. Hence, knowing the LSN of a page allows the system to correlate the state of the page with respect to those logged updates relating to that page. That is, at the time of recovery, the page LSN and the log record's LSN can be compared to determine unambiguously whether or not that log record's update is already reflected in that page [MHLPS89].

The nonvolatile version of the log is stored on what is generally called **stable storage** (e.g., disk). This storage remains intact and available across system failures. Whenever log records are written, they are placed first only in the *volatile* storage (i.e., virtual storage) buffers of the log file. Only at certain times (e.g., at *prepare* time during the execution of the two-phase commit protocol [MoLi83]) are the log records up to a certain LSN written, in log page sequence, to stable storage. This is called **forcing** the log up to that LSN.

There are two general approaches to recovery: the **write-ahead logging (WAL)** approach [Gray78, MHLPS89] and the **shadow-page technique** [GMBLL81, MHLPS89]. Detailed comparisons between the two methods are given in [MHLPS89]. WAL is the recovery method of choice in most systems, even though the shadow-page technique of System R or a variation of it is used in systems like SQL/DS and Informix-Turbo<sup>1</sup>. In WAL systems, an updated page is written back to the same nonvolatile storage location from which it was read. The **WAL protocol** asserts that the log records representing changes to some data must already be on stable storage **before** the changed data is allowed to replace the previous version of that data on nonvolatile storage. The buffer manager uses the LSN associated with the page to ensure that the log has been forced up to that LSN before it writes the modified page to nonvolatile storage.

Generally, each log record describes the update performed on only a single page. The undo (respectively, redo) portion of a log record provides information on how to undo (respectively, redo) changes performed by the transaction. A log record which contains both the undo and the redo information is called an **undo-redo log record**. Sometimes, a log record may be written to contain only the redo information or only the undo infor-

mation. Such a record is called a **redo-only log record** or **undo-only log record**, respectively. Depending on the action performed, the undo-redo information may be recorded **physically** (e.g., before and after images of specific fields within the object) or **operationally** (e.g., add 5 to field 3 of record 15, subtract 3 from field 4 of record 10). Operation logging permits the supporting of high concurrency lock modes which exploit the semantics of the operation performed on the data. For example, the same field of a record may be permitted to have uncommitted updates of multiple transactions.

**Forward processing** refers to the updates performed when the system is in normal processing and the transaction is making changes to the data base due to the data base calls (e.g., SQL) issued by the user or the application program (i.e., the transaction is not rolling back and using the log to generate the (undo) update calls). **Partial rollback** refers to the ability to set up **savepoints** during the execution of a transaction and later on in the transaction request the rolling back of the changes performed by the transaction since the establishment of a previous savepoint [GMBLL81]. This is to be contrasted with **total rollback** in which *all* the updates of the transaction are undone. **Normal undo** refers to rollback when the system is in normal operation. **Restart undo** refers to rollback when the system is recovering after a failure. Normally, all the log records written by a transaction are linked together (via the **PrevLSN** field) in reverse chronological order. Typically, the updates performed during a rollback are logged using what are called **compensation log records (CLRs)** [MHLPS89]. In ARIES [MHLPS89], CLRs have the property that they are redo-only log records and hence they do not contain any undo information.

**Page-oriented redo** means that, when an operation is redone at recovery time, the log record describes which page of the data base was originally affected during normal processing and the same page is affected during the redo processing also. No internal descriptors of tables or indexes need to be accessed to redo the update. That is, no other page of the data base needs to be examined. This is to be contrasted with **logical redo** which is required for indexes in systems like Informix-Turbo [Curt88] and System R [MoLe89]. In those systems, since index changes are not logged separately but are redone using the log records for the data pages, performing a redo requires accessing a number of descriptors and pages of the data base. The index pages modified as a result of this redo operation may be different from the index pages that were originally modified during normal processing. In a similar fashion, we can define **page-oriented undo** and **logical undo**. Being able to perform logical undos allows the system to provide higher levels of concurrency than what would be possible with page-oriented undo (see [MoLe89, Moha90b]). ARIES supports high efficiency and high concurrency via page-oriented redos and logical undos. It pays the price of logical undo only if page-oriented undo is not feasible.