An Efficient and Flexible Method for Archiving a Data Base

C. Mohan, Inderpal Narang

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
{mohan, narang}@almaden.ibm.com

Abstract We describe an efficient method for supporting incremental and full archiving of data bases (e.g., individual files). Customers archive their data bases quite frequently to minimize the duration of data outage. Because of the growing sizes of data bases and the ever increasing need for high availability of data, the efficiency of the archive copy utility is very important. The method presented here minimizes interferences with concurrent transactions by not acquiring any locks on the data being copied. It significantly reduces disk I/Os by not keeping on data pages any extra tracking information in connection with archiving. These features make the archive copy operation be more efficient in terms of resource consumption compared to other methods. The method is also flexible in that it optionally supports direct copying of data from disks, bypassing the DBMS's buffer pool. This reduces buffer pool pollution and processing overheads, and allows the utility to take advantage of device geometries for efficiently retrieving data. We also describe extensions to the method to accommodate the multisystem shared disks transaction environment. The method tolerates gracefully system failures during the archive copy operation.

1. Introduction

Data availability is a key customer requirement. Data base management systems (DBMSs) provide an archive copy (***AC***) utility to copy (backup) data bases in case recovery is needed. Recovery may be needed after a media failure, such as, a disk crash or application logic error which has corrupted the data. This paper describes an efficient way to copy pages of a data base (e.g., a file containing a relational table's data) which were changed since the time the previous AC operation was executed. Customers execute the AC operation on important data bases almost *daily* (as opposed to weekly or monthly) to reduce the data outage time when a media failure does occur and data needs to be recovered using archive copies of the data and log records. Because of the frequency at which this utility is run and the growing sizes of the data bases [Moha92], the efficiency of the AC utility is an important issue in a highly available computing environment.

A ***full archive copy*** (***FAC***) copies all the pages of the data base. If the copy utility were to be required to copy only those pages of the data base which had been updated since the time the most recent AC was taken, then we call it an ***incremental archive copy*** (***IAC***). Generically, we refer to FAC and IAC executions as ***AC utility executions***. If concurrent updates are permitted when an archive copy is taken, then such an AC is called a ***fuzzy AC***. Many DBMSs, such as DB2™ [IBM], support fuzzy archive copy. Some details of the method employed by DB2 to support incremental, fuzzy archive copies are described in [CPM82]. DB2's method is described briefly as follows.

- To support incremental archive copy, i.e., to track the pages which have been modified since the last AC was run, one bit per data page is allocated in the space map page which covers those data pages. This bit is called the ***archive copy bit*** (***ACB***). The ACB is set by the transaction when it updates a data page for the first time since the last AC was run. In order to reduce the overhead of visiting the space map page on every update of the data page, a flag is also maintained in every data page. The purpose of the data page flag is that if it is already set then the transaction which updates the data page would not visit the space map page; otherwise, it would set the data page flag and access the space map page to set the ACB.

- IAC examines the ACBs to determine the pages which need to be copied, whereas FAC copies all the pages. AC would reset the ACBs and the flag in the copied data pages as it copies each page. Note that a disk write I/O needs to be performed just for resetting the flag on the disk version of every copied data page during the copy operation.[1]

- DB2 supports only page locking as the smallest granularity of locking (i.e., record locking is not supported) and hence there is no concept of latching pages [MHLPS92]. In order to serialize page updates by a transaction with the maintenance of the flag by the copy utility, AC acquires a *lock* on every page as it is accessed. This impacts concurrency and performance since the lock is held across a disk I/O. Also, a wait will be caused if the page has uncommitted data on it. These in turn elongate the duration of the entire archive copy operation.

- If AC's execution is interrupted by a system failure, then the data base is put in a *full-archive-copy-required* state when the system is restarted.

We describe a more efficient method for producing **fuzzy, incremental archive copies** of a data base than what is described in [CPM82]. The method provides significant improvements over the prior art, in terms of locking overhead, disk I/Os and the degree of concurrency supported. In fact, the method eliminates the need for a flag in a data page, and hence the need for a disk I/O for resetting the flag. Also, the method acquires no locks to make a copy of a

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0139...$1.50

™DB2 and IBM and are trademarks of International Business Machines Corp. INFORMIX is a registered trademark of Informix Software, Inc. ORACLE is a trademark of Oracle Corp. Rdb/VMS is a trademark of Digital Equipment Corp.

page, but rather uses a cheaper latch protocol. Our method is applicable to the single system as well as the multisystem shared disks (also called *data sharing*) transaction environment. In a multisystem shared disks environment, the advantages of our method become even more significant. This is because of the following:

- By avoiding acquiring a page lock to make a copy of a page, the overhead of global locking is eliminated. Global locking can have significant overhead because of messages that need to be exchanged between systems [MoNa92a].
- By avoiding the need for a flag in every data page, the coherency overheads for keeping the copies of a page consistent among the different sharing systems' buffer pools are avoided. Coherency overheads involve messages and refresh of a page [MoNa91].

The rest of the paper is organized as follows. First, we describe the assumptions relating to the data structures in the data base and in virtual storage, the logging protocol, and the latching protocol. Then, we describe the processing relating to the copy utility, a transaction's update to set the ACB (if required), and the recovery of a data base using the backup copy. We also describe the processing involved in the rollback of an AC, in case its execution is interrupted by a failure. Third, we describe how the method can be used when it is possible to invoke a utility which can copy the data directly from disk instead of going through the DBMS's buffer pool. Then, we describe the extensions of the copy utility to the multisystem transaction processing system with shared disks. We also compare our method with those in commercial products and in the research literature. Finally, we summarize.

1.1. Assumptions

In this section, first, we describe the data structures in the data base and in virtual storage, and other variables which are needed for the implementation of our method. Then, we describe the logging and latching protocols.

1.1.1. Data Base Data Structures

- A data base consists mostly of the pages of user data (for the purposes of describing our method, without loss of generality, *a file can be equated to a data base*, and no distinction needs to be made between data and index pages). However, the system reserves some pages in the data base for its use for tracking the allocation status and other information relating to the user's data pages. A page containing user data is called a ***data page*** (***DP***), a system-owned page tracking the allocation and space availability status of DPs is called a ***space map page*** (***SMP***), and a system-owned page containing system related information is called a ***header page***.[2] Archive copies include copies of not only DPs but also those of SMPs and the header page.

- An SMP tracks allocation status of several DPs. Also, an SMP contains one ACB for each DP that it covers, as described in [CPM82]. The purpose of the ACB is to track whether the corresponding data page has been modified since the last time the AC utility was run.

- In the header page, a field called ***archive_copy_bit_update_LSN*** (***ACBU_LSN***) is maintained. There is only one ACBU_LSN for a file. The purpose of the ACBU_LSN is that by using it a transaction updating a DP can efficiently determine if the ACB in the SMP for that DP may need to be set. This is desirable because the ACB needs to be set only if it is the first update to a DP since the last AC operation. Accessing the SMP for subsequent updates of a DP must be avoided because of the overheads involved. The purpose and use of ACBU_LSN will become increasingly clearer as we describe the details of the method.

  The initial value of ACBU_LSN is set to the LSN of the current-end-of-log after the data base is initially loaded. A *full* AC of the data base must be taken *before* any update to the data base can be permitted after its initial load, assuming that loading was done without logging. This is required if media recovery must be possible without the load having to be repeated before media recovery begins.

- Associated with each archive copy is an ***archive_copy_roll_forward_LSN*** (***ACRF_LSN***). During media recovery, this is the LSN from which the log would have to be scanned to identify log records whose updates might have to be redone to recover the data base after reloading the relevant archive copies (the latest FAC and any subsequent IACs). ACRF_LSN is remembered in a system catalog along with the information such as the name of the file which contains the archive copy.

1.1.2. Data Structures in Virtual Storage

As long as a data base is in the *open* state, the DBMS's buffer manager maintains a ***data base control block*** (***DBCB***) for it in virtual storage.

To make the operation of looking up ACBU_LSN efficient by update transactions, the value of ACBU_LSN is copied from the header page into a field in DBCB when a data base is opened. Subsequently, this field is updated only by the AC utility.

1.1.3. Logging

The DBMS uses a log for describing its updates for recovery purposes. When a page is updated, the DBMS assigns a system-wide monotonically increasing log sequence number (***LSN***) to the log record describing the update and records that LSN in the modified page in a field called ***page_LSN*** [MHLPS92].

We assume that AC runs as a single transaction and writes undo-redo log records whenever it resets ACBs.

1.1.4. Page Latching

The DBMS's buffer manager supports page latches to serialize concurrent readers and updaters, or multiple concurrent updaters of a single page. A share (S) latch is used

---

1 In DB2, the disk write I/O is performed *immediately* after resetting the bit to do "immediate commit". This is a significant overhead since an I/O is done for each page *individually* (i.e., there is no batching of writes as is done for normal dirty pages [TeGu84]).

2 As in DB2, a large table can be divided into many partitions each of which is a separate operating system file [TeGu84]. Each partition would have a header page, one or more space-map pages and numerous data pages.

by a reader and an exclusive (X) latch is used by an updater, with the usual compatibility rules between the S and X modes. A page latch would be used by AC to make a fuzzy copy of a page, i.e., the copied page may have uncommitted data.

2. Correctness Criteria

The correctness requirement for an IAC is that the current IAC must capture the effects of all data page updates which are represented by log records written prior to the current IAC's ACRF_LSN point in the log except for those updates whose effects have already been captured in the most recent FAC, or in an IAC made subsequent to that FAC and prior to the current IAC.

The correctness requirement for a FAC is that the current FAC must capture the effects of all data page updates which are represented by log records written prior to the current FAC's ACRF_LSN point in the log.

3. AC Algorithms - Single System DBMS

First, we describe the copy utility algorithm for a DBMS in a single system environment where the copy utility brings data base pages into the DBMS buffer pool. Then, we describe the algorithm that is to be used when the data is read directly from disk, i.e., the pages are not brought into the DBMS buffer pool. In the next section, we describe the algorithm which would be used for a DBMS in a multisystem shared disks environment.

3.1. Copying Data via DBMS's Buffer Pool

3.1.1. Logic for the Archive Utility

We describe the algorithm which consists of the logic for a) the copy utility, b) update of a data page by a transaction and, c) the recovery of the data base after a media failure. We also describe the rollback processing in case AC's execution is interrupted by a failure.

AC uses the following logic while copying the pages. Note that user transactions could be updating (as well as reading) the pages of this data base while AC is copying the pages.

- Register the fact that AC has started for this data base in the system catalog to ensure that only one copy utility is running against the same data base at one time.

- Open the data base if it is not already open. This would cause DBCB to be created. Read the header page of the data base to set the value of ACBU_LSN in DBCB. If the data base was already open, then ACBU_LSN in DBCB would have already been set.

- Make sure that all the SMPs of this data base are present in the buffer pool. For those SMPs that aren't already in the buffer pool, of course, this would involve initiating I/Os to bring them into the buffer pool and waiting for those I/Os complete.

  This prefetching of SMPs is just an optimization to minimize the time period during which the ACBU_LSN field in DBCB has the maximum possible value (X'FF .. FF' - see the next step).

- Change the value of ACBU_LSN in DBCB to X'FF..FF' (i.e., the maximum possible value) atomically.

  The purpose of setting ACBU_LSN to the maximum value would become clear when we describe the logic for a transaction updating a page in section "3.1.1.1. Logic for a Transaction Updating a Page". It is also discussed further in the section "Appendix A. Reason for Setting ACBU_LSN to X'FF..FF' Initially".

- For use during media recovery, record the LSN of the current-end-of-log as the ACRF_LSN in the appropriate system catalog and log that action.

- For each SMP in the data base, do the following:

  - X latch the SMP.
  - Examine all the ACBs in the SMP. Reset those ACBs that have a value of '1' to '0'. For an IAC, the corresponding DPs are the ones that must have been modified since the last archive copy (full or incremental) and would be copied. For a FAC, all DPs would be copied.
  - Write a *redo-undo* log record that describes all the ACBs that were reset on this SMP. For an IAC, keep a copy of the log record in virtual storage for use later on to know which DPs to copy.
  - Unlatch the SMP.

  Note that *after* the ACRF_LSN value is picked, if new SMPs get added to the data base, they do not have to be examined as part of this AC operation.

  Note that delaying the copying of data pages until after all the SMPs have been examined and modified is an optimization to minimize the time period during which the ACBU_LSN field in DBCB has the value X'FF .. FF'.

- Get the current-end-of-log LSN. Update the ACBU_LSN fields in DBCB and in the header page to that value. Log the update to the header page using a *redo-only* log record. Note that even if the AC transaction were to rollback, it would be correct to retain the new ACBU_LSN instead of reverting back to the old ACBU_LSN. This is the reason we are writing a redo-only log record, as opposed to a redo-undo log record (this approach avoids some special case handling of undos of header page updates in the shared disks environment).

- For each SMP in the data base do the following:

  For an IAC, by examining the cached version of the earlier-written log record for that SMP, determine the ACBs that were reset (to '0') by the utility (ACBi relates to DPi). For each such ACBi, do the following:

  - S latch DPi.
  - Copy DPi into the archive copy file.
  - Unlatch DPi.

  For a FAC, all DPs covered by this SMP are copied.

  To make these operations very efficient, and to reduce the I/O costs and delays, batched I/Os (i.e., reading more than one page in a single I/O call) and prefetching of desired data pages can be employed [TeGu84]. Parallelism also can be employed to further reduce the elapsed time for taking the archive copy.

- At the end of the AC operation (i.e., after all the relevant data pages have been copied), do the following:

- Note the completion of this AC so that, once this AC transaction commits, a subsequent invocation of the AC utility for this data base can begin execution without being delayed.
- Commit the AC transaction.

If no concurrent updates are allowed when AC is running, the logic would be a subset of what is described above. The utility would get an S lock on the data base to disallow concurrent updates (the lock held by an update transaction has to be in IX, SIX, or X mode, all of which are incompatible with S). The key differences are:

- The ACRF_LSN value would be picked *after* all the required data base pages have been copied in order to reduce the amount of log that needs to be processed during a subsequent media recovery. This is sufficient because no pages of this data base would be updated while the copy utility is running.
- ACBU_LSN can be set to the same value as ACRF_LSN.
- Page latching is not needed when the utility copies a data page or modifies the SMP.

3.1.1.1. Logic for a Transaction Updating a Page

An update transaction, while updating a DP, has to determine whether the ACB for that DP needs to be set ('1') in the SMP. It does this by comparing the current (i.e., prior to this update) page_LSN of the page being updated with the ACBU_LSN (end-of-log LSN when the previous execution of the copy utility completed execution). This check determines whether the data page has already been modified since the last execution of the copy utility or this is the first update since that time. The logic followed by an update transaction is:

- X latch DP.
- If current page_LSN < ACBU_LSN in DBCB then do the following:
  - Access SMP which covers this DP and S latch it.

    Since soon after the copy utility starts its execution it sets ACBU_LSN to X'FF..FF', it causes every update to this data base to visit the relevant SMP while the utility is resetting ACBs in all the appropriate SMPs.

  - If ACB is not already set, then set it and log that action using a **redo-only** log record. Multiple transactions could be updating the SMP concurrently since only an S latch is acquired on the SMP. S latching is done to improve concurrency on such hot-spot pages. As a result, updates to the ACB as well as the page_LSN field must be done carefully (e.g., don't update the page_LSN field if it has already been updated to a higher value by another transaction - compare and swap logic should be used).

    The ACB could be *already* set in the following case: DP was updated one or more times while the copy utility was running with ACBU_LSN in DBCB set to X'FF..FF'. One possible way to avoid visiting the SMP multiple times during such times is to have a flag in the buffer manager's ***page control block*** (***PCB***) for that page which could indicate, while ACBU_LSN is X'FF....FF', that the DP's ACB has already been set. AC resets that flag while holding the latch on DP. That flag helps only if DP remains cached from one update to another, i.e., the buffer slot for the page is not stolen. If the page gets replaced and is cached again, the PCB flag is initialized such that it implies that the ACB has not been set.

    The reason for writing the log record as a *redo-only* record for the SMP update is that even if the transaction which updated the DP were to rollback, the ACB value must remain set (='1') in the SMP. This is because a subsequent update to the same DP by another transaction would presume that the ACB had been set because the page_LSN would have been set to a value greater than ACBU_LSN by the first updater (see below).

  - Unlatch SMP.
- Update DP, log the update and set DP's page_LSN to the LSN of the log record just written.
- Unlatch DP.

For a newly allocated page, ACB is always set ('1').

Note that if the ACB needs to be set, then it is important that the logging of the corresponding DP's update be done **after** the ACB is set and the ACB setting operation is logged. The opposite order could cause incorrect media recovery. Incorrect recovery could be caused by a scenario like the following:

- LSN 50: T1 updates DP10 and logs that update.
- ...
- LSN 100: IAC begins and sets ACRF_LSN to 100. Since ACB for DP10 is not set, IAC decides that DP10 has not been updated since the last AC and hence does not copy DP10's contents.
- ...
- LSN 200: IAC ends and sets ACBU_LSN to 200.
- LSN 250: T1 sets ACB for DP10 and logs that update.
- LSN 280: T1 commits.
- LSN 350: Media failure occurs, thereby necessitating media recovery (see section "3.1.1.3. Recovery After Media Failure"). The recovery utility reloads the data from the most recent FAC and all the subsequent IACs of the data. Redo processing is begun from ACRF_LSN of the most recent AC, namely LSN 100. As a result, the update represented by log record with LSN 50 is not redone. This is incorrect since that is a committed update and its effect is not present in any of the ACs.

3.1.1.2. Failure During AC's Execution

In case AC's execution is interrupted due to some failure, the normal rollback logic, using the log records written by AC, will restore the old settings of those ACBs whose settings were modified by AC. It is important that no *other* ACBs' values be modified by the rollback logic to contain their original values (i.e., no ACBs are assigned '0's during rollback). Also, the information in the system catalog about the current execution of AC (e.g., ACRF_LSN value) would be deleted.

3.1.1.3. Recovery After Media Failure

The logic for media recovery is very similar to that in the prior art [CPM82, MHLPS92]. The log scan is started from the ACRF_LSN value of the last completed copy utility. This scan is begun after the data is copied from the most recent FAC and from all the subsequent IACs in time sequence.

3.2. *Bypassing DBMS's Buffer Pool*

In [MHLPS92], we presented a method for doing FAC which permitted the utility to copy the pages of the data base directly from the data base disk to the archive copy file. It avoided the expenses (e.g., space management, latching) associated with reading the pages through the DBMS's buffer manager and then copying them. That method is also better since it can take advantage of information about device geometry to make the copying operation be much more efficient. It also does not *pollute* the buffer pool with pages of the data base being copied, thereby not reducing the buffer hit ratio for user transactions and queries. It does not cause pages belonging to other data bases from being replaced in the buffer pool.

Unfortunately, that method did not permit IACs. Here, we describe a method which satisfies both requirements. The steps to be followed during a FAC are:

- Register the fact that FAC has started for this data base in the system catalog.
- Open the data base if it is not already open and read the header page of the data base to set the value of ACBU_LSN in DBCB.
- Make sure that all the SMPs of this data base are present in the buffer pool.
- Change the value of ACBU_LSN in DBCB to X'FF..FF' atomically.
- Record the LSN of the current-end-of-log as the ACRF_LSN in the appropriate system catalog and log that action.
- For each SMP in the data base, do the following:
  - X latch the SMP.
  - Examine all the ACBs in the SMP. Reset those ACBs that have a value of '1' to '0'.
  - Write a *redo-undo* log record that describes all the ACBs that were reset on this SMP.
  - Unlatch the SMP.
- Get the current-end-of-log LSN. Update the ACBU_LSN fields in DBCB and in the header page to that value. Log the update to the header page using a redo-only log record.
- Initiate I/Os to write back to disk the *dirty* (i.e., updated) pages of the data base that are currently in the buffer pool. Only those pages that were already dirty need to be written back. Since updates by transactions are not being quiesced, if *other* pages become dirty after the resetting of ACBs is complete, they do not have to be written back to disk as part of this FAC operation.

  These write I/Os are necessary to ensure that the correctness criterion discussed earlier is satisfied.
- Wait for all the I/Os initiated in the last step to complete.
- Copy the data base directly from the disk(s) to the FAC file using the most efficient means.
- At the end of the FAC operation (i.e., after all the data pages have been copied), do the following:
  - Note the completion of this FAC.
  - Commit the FAC transaction.

The assumption that is necessary to ensure that the above method works correctly is that the operating system software will not permit interleaving of reads and writes of the disk version of the data base in such a way that the effects of a partial write of a page are read.

It should be easy to see that direct copying from disk to the archive file can be done even for IACs, by following the previously specified steps (for an IAC and a disk-to-archive-file FAC) with a few changes:

- After identifying the pages to be copied into the IAC by going through the SMPs, if any of those pages are currently in the dirty state in the buffer pool, then write those dirty pages back to disk.
- If any dirty pages had to be written to disk, then wait for those write I/Os to complete. Once they complete, direct disk to archive file copying of the desired pages can be performed.

4. AC Utility in Multisystem DBMS with Shared Disks

First, we describe the functions which are relevant to this paper for a DBMS to operate in a multisystem environment with shared disks [MoNa91, MoNa92a]. The DBMS software residing on different processors can concurrently read and modify the same data base which is stored on disks connected to all these systems, i.e., the data base is said to be stored on shared disks. Each DBMS instance has its own buffer pool where it caches data base pages. A multisystem DBMS requires functions such as, message passing between systems, global locking, coherency of its cached data, assignment of complex-wide unique LSNs for the log records, and merging of log record streams produced by the individual systems. In this paper, we assume that there is a global locking function to coordinate locks on data base pages, records, etc.; a page invalidation mechanism for DBMS cache (buffer pool) coherency [MoNa91]; local LSN assignment by each of the systems as described in [MoNa92b];[3] each system has its own local log file(s), but there is a facility to merge those logs on the basis of the log records' LSNs to support media recovery and, possibly, restart recovery [MoNa91, MoNa92b]. Many commercial DBMSs support the shared disks environment. Example systems are IMS, Oracle™ and Rdb/VMS™, and DBMSs from Hitachi and Fujitsu.

The two major challenges in developing algorithms for the copy utility in a multisystem environment, when transactions are allowed to update the data base concurrently with the copy utility, are the following:

3 That method assures properties like the following: (1) the LSNs assigned by a system will monotonically increase as new log records are written; (2) if the same page were to be updated over time by different systems, then the LSNs assigned to the corresponding log records will monotonically increase; (3) if necessary, periodically all the systems will be logically synchronized to make their current end of log LSNs be close together.

1. Minimizing the overheads involved.

   Those overheads are, for example, lock contention on hot-spot pages such as SMPs (called the overhead of global locking) and transfer of current copy of a page from one system to another (called the overhead of coherency).

   Our AC method avoids acquiring a physical lock on a page to copy the page's contents. This saves locking and I/O overheads.

2. Dealing with the fact that some systems which had earlier updated the data base pages of interest might have failed before they externalized those pages.

4.1. Logic for the Archive Utility

- Register the start of the AC utility.

- Open the data base if it is not already open and read the header page of the data base to set the value of ACBU_LSN in DBCB.

- Make sure that all the SMPs of this data base are present in the buffer pool. In this process, acquire update locks (*physical*) on all the SMPs of interest [MoNa91]. This may require first having to recover those SMPs on which some other currently down (failed) system(s) has update locks by redoing their missing updates [MoNa91].

- Change the ACBU_LSN in DBCB to X'FF..FF' atomically.

- Send a message to inform each of the other *operational* systems where this data base is currently open to make it change ACBU_LSN in its version of DBCB to X'FF....FF'. Wait for acknowledgements from those systems. It is assumed that if subsequently some other system were to open that data base for update access, then that system would notice in the catalog entry for the data base that an archive copy operation is in progress and hence would initialize DBCB with X'FF....FF' rather than with the value found in the header page.

- Pick the ACRF_LSN value as the minimum of (LSNs of current-end-of-log of all currently *operational* systems in the data sharing complex). The assumption in the case of the currently down systems is that when they recover the first LSNs that they assign will be at least as high as the highest so far assigned by all the other systems (operational and down) in the complex. Record ACRF_LSN in the appropriate system catalog and log that action.

- For each SMP in the data base, do the following:
  - X latch the SMP and if the previously acquired update (physical) lock on it is not held now, then reacquire it. Depending on the specific cache coherency method in use (see [MoNa91]), take the appropriate steps to ensure that the latest version of the SMP is in the buffer pool before executing the next step (we do not repeat this point later at other such stages in the logic).
  - Examine all the ACBs in the SMP; reset those ACBs that have a value of '1' to '0'.
  - Write a *redo-undo* log record that describes all the ACBs that were reset on this SMP. For an IAC, keep a copy of the log record in virtual storage for use later on to know which DPs to copy.
  - Unlatch the SMP. Depending on the specific cache coherency method in use, immediately or at some later point in time, take the appropriate steps regarding the externalization of the SMP and the releasing of the update lock on the page (we do not repeat this point later at other such stages in the logic).

- Get the maximum value of the current-end-of-log LSNs of all the systems in the complex. Update the ACBU_LSN fields in the local DBCB and in the header page to that maximum value. Log the update to the header page using a redo-only log record and externalize the header page. The immediate externalization of the header page is done so that if another system (which currently does not have the data base open) were to open the data base, then it would see the new ACBU_LSN. Broadcast the new value of ACBU_LSN to the other systems which currently have that data base open so that they update their DBCBs. Update the AC-utility-in-progress catalog information so that systems which open the data base in the future will initialize their DBCB to the ACBU_LSN value in the header page rather than to X'FF....FF'.

- Broadcast a message to each of the other currently operational systems which currently has that data base open requesting it to externalize all dirty pages belonging to that data base which are in its buffer pool. Those systems acknowledge that message after they have finished those writes of dirty pages. Wait for acknowledgements to be received from all those systems.

  The rationale for this step will be explained shortly.

- If any of the currently down systems holds update locks on any of the pages belonging to that data base, then recover those pages by redoing all their missing updates.

- For each SMP in the data base do the following:

  For an IAC, by examining the cached version of the earlier-written log record for that SMP, determine the ACBs that were reset earlier by the utility. For each such ACBi, do the following:

  - S latch DPi.

    Note that since IAC does not acquire a physical lock on the page, another system could be concurrently creating or already created an updated version of the same page. What is important for correctness is that the AC should capture that version of the page that is at least as up to date in the log as of the time when the log record resetting that page's ACB was written. It was to accomplish this that the utility made the other systems externalize their dirty pages **after** the utility wrote all the log records describing the SMPs' ACB resetting operations.

  - Copy DPi into the archive copy file.

  - Unlatch DPi.

  For a FAC, all DPs are copied by the same method.

- At the end of the AC operation (i.e., after all the relevant data pages have been copied), do the following:
  - Note the completion of this AC.
  - Commit the AC transaction.

4.2. Failure During AC Execution

In case AC's execution were to fail, due to a process (task) failure or the failure of the executing system, when the ACBU_LSN value in DBCB was X'FF..FF', then, from a performance standpoint, it is preferable to change the ACBU_LSN value in the DBCBs in the surviving systems which have that data base open to the old ACBU_LSN value in the header page. This step is necessary to reduce the number of visits to the SMPs by the surviving systems. To achieve this effect, a message with the old value can be sent to the other systems either by the system doing the AC in case that system did not fail or by a backup system which is nominated for this purpose in case the system doing the AC fails.

4.3. Logic for a Transaction Updating a Page

It is the same as the one for the single system DBMS case except for having to get a global page lock to serialize concurrent updates to the same page by different systems.

- Get X latch and update (physical) lock on DP.
- If current page_LSN < ACBU_LSN, then access the corresponding SMP and if the corresponding ACB is not already set, then do the following:
  - Get S latch and update lock on SMP.
  - Set the corresponding ACB and log that action using a *redo-only* log record.
  - Unlatch SMP.
- Update DP, log the update and set DP's page_LSN to the LSN of the log record just written.
- Unlatch DP.

4.4. Recovery After Media Failure

The only change to the logic for media recovery from the single system case is that logs from multiple systems where the data base has been updated past the ACRF_LSN have to be merged.

5. Related Work

5.1. DB2's Method

Our method achieves improvements over the DB2 method because of the following:

- It avoids maintenance of a flag in the header of every page of the data base. In DB2, this flag is set at the time the page is updated the first time after the copy utility was run and is reset at the time the page is copied by the copy utility. A disk write I/O needs to be performed just for resetting the flag on the disk version of the page during the copy operation.[4]
- It uses page-latching rather than (transaction) page locking to make a copy of the page. A latch is a very efficient short-term *physical* serialization mechanism and is discussed more in [MHLPS92]. Though page locking guarantees that the copied page does not have any uncommitted data, it is not a requirement for fuzzy copy. Page latch/unlatch operations are much cheaper in pathlength than page lock/unlock operations. Also, note that acquiring a page lock as every page is accessed impacts concurrency and performance since the lock is held across a disk I/O. Also, a wait will be caused if the page has uncommitted data on it. These in turn elongate the duration of the entire archive copy operation.[5] Our method avoids these problems by not using a flag on the page and by only latching a page while making a copy of its contents.
- In DB2, if an IAC's execution were to fail before its completion, then the data base is put in a *full-archive-copy-required* state! Our method does not place this undesirable requirement since we execute IAC as a single transaction and log the changes to the bits which are maintained to support IACs.
- DB2 writes one log record every time the archive copy utility copies a data page and updates that data page's corresponding archive copy bit in the space map page. Our method reduces the logging overhead dramatically by writing only one log record for all the data pages covered by a particular space map page. All the archive copy bits are reset in one access to a space map page.

5.2. INFORMIX's Method

The only other implemented method about which we have enough information is the one used in INFORMIX™ OnLine [Info91]. That method also supports fuzzy IAC and FAC. Unlike the other methods, the Informix method does not use bits like the ACBs in any pages. Instead, it relies on logical timestamps (which are similar to LSNs) which are stored on all the page headers to detect which pages have changed since the time of the last AC. While this means that AC does not modify any fields in any of the copied data base's pages, the major disadvantage is that, even for an IAC, *all* the pages of the data base must be brought into the buffer pool and the current timestamp of each page compared with the timestamp of the last AC to determine which pages must be copied. Actually, the Informix method is much more complicated since the AC is made to be *action consistent*. That is, the AC state is the state of the relevant pages as of the time of the checkpoint that is taken at the beginning of the AC (***AC_Begin Checkpoint***). Like System R and SQL/DS, Informix also quiesces all update activity during a checkpoint to make the checkpoint state of all the data be action consistent [GMBLL81, MHLPS92]. But unlike System R and SQL/DS, Informix does shadowing using a *physical* log. This means that, whenever a page is modified for the first time after a checkpoint, a copy of the old version of the page is logged in the physical log. Hence, for those pages that have been modified since the start of

---

4 In DB2, the disk write I/O is performed *immediately* after resetting the bit. This is a significant overhead since an I/O is done for each page *individually* (i.e., there is no batching of writes as is normally done for dirty pages).

5 It might have been possible to write a log record for resetting the flag. Even then the page lock is held across the writing of the log record. The writing of the log record would increase the log volume. Though it is possible to acquire the page lock in Share mode when copying the page, the duration of the lock is equal to the time to do disk I/O in DB2.

AC, Informix's AC utility retrieves their AC_Begin Checkpoint state from the log records of the physical log and checks to see if they are candidates for being included in the IAC. Presumably because of all this complicated processing, Informix does not support direct copying from disk to the archive file.

5.3. *Other Methods*

An algorithm for archiving is also discussed in [MHLPS92], but that method does not support incremental archiving. That is, in that method all the pages of the data base are copied instead of only those pages that had been modified since the last archive copy. The requirement for supporting incremental archive copying necessitated major changes to even the algorithm for full archive copying.

6. Summary

We have described an efficient and flexible method for supporting incremental and full archive copying of data bases. Archiving data bases is one of the most frequently performed administrative tasks. It is done to minimize the duration of data outage. Because of the growing sizes of data bases and the ever increasing need for high availability of data, the efficiency of the archive copy utility is very important. The method described in this paper is very efficient since it minimizes interferences with concurrent transactions by not acquiring any locks on the data being copied and, eliminates any extra tracking information for archive copy from data pages which results in significant reduction in the number of disk I/Os. These features make the archive copy operation be more efficient in terms of resource consumption compared to the other methods. The method is flexible in that it optionally supports direct copying of data from disks which reduces buffer pool pollution and processing overheads. Such copying also allows the archive utility to take advantage of device geometries to improve efficiency. We also described extensions to the method to accommodate the multisystem shared disks transaction environment. The method tolerates gracefully system failures during the archive operation. Our method has been implemented in a forthcoming product.

Acknowledgements We would like to convey our thanks to Norm Pass for his enthusiasm and encouragement, and to Wayne Hineman and Bob Rees for their questions and for implementing our method.

7. References

CPM82 Crus, R., Putzolu, F., Mortenson, J. *Incremental Data Base Log Image Copy*, **IBM Technical Disclosure Bulletin**, Vol. 25, No. 7B, p3730-3732, December 1982.

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.

IBM *IBM Database 2 Version 2 Command and Utility Reference*, **Document Number SC26-4378**, IBM Corp.

Info91 *INFORMIX OnLine Administrator's Guide - Database Server Version 5.0*, **Part No. 000-7106**, Informix Software, Inc., December 1991.

MHLPS92 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*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992.

Moha92 Mohan, C. *Supporting Very Large Tables*, **Proc. 7th Brazilian Symposium on Database Systems**, Porto Alegre, May 1992. Also available as **IBM Research Report RJ8687**, IBM Almaden Research Center, March 1992.

Moha93 Mohan, C. *IBM's Relational DBMS Products: Features and Technologies*, **Proc. ACM SIGMOD International Conference on Management of Data**, Washington, D.C., May 1993.

MoNa91 Mohan, C., Narang, I. *Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment*, **Proc. 17th International Conference on Very Large Data Bases**, Barcelona, September 1991. A longer version of this paper is available as **IBM Research Report RJ8017**, IBM Almaden Research Center, March 1991.

MoNa92a Mohan, C., Narang, I. *Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment*, **Proc. International Conference on Extending Data Base Technology**, Vienna, March 1992.

MoNa92b Mohan, C., Narang, I. *Data Base Recovery in Shared Disks and Client-Server Architectures*, **Proc. 12th International Conference on Distributed Computing Systems**, Yokohama, June 1992.

TeGu84 Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.

Appendix A. Reason for Setting ACBU_LSN to X'FF..FF' Initially

To describe the rationale for setting the ACBU_LSN value in DBCB to X'FF..FF' initially, we consider the scenario where we reset the ACBs while keeping the ACBU_LSN value in DBCB as it was before AC started execution (i.e., the field continues to contain the *old* value). Let us further consider a particular page and examine its page_LSN before a concurrent transaction updates it. There are two possibilities:

- Page_LSN < ACBU_LSN - This is not a problem case because the SMP would be visited by the transaction and the page's ACB would be set (to '1'). Hence, a subsequent IAC would copy this page in case the current one does not copy it.

- Page_LSN > ACBU_LSN - This may be a problem case if the current AC does *not* copy the page *after* the current transaction updates it. Then, a subsequent IAC will not copy the page if no *other* update were to be performed to the page before that subsequent IAC begins. The copying will not happen because the current transaction would not visit the SMP (since page_LSN is not less than ACBU_LSN) and set the ACB. The subsequent IAC, which does not copy the page, will pick its ACRF_LSN such that the current transaction's update log record would be before that point in the log (i.e., log record's LSN would be less than ACRF_LSN). If subsequent to the successful completion of that IAC, a media failure were to happen and media recovery were to be performed using that subsequent IAC's copy, then the restored data base will not contain the current transaction's update. Clearly, this does not meet the correctness criterion and causes a data integrity problem.

  Hence, the need for setting ACBU_LSN to X'FF..FF' prior to setting the ACBs to zero. In the above scenario, that X'FF..FF' value would have forced the current transaction to set the ACB in SMP.