Notes on Database System Reliability*

M. Tamer Özsu

We have so far assumed that no system failures occur. Within this context, concurrency control algorithms enforce the isolation property as well as database consistency. The no-failure assumption is, of course, unrealistic; there will be various failures in any computer system. Therefore appropriate protocols have to be implemented such that DBMS reliability can be guaranteed.

DBMS reliability protocols address transaction atomicity and durability. As we discussed, atomicity refers to the "all-or-nothing" property of transactions. Consequently, we need to guarantee that, in the face of failures, either the effects of all of the actions of a transaction are reflected in the database, or none of the effects are. For example, if we are running a transaction that increases the salaries of employees by 10%, and a failure occurs during its execution, the DBMS has to take special care to ensure that the database reflects the increase to the salaries of either all the employees or to none. Durability, on the other hand, requires that the effects of *committed* transactions on the database survive failures. This also requires special care, since failures may wipe out some of these updates.

1 Architectural Considerations

The responsibility for maintaining atomicity and durability of transactions falls on the DBMS Recovery Manager (RM). Applications interface with RM interface using the (abstract) commands: `begin_transaction`, `read`, `write`, `commit`, and `abort`[1]; RM also has an interface to the operating system via the `recover` command.

For ease of exposition, we will assume the system architecture given in Figure 1. This is similar to that given in [HR83] and [BHG87].

In this discussion we assume that the database is stored permanently on secondary storage, which in this context is called the *stable storage* [LS76]. The stability of this storage medium is due to its robustness to failures. A stable storage device would experience considerably less-frequent failures than would a nonstable storage device. In today's technology, stable storage is typically implemented by means of duplexed magnetic disks which store duplicate copies

*Much of this material comes from Chapter 12 of *Principles of Distributed Database Systems, 2nd edition*, M.T. Özsu and P. Valduriez, Prentice-Hall, 1999.

[1]Note that these are abstract operations that do not appear in this form within SQL. However, they have SQL correspondences.

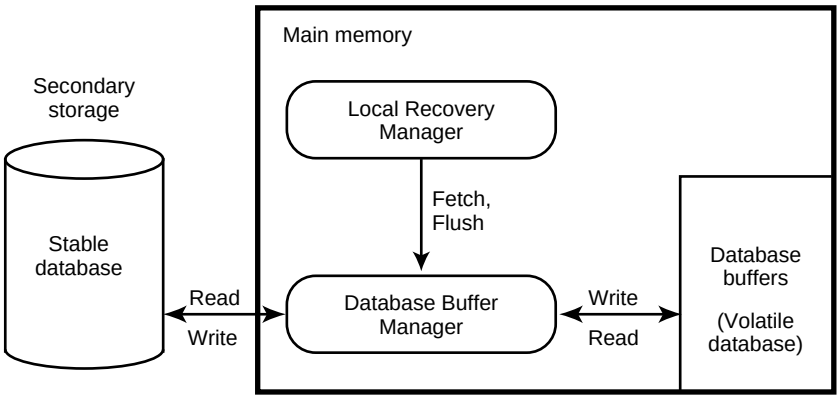

Figure 1: Recovery Architecture

of data that are always kept mutually consistent. We call the version of the database that is kept on stable storage the *stable database*. The unit of storage and access of the stable database is typically a *page*.

The database buffer manager (BM) keeps some of the recently accessed data in main memory buffers. This is done to enhance access performance. Typically, the buffer is divided into pages that are of the same size as the stable database pages. The part of the database that is in the database buffer is called the *volatile database*. Transactions update data that is on the volatile database, which, at a later time, is written back to the stable database.

When the RM wants to read a page of data[2] on behalf of a transaction—strictly speaking, on behalf of some operation of a transaction—it issues a **fetch** command, indicating the page that it wants to read. The buffer manager checks to see if that page is already in the buffer (due to a previous fetch command from another transaction) and if so, makes it available for that transaction; if not, it reads the page from the stable database into an empty database buffer. If no empty buffers exist, it selects one of the buffer pages to write back to stable storage and reads the requested stable database page into that buffer. There are a number of different algorithms by which the buffer manager may choose the buffer page to be replaced (such as the Least Recently Used -LRU) that you may have seen in operating system textbooks.

The BM also provides the interface by which the RM can actually force it to write back some of the buffer pages. This can be accomplished by means of the **flush** command, which specifies the buffer pages that the RM wants to be written back. We should indicate that different RM implementations may or may not use this forced writing. This issue is discussed further in subsequent sections.

As its interface suggests, the buffer manager acts as a conduit for all access

[2]The RM's unit of access may be in blocks which have sizes different from a page. However, for simplicity, we assume that the unit of access is the same.

to the database via the buffers that it manages. It provides this function by fulfilling three tasks:

1. *Searching* the buffer pool for a given page;
2. If it is not found in the buffer, *allocating* a free buffer page and *loading* the buffer page with a data page that is brought in from secondary storage;
3. If no free buffer pages are available, choosing a buffer page for *replacement*.

Searching is quite straightforward. Typically, the buffer pages are shared among the transactions that execute against the database, so search is global.

Allocation of buffer pages is typically done dynamically. This means that the allocation of buffer pages to processes is performed as processes execute. The buffer manager tries to calculate the number of buffer pages needed to run the process efficiently and attempts to allocate that number of pages. The best known dynamic allocation method is the *working-set algorithm* [Den68, Den80].

A second aspect of allocation is fetching data pages. The most common technique is *demand paging*, where data pages are brought into the buffer as they are referenced. However, a number of operating systems prefetch a group of data pages that are in close physical proximity to the data page referenced. Buffer managers choose this route if they detect sequential access to a file.

In replacing buffer pages, the best known technique is the least recently used (LRU) algorithm that attempts to determine the *logical reference strings* [EH84] of processes to buffer pages and to replace the page that has not been referenced for an extended period. The anticipation here is that if a buffer page has not been referenced for a long time, it probably will not be referenced in the near future.

Clearly, these functions are similar to those performed by operating system (OS) buffer managers. However, quite frequently, DBMSs bypass OS buffer managers and manage disks and main memory buffers themselves due to a number of problems (see, e.g., [Sto81]) that are outside the scope of this course.

2 Failure Types in DBMSs

A RM has to deal with three types of failures: transaction failures (aborts), system failures, and media (disk) failures.

2.1 Transaction Failures

Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by incorrect input data as well as the detection of a present or potential deadlock. Furthermore, some concurrency control algorithms do not permit a transaction to proceed or even to wait if the data that they attempt to access are currently being accessed by another transaction. This might also be considered a failure. The usual approach to take in cases of

transaction failure is to *abort* the transaction, thus resetting the database to its state prior to the start of this transaction.

The frequency of transaction failures is not easy to measure. It is indicated that in System R, 3% of the transactions abort abnormally [GMB+81]. In general, it can be stated that (1) within a single application, the ratio of transactions that abort themselves is rather constant, being a function of the incorrect data, the available semantic data control features, and so on; and (2) the number of transaction aborts by the DBMS due to concurrency control considerations (mainly deadlocks) is dependent on the level of concurrency (i.e., number of concurrent transactions), the interference of the concurrent applications, the granularity of locks, and so on.

2.2 System) Failures

The reasons for system failure can be traced back to a hardware failure (processor, main memory, power supply, etc.) or to a software failure (bug in the operating system or in the DBMS code). The important point from the perspective of this discussion is that a system failure is always assumed to result in the loss of main memory contents. Therefore, any part of the database that was in main memory buffers is lost as a result of a system failure. However, the database that is stored in secondary storage is assumed to be safe and correct.

2.3 Media Failures

Media failure refers to the failures of the secondary storage devices that store the database. Such failures may be due to operating system errors, as well as to hardware faults such as head crashes or controller failures. The important point from the perspective of DBMS reliability is that all or part of the database that is on the secondary storage is considered to be destroyed and inaccessible.

Duplexing of disk storage and maintaining archival copies of the database are common techniques that deal with this sort of catastrophic problem.

3 Update Approaches

There are two possible update approaches in DBMSs: in-place updating and out-of-place updating. *In-place updating* physically changes the value of the data item in the stable database. As a result, the previous values are lost. *Out-of-place updating* on the other hand, does not change the value of the data item in the stable database, but maintains the new value separately. Of course, periodically, these updated values have to be integrated into the stable database. Reliability issues are somewhat simpler if in-place updating is not used. However, most DBMSs use it due to its better performance.

3.1 In-Place Update

Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the database state changes to facilitate the recovery of the database to a consistent state following a failure. This information is typically maintained in a database log. Thus each update transaction not only changes the database but is also recorded in the *database log* (Figure 2).

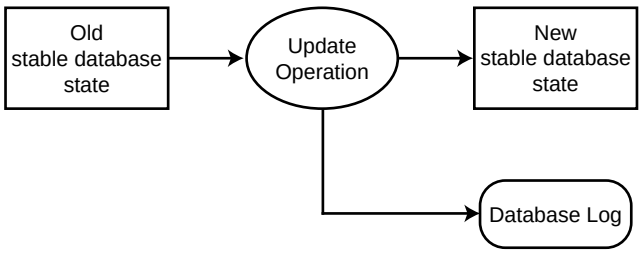

Figure 2: Update Operation Execution

We will discuss the log contents and log management shortly. Our primary focus will be on the recovery issues that arise due to in-place updates. We summarize out-of-place update issues in the next section; more details can be found in [Ver 78].

3.2 Out-of-Place Update

Typical techniques for out-of-place updating are *shadowing* ([ABC+79, Gra79]) and *differential files* [SL76]. Shadowing uses duplicate stable storage pages in executing updates. Thus every time an update is made, the old stable storage page, called the *shadow page*, is left intact and a new page with the updated data item values is written into the stable database. The access path data structures are updated to indicate that the shadow page contains the current data so that subsequent accesses are to this page. The old stable storage page is retained for recovery purposes (to perform undo).

Recovery based on shadow paging was implemented in the early prototype versions of System R's recovery manager [GMB+81]. This implementation uses shadowing together with logging.

The differential files approach maintains each stable database file as a read-only file. In addition, it maintains a corresponding read-write differential file which stores the changes to that file. Given a logical database file F, let us denote its read-only part as FR and its corresponding differential file as DF. DF consists of two parts: an insertions part, which stores the insertions to F,

denoted DF^+, and a corresponding deletions part, denoted DF^-. All updates are treated as the deletion of the old value and the insertion of a new one. Thus each logical file F is considered to be a view defined as $F = (FR \cup DF^+) - DF^-$. Periodically, the differential file needs to be merged with the read-only base file.

Recovery schemes based on this method simply use private differential files for each transaction, which are then merged with the differential files of each file at commit time. Thus recovery from failures can simply be achieved by discarding the private differential files of noncommitted transactions.

There are studies that indicate that the shadowing and differential files approaches may be advantageous in certain environments. One study by [AD85] investigates the performance of recovery mechanisms based on logging, differential files, and shadow paging, integrated with locking and optimistic (using timestamps) concurrency control algorithms. The results indicate that shadowing, together with locking, can be a feasible alternative to the more common log-based recovery integrated with locking if there are only large (in terms of the base-set size) transactions with sequential access patterns. Similarly, differential files integrated with locking can be a feasible alternative if there are medium-sized and large transactions.

4 Logging

As indicated above, when in-place update scheme is used, each database update is recorded in a log. To motivate the need for a log, consider the following scenario. The DBMS began executing at time 0 and at time t a system failure occurs. During the period [0, t], two transactions (say, T_1 and T_2) pass through the DBMS, one of which (T_1) has completed (i.e., committed), while the other one has not (see Figure 3). The durability property of transactions would require that the effects of T_1 be reflected in the stable database. Similarly, the atomicity property would require that the stable database not contain any of the effects of T_2. However, special precautions need to be taken to ensure this.

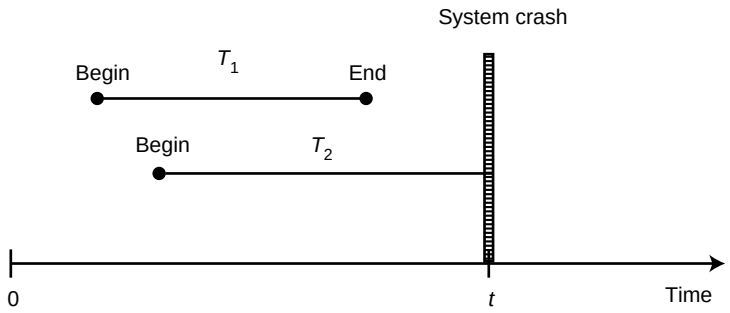

Figure 3: Occurrence of a System Failure

Let us assume that the RM and BM algorithms are such that the buffer

pages are written back to the stable database only when the buffer manager needs new buffer space. In other words, the **flush** command is not used by the RM and the decision to write back the pages into the stable database is taken at the discretion of the buffer manager. In this case it is possible that the volatile database pages that have been updated by T_1 may not have been written back to the stable database at the time of the failure. Therefore, upon recovery, it is important to be able to *redo* the operations of T_1. This requires some information to be kept about the effects of T_1 – such information is kept in the database log. Given this information, it is possible to recover the database from its "old" state to the "new" state that reflects the effects of T_1 (Figure 4).

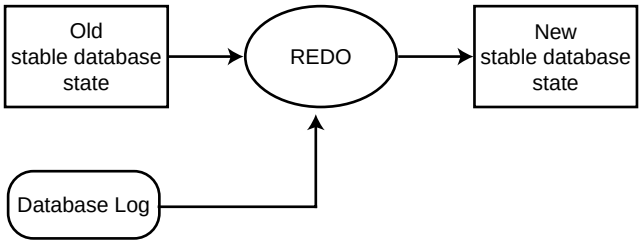

Figure 4: REDO Action

Similarly, it is possible that the buffer manager may have had to write into the stable database some of the volatile database pages that have been updated by T_2. Upon recovery from failures it is necessary to *undo* the operations of T_2.[3] Thus the recovery information should include sufficient data to permit the undo by taking the "new" database state that reflects partial effects of T_2 and recovers the "old" state that existed at the start of T_2 (Figure 5).

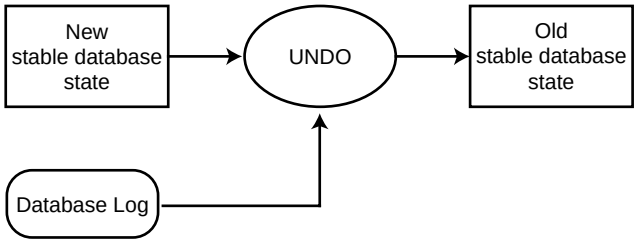

Figure 5: UNDO Action

[3]One might think that it could be possible to continue with the operation of T_2 following restart instead of undoing its operations. However, in general it may not be possible for the RM to determine the point at which the transaction needs to be restarted. Furthermore, the failure may not be a system failure but a transaction failure (i.e., T_2 may actually abort itself) after some of its actions have been reflected in the stable database. Therefore, the possibility of undoing is necessary.

It is important to note that the undo and redo actions are assumed to be idempotent. In other words, their repeated application to a transaction would be equivalent to performing them once.

The contents of the log may differ according to the implementation. However, the following minimal information for each transaction is contained in almost all database logs: a `begin` (transaction) record, update records each of which indicates the data item that is updated, its value before the update (called the *before image*), and its updated value (called the *after image*), and a termination record indicating the transaction termination condition (commit, abort). The granularity of the before and after images may be different, as it is possible to log entire pages or some smaller unit. This is called *operational logging*, since update operations and their effects are individually recorded. This method is used in many systems, including ARIES, but is not the only form of logging that is possible.

Similar to the volatile database, the log is also maintained in main memory buffers (called *log buffers*) and written back to stable storage (called *stable log*) similar to the database buffer pages (Figure 6). The log pages can be written to stable storage in one of two ways. They can be written *synchronously* (more commonly known as *forcing a log*) where the addition of each log record requires that the log be moved from main memory to stable storage. It can also be written *asynchronously*, where the log is moved to stable storage either at periodic intervals or when the buffer fills up. When the log is written synchronously, the execution of the transaction is suspended until the write is complete. This adds significant delay to the response-time performance of the transaction. On the other hand, if a failure occurs immediately after a forced write, it is relatively easy to recover to a consistent database state.

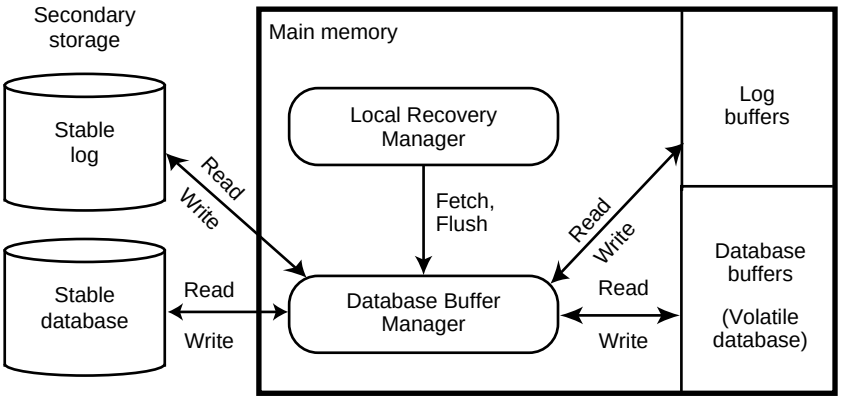

Figure 6: Logging Interface

Whether the log is written synchronously or asynchronously, one very important protocol has to be observed in maintaining logs. Consider a case where the updates to the database are written into the stable storage before the log

is modified in stable storage to reflect the update. If a failure occurs before the log is written, the database will remain in updated form, but the log will not indicate the update that makes it impossible to recover the database to a consistent and up-to-date state. Therefore, the stable log is always updated prior to the updating of the stable database. This is known as the *write-ahead logging* (*WAL*) protocol [Gra79] and can be precisely specified as follows:

1. Before a stable database is updated (perhaps due to actions of a yet uncommitted transaction), the before images should be stored in the stable log. This facilitates undo.
2. When a transaction commits, the after images have to be stored in the stable log prior to the updating of the stable database. This facilitates redo.

5 Recovery Methods

Recovery methods can be classified according to the way they address the relationship between the RM and the BM. There are two fundamental questions that identify the possible alternatives:

1. *Can the BM write the buffer pages updated by a transaction into stable storage during the execution of that transaction, or does it have to wait for the RM to instruct it to write them back?*

   BM may need to write pages dirtied by transaction T_1 to stable storage before the completion of T_1, because it may run out of buffer frames and another transaction T_2 may need to bring data pages into the database buffer. If BM can proceed with writing the dirty pages to stable database, it is said to implement a *steal* policy (since T_2 would be stealing buffer frames from T_1).

   It is possible, however, for the RM to *pin* (or *fix* these pages in the buffer. In that case, BM cannot flush them to stable database whenever it wants (or needs) to. This is called a *no-steal* policy.

2. *Can the RM force the BM to flush the buffer pages updated by a transaction into the stable storage at the end of that transaction (i.e., the commit point), or does the BM flush them out whenever it needs to according to its buffer management algorithm?*

   If the RM can force the BM into flushing pages, this is called the *force* policy; otherwise the system implements a *no-force* policy.

Accordingly, four alternatives can be identified: (1) steal/no-force, (2) steal/force, (3) no-steal/no-force, and (4) no-steal/force. We will consider each of these in more detail. What changes in these alternative policies is how the system deals with transaction commits and aborts, and how it does recovery. In other words, these policies determine how the DBMS implements the (abstract) `commit`, `abort`, and `recover` commands.

5.1 Steal/No-force

This type of RM policy is called a redo/undo algorithm in [BHG87] since it requires, as we will see, performing both the redo and undo operations upon recovery.

Abort Processing. As we indicated before, abort is an indication of transaction failure. Since the buffer manager may have written the updated pages into the stable database, abort will have to undo the actions of the transaction. Therefore, the RM reads the log records for that specific transaction and replaces the values of the updated data items in the volatile database with their before images. The scheduler is then informed about the successful completion of the abort action. This process is called the *transaction undo* or *partial undo*.

An alternative implementation is the use of an *abort list*, which stores the identifiers of all the transactions that have been aborted. If such a list is used, the abort action is considered to be complete as soon as the transaction's identifier is included in the abort list. We won't consider this alternative in the remainder.

Note that even though the values of the updated data items in the stable database are not restored to their before images, the transaction is considered to be aborted at this point. The buffer manager will write the "corrected" volatile database pages into the stable database at a future time, thereby restoring it to its state prior to that transaction.

Commit Processing. The **commit** command causes an `commit` record to be written into the log by the RM. Under this scenario, no other action is taken in executing a commit command other than informing the scheduler about the successful completion of the commit action.

An alternative to explicitly writing a `commit` record into the log is to add the transaction's identifier to a *commit list*, which is a list of the identifiers of transactions that have committed. In this case the commit action is accepted as complete as soon as the transaction identifier is stored in this list. Again, we will not consider this alternative.

Recovery Processing. The RM starts the recovery action by analyzing the log to determine which transactions need to be redone and which ones need to be undone. Any transaction that has both a `begin` record and a `commit` record needs to be redone. This is called *partial redo*. Similarly, any transaction that has a `begin` record but no corresponding `commit` record needs to be undone. This action is called *global undo*, as opposed to the transaction undo discussed above. The difference is that the effects of all incomplete transactions need to be rolled back, not one.

5.2 Steal/Force

These are also called undo/no-redo in [BHG87].

Abort Processing. The execution of **abort** is identical to the previous case. Upon transaction failure, the RM initiates a partial undo for that particular transaction.

Commit Processing. The RM issues forces the BM to flush all the updated volatile database pages into the stable database. The commit command is then executed by placing a `commit` record in the log. At this point commit processing is considered to be completed.

Recovery Processing. Since all the updated pages are written into the stable database at the commit point, there is no need to perform redo; all the effects of successful transactions will have been reflected in the stable database. Therefore, the recovery action initiated by the RM consists of a global undo.

5.3 No-steal/No-force

In this case the RM controls the writing of the volatile database pages into stable storage. The key here is not to permit the BM to write any updated volatile database page into the stable database until at least the transaction commit point. This is accomplished by "fix"ing the specified page in the database buffer, thus preventing it from being written back to the stable database by the BM. Thus any page fetch request to the buffer manager for a write operation specifies that the page needs to be fixed[4]. Note that this precludes the need for a global undo operation and is therefore called a redo/no-undo algorithm in [BHG87].

Abort Processing. Since the volatile database pages have not been written to the stable database, no special action is necessary. However, the RM needs to release the pages that have been fixed by the transaction by sending an unfix request to the BM. It is then sufficient to carry out the abort action either by writing an abort record in the log or by including the transaction in the abort list, informing the scheduler and then forgetting about the transaction.

Commit Processing. The RM requests the BM to unfix the volatile database pages that were previously fixed by that transaction. These pages may now be written back to the stable database at the discretion of the BM. Then a `commit` record is placed in the log to complete commit processing.

Recover. As we mentioned above, since the volatile database pages that have been updated by ongoing transactions are not yet written into the stable database, there is no necessity for global undo. The RM, therefore, initiates a partial redo action to recover those transactions that may have already committed, but whose volatile database pages may not have yet written into the stable database.

[4]Of course, any lock escalation messages also cause the page to be fixed.

5.4 No-steal/Force

This is the case where the RM forces the BM to write the updated volatile database pages into the stable database at precisely the commit point—not before and not after. This strategy is called no-undo/no-redo in [BHG87].

Abort Processing. Abort processing in this case is identical to that of the no-steal/no-force case.

Commit Processing. The RM asks the BM to an unfix and flush the volatile database pages that were previously fixed by that transaction. This forces the BM to write back all the volatile database pages into the stable database. It then writes a `commit` record into the log. The important point to note here is that all three of these operations have to be executed as an atomic action. One step that can be taken to achieve this atomicity is to issue only a **flush** command, which serves to unfix the pages as well. Methods for ensuring this atomicity are beyond the scope of our discussion (see [BHG87]).

Recovery Processing. The DBMS does not need to do anything to recover in this case. This is true since the stable database reflects the effects of all the successful transactions and none of the effects of the uncommitted transactions.

5.5 Which one is better?

Each of these recovery approaches have their advantages and disadvantages. The choice depends on what one attempts to optimize. If the critical optimization criterion is to reduce the recovery work, then, of course, no-steal/force is the best strategy. As indicated above, it ensures that no dirty pages are written to stable database before transaction commits, and that all of the dirty pages are flushed to database at commit point (of course usingthe WAL protocol). Thus, recovery does not need to do anything. However, it suffers from two basic problems. First is that it has a high run-time overhead for each transaction, because of the I/O that results from forcing dirty pages at the end of transactions. This eliminates any possibility of doing I/O optimization. Second, since it does not allow the BM to steal buffer frames, the buffer space becomes a critical resource. Transactions are limited by the amount of available buffer space. This limits the number of concurrently running transactions and, thus, the system throughput.

On the other extreme, steal/no-force has a high recovery overhead since it lets the BM to flush out dirty pages before transaction commits, thus requiring undo when transaction aborts or when system crashes. It also allows the BM to hold dirty pages of committed transactions in volatile storage, thus requiring redo when recovering from crashes. However, its run-time overhead is minimal. It also minimizes the dependence between the RM and the BM.

The other alternatives have a combination of the advantages and disadvantages of these two approaches. Most commercial systems implement a steal/no-force crash recovery algorithm, because of its preferable performance charac-

teristics. These systems are optimized for "normal" processing to reduce the transaction execution overhead when failures do not occur (more common case). Even when failures occur, the entire recovery process is already so long that it does not matter that database recovery adds a bit more overhead. We will consider detailed implementation issues of steal/no-force algorithms in Section 8.

6 Checkpointing

In most of the RM implementation strategies, the execution of the recovery action requires searching the entire log. This is a significant overhead because the RM is trying to find all the transactions that need to be undone and redone. The overhead can be reduced if it is possible to build a wall which signifies that the database at that point is up-to-date and consistent. In that case, the redo has to start from that point on and the undo only has to go back to that point. This process of building the wall is called *checkpointing*.

Checkpointing is achieved in three steps [Gra79]:

1. Write a `begin_checkpoint` record into the log.
2. Collect the checkpoint data into the stable storage.
3. Write an `end_checkpoint` record into the log.

The first and the third steps enforce the atomicity of the checkpointing operation. If a system failure occurs during checkpointing, the recovery process will not find an `end_checkpoint` record and will consider checkpointing not completed.

There are a number of different alternatives for the data that is collected in Step 2, how it is collected, and where it is stored. We will consider one example here, called *transaction-consistent checkpointing* ([Gra79, GMB+81]). The checkpointing starts by writing the `begin_checkpoint` record in the log and stopping the acceptance of any new transactions by the RM. Once the active transactions are all completed, all the updated volatile database pages are flushed to the stable database followed by the insertion of an `end_checkpoint` record into the log. In this case, the redo action only needs to start from the `end_checkpoint` record in the log. The undo action can go the reverse direction, starting from the end of the log and stopping at the `end_checkpoint` record.

Transaction-consistent checkpointing is not the most efficient algorithm, since a significant delay is experienced by all the transactions. There are alternative checkpointing schemes such as action-consistent checkpoints, fuzzy checkpoints, and others ([Gra79, Lin79]).

7 Handling Media Failures

The previous discussion focused on nonmedia failures, where the database as well as the log stored in the stable storage survive the failure. Media failures

may either be quite catastrophic, causing the loss of the stable database or of the stable log, or they can simply result in partial loss of the database or the log (e.g., loss of a track or two).

The methods that have been devised for dealing with this situation are again based on duplexing. To cope with catastrophic media failures, an *archive* copy of both the database and the log is maintained on a different (tertiary) storage medium, which is typically the magnetic tape or CD-ROM. Thus the DBMS deals with three levels of memory hierarchy: the main memory, random access disk storage, and magnetic tape (Figure 7). To deal with less catastrophic failures, having duplicate copies of the database and log may be sufficient.

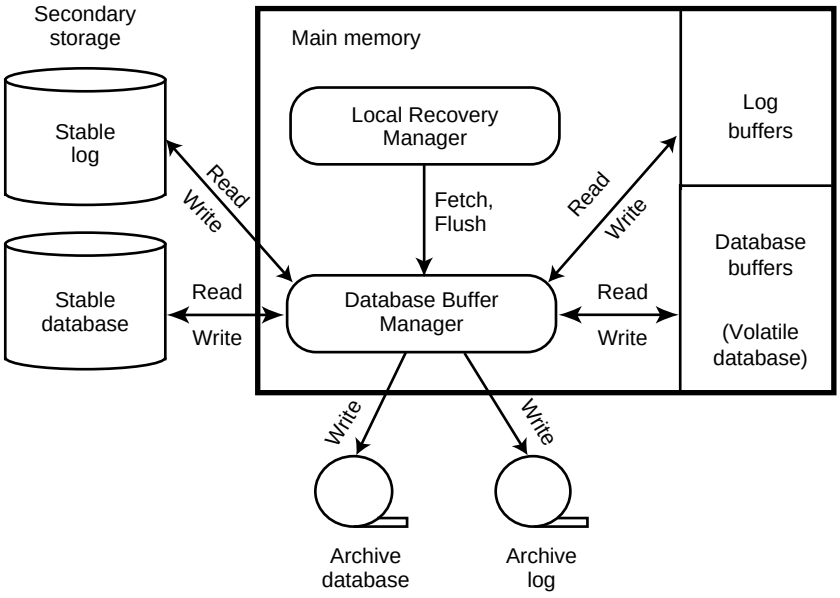

Figure 7: Full Memory Hierarchy Managed by RM and BM

When a media failure occurs, the database is recovered from the archive copy by redoing and undoing the transactions as stored in the archive log. The real question is how the archive database is stored. If we consider the large sizes of current databases, the overhead of writing the entire database to tertiary storage is significant. Two methods that have been proposed for dealing with this are to perform the archiving activity concurrent with normal processing and to archive the database incrementally as changes occur so that each archive version contains only the changes that have occurred since the previous archiving.

8 Implementation Details of Steal/No-Force Recovery

Now go and read Chapter 20 of the textbook

References

[AD85] R. Agrawal and D. J. DeWitt. Integrated Concurrency Control and Recovery Mechanisms. *ACM Trans. Database Syst.*, (December 1985), 10(4): 529–564.

[ABC+79] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: A Relational Database Management System. *ACM Trans. on Database Syst.* (June 1976), 1(2):97–137.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Reading, Mass.: Addison-Wesley, 1987.

[Den68] P. J. Denning. The Working Set Model for Program Behavior. *Commun. ACM* (May 1968), 11(5): 323–333.

[Den80] P. J. Denning. Working Sets: Past and Present. *IEEE Trans. Software Eng.* (January 1980), SE-6(1): 64–84.

[EH84] W. Effelsberg and T. Härder. Principles of Database Buffer Management. *ACM Trans. Database Syst.* (December 1984), 9(4): 560–595.

[Gra79] J. N. Gray. Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmüller (eds.), *Operating Systems: An Advanced Course*, New York: Springer-Verlag, 1979, pp. 393–481.

[GMB+81] J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Comput. Surv.* (June 1981), 13(2): 223–242.

[HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* (December 1983), 15(4): 287–317.

[LS76] B. Lampson and H. Sturgis. *Crash Recovery in Distributed Data Storage System*. Technical Report, Palo Alto, Calif.: Xerox Palo Alto Research Center, 1976.

[Lin79] B. Lindsay. *Notes on Distributed Databases*. Technical Report RJ 2517, San Jose, Calif.: IBM San Jose Research Laboratory, 1979.

[SL76] D. G. Severence and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.* (September 1976), 1(3): 256–261.

[Sto81] M. Stonebraker. Operating System Support for Database Management. *Commun. ACM* (July 1981), 24(7): 412–418.

[Ver 78] J. S. Verhofstadt. Recovery Techniques for Database Systems. *ACM Comput. Surv.* (June 1978), 10(2): 168–195.