**Recovery System**

An integral part of a database system is a **recovery scheme** that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide **high availability**; that is, it must ensure that within minimum time the database should be available after failure.

**Failure Classification**

A Failure is an event at which the system does not perform according to specifications. There are various types of failure that may occur in a system, each of which needs to be dealt within a different manner. We consider only the following types of failure:

* **Transaction failure**. There are two types of errors that may cause a transaction to fail:
* **Logical error**. The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, or resource limit exceeded.
* **System error**. The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re -executed at a later time.
* **System crash**. There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that bring the system to a halt when there is an error.

* **Disk failure**. A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure.

The determination of how the system should recover from failures, depends on failure modes of those devices used for storing data. Based on such failures, recovery manager employs algorithm which takes care of the following things:

1. Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.
2. Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

**Storage Structure**

The storage media can be distinguished by their relative speed, capacity, and resilience to failure. We identify three categories of storage:

* + Volatile storage
  + Nonvolatile storage
  + Stable storage

Stable storage, plays a critical role in recovery algorithms.

**Stable-Storage Implementation**

To implement stable storage, we need to replicate the needed information in several nonvolatile storage media (usually disk) that RAID systems guarantee that the failure of a single disk (even during data transfer) will not result in loss of data. The simplest and fastest form of RAID is the mirrored disk, which keeps two copies of each block, on separate disks. Other forms of RAID offer lower costs, but at the expense of lower performance.

RAID systems, however, cannot guard against data loss due to disasters such as fires or flooding. Many systems store archival backups of tapes off site to guard against such disasters. However, since tapes cannot be carried off site continually, updates since the most recent time that tapes were carried off site could be lost in such a disaster. More secure systems keep a copy of each block of stable storage

at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system. Since the blocks are output to a remote system as and when they are output to local storage, once an output operation is complete, the output is not lost, even in the event of a disaster such as a fire or flood.

Block transfer between memory and disk storage can result in:

• **Successful completion:** The transferred information arrived safely at its destination.

• **Partial failure:** A failure occurred in the midst of transfer, and the destination block has incorrect information.

• **Total failure**. The failure occurred sufficiently early during the transfer that the destination block remains intact.

We require that, if a **data-transfer failure** occurs, the system detects it and invokes a recovery procedure to restore the block to a consistent state. To do so, the system must maintain two physical blocks for each logical database block; in the case of mirrored disks, both blocks are at the same location; in the case of remote backup, one of the blocks is local, whereas the other is at a remote site. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. The output is completed only after the second write completes successfully.

If the system fails while blocks are being written, it is possible that the two copies of a block are inconsistent with each other. During recovery, for each block, the system would need to examine two copies of the blocks. If both are the same and no detectable error exists, then no further actions are necessary.

If the system detects an error in one block, then it replaces its content with the content of the other block. If both blocks contain no detectable error, but they differ in content, then the system replaces the content of the first block with the value of the second.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

**Recovery and Atomicity**

Consider again our simplified banking system and a transaction *Ti* that transfers $50 from account *A* to account *B*, with initial values of *A* and *B* being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of *Ti*, after output(*BA*) has takenplace, but before output(*BB*)was executed,where *BA* and *BB* denote the buffer blocks on which *A*and *B* reside. Since the memory contents were lost, we do not know the fate of the transaction.

When the system restarts, the value of *A* would be $950, while that of *B* would be $2000, which is clearly inconsistent with the atomicity requirement for transaction *Ti* . Unfortunately, there is no way to find out by examining the database state what blocks had been output, and what had not, before the crash.

To achieve our goal of atomicity, we must first output to stable storage information describing the modifications, without modifying the database itself. As we shall see, this information can help us ensure that all modifications performed by committed transactions are reflected in the database later on.

**Log Records**

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database. There are several types of log records. An **update log record** describes a single database write. It has these fields:

* **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
* **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
* **Old value**, which is the value of the data item prior to the write.
* **New value**, which is the value that the data item will have after the write.

We represent an update log record as*<Ti , Xj , V*1*, V*2*>*, indicating that transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and has value *V*2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.

Among the types of log records are:

* *<Ti* start*>*. Transaction *Ti* has started.
* *<Ti* commit*>*. Transaction *Ti* has committed.
* *<Ti* abort*>*. Transaction *Ti* has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created.

**Log Records and Database Modification**

The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk. We need to consider the steps a transaction takes in modifying a data item:

**1.** The transaction performs some computations in its own private part of main memory.

**2.** The transaction modifies the data block in the disk buffer in main memory holding the data item.

**3.** The database system executes the output operation that writes the data block to disk.

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and the new value that is to be written for the data item. This allows the system to perform *undo* and *redo* operations as appropriate.

• **Undo** using a log record sets the data item specified in the log record to the old value.

• **Redo** using a log record sets the data item specified in the log record to the new value.

**Using the Log to Redo and Undo Transactions**

We now provide an overview of how the log can be used to recover from a system crash, and to roll back transactions during normal operation.

Consider our simplified banking system.

Let *T*0 be a transaction that transfers $50 from account *A* to account *B*:

*T*0: read(*A*);

*A* := *A* − 50;

write(*A*);

read(*B*);

*B* := *B* + 50;

write(*B*).

Let *T*1 be a transaction that withdraws $100 from account *C*:

*T*1: read(*C*);

*C* := *C* − 100;

write(*C*).

Figure 1 shows the log records of the combined effect of both the transactions:

<*T*0 start>

<*T*0 , *A*, 1000, 950>

<*T*0 , *B*, 2000, 2050>

<*T*0 commit>

<*T*1 start>

<*T*1 , *C*, 700, 600>

<*T*1 commit>

Figure 1 Log records for transaction T0 and T1

After a system crash has occurred, the system consults the log to determine which transactions need to be redone, and which need to be undone so as to ensure atomicity.

* Transaction *Ti* needs to be undone if the log contains the record *<Ti* start*>*, but does not contain either the record *<Ti* commit*>*or the record *<Ti* abort*>*.
* Transaction *Ti* needs to be redone if the log contains the record*<Ti* start*>*and either the record *<Ti* commit*>* or the record *<Ti* abort*>*. It may seem strange to redo *Ti* if the record *<Ti* abort*>* is in the log. To see why this works, note that **if *<Ti* abort*>* is in the log, so are the redo-only records written by the undo operation. Thus, the end result will be to undo *Ti* ’s modifications in this case.**

As an illustration, return to our banking example, with transaction *T*0 and *T*1 executed one after the other in the order *T*0 followed by *T*1. Suppose that the system crashes before the completion of the transactions.



Figure 2 Log records for three different cases of system crash

The state of the logs for each of these cases appears in Figure 2 First, let us assume that the crash occurs just after the log record for the step: **write(*B*)** of transaction *T*0 has been written to stable storage (Figure 2a).When the system comes back up, it finds the record *<T*0 start*>* in the log, but no corresponding

*<T*0 commit*>* or *<T*0 abort*>* record. Thus, transaction *T*0 must be undone, so an undo(*T*0) is performed. As a result, the values in accounts *A* and *B* (on the disk) are restored to $1000 and $2000, respectively.

Next, let us assume that the crash comes just after the log record for the step: **write(*C*)** of transaction *T*1 has been written to stable storage (Figure 2b). When the system comes back up, two recovery actions need to be taken. The operation undo(*T*1) must be performed, since the record *<T*1 start*>* appears in the log, but there is no record *<T*1 commit*>* or *<T*1 abort*>*. The operation redo(*T*0) must be performed, since the log contains both the record *<T*0 start*>* and the record *<T*0 commit*>*. At the end of the entire recovery procedure, the values of accounts *A*, *B*, and *C* are $950, $2050, and $700, respectively.

Finally, let us assume that the crash occurs just after the log record: ***<T*1 commit*>*** has been written to stable storage (Figure 2c). When the system comes back up, both *T*0 and *T*1 need to be redone, since the records *<T*0 start*>* and *<T*0 commit*>* appear in the log, as do the records*<T*1 start*>*and*<T*1 commit*>*.After the system performs the recovery procedures redo(*T*0) and redo(*T*1), the values in accounts *A*, *B*, and *C* are $950, $2050, and $600, respectively.

**Checkpoints**

When a system crash occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. There are two major difficulties with this approach:

1. The search process is time-consuming.

2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer

To reduce these types of overhead, we introduce another type of entry in the log called a checkpoint. A [checkpoint, list of active transactions] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, T ] entries in the log before a [checkpoint] entry do not need to have their WRITE operations redone in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every m minutes—or in the number t of committed transactions since the last checkpoint, where the values of m or t are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.

2. Force-write all main memory buffers that have been modified to disk.

3. Write a [checkpoint] record <checkpoint L> (where L is a list of transactions active at the time of the checkpoint) to the log, and force-write the log to disk.

4. Resume executing transactions.

After a system crash has occurred, the system examines the log to find the last <checkpoint L> record (this can be done by searching the log backward, from the end of the log, until the first <checkpoint L> record is found).

The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the <checkpoint L> record was written to the log. Let us denote this set of transactions as T.

* For all transactions Tk in T that have no <Tk commit> record or <Tk abort> record in the log, execute undo(Tk ).
* For all transactions Tk in T such that either the record <Tk commit> or the record <Tk abort> appears in the log, execute redo(Tk ).

Note that we need only examine the part of the log starting with the last checkpoint log record to find the set of transactions T, and to find out whether a commit or abort record occurs in the log for each transaction in T.

Example: Consider the set of transactions {T0, T1, . . . , T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69. . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has

completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

**Shadow Paging**

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method.

Shadow paging considers the database to be made up of a number of fixed size disk pages (or disk blocks)—say, n—for recovery purposes. A directory with n entries is constructed, where the ith entry points to the ith database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the current directory—whose entries point to the most recent or current database pages on disk—is copied into a shadow directory. The shadow directory is then saved on disk while the current directory is used by the transaction. During transaction execution, the shadow directory is never modified. When a write\_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 3 illustrates the concepts of shadow and current directories. For pages updated by the

transaction, two versions are kept. **The old version is referenced by the shadow directory and the new version by the current directory.**



Figure 3 An example of shadow paging

To recover from a failure (**not committed**) during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by restoring the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. **Committing** a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NOUNDO/ NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. This technique has following disadvantages:

* The updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies.
* If the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant.
* A further complication is how to handle garbage collection when a transaction commits. The old

pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits.

* Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

References:

1. Database System Concepts By Abraham Silberschatz, Henry F. Korth, S. Sudarshan
2. Fundamentals of Database Systems By Elmsari & Navathe