K.D.K COLLEGE OF ENGINEERING,NAGPUR

Database Management Systems

6.1 Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information. In this chapter,we shall 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 reexecuted 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. Hence, the fail-stop assumption is a reasonable one.

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 tapes, are used to recover from the failure.

6.2 Data Access

The database system resides permanently on nonvolatile storage (usually disks), and is partitioned into fixed-length storage units called **blocks**. Blocks are the units of data transfer to and from disk, and may contain several data items.We shall assume that no data item spans two or more blocks. This assumption is realistic for most data-processing applications, such as our banking example. Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks residing on the disk are referred to as **physical blocks**; the blocks residing temporarily in main memory are referred to as **buffer blocks**. The area of memory where blocks reside temporarily is called the **disk buffer**. Block movements between disk and main memory are initiated through the following two operations:

1. input(*B*) transfers the physical block *B* to main memory.
2. output(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

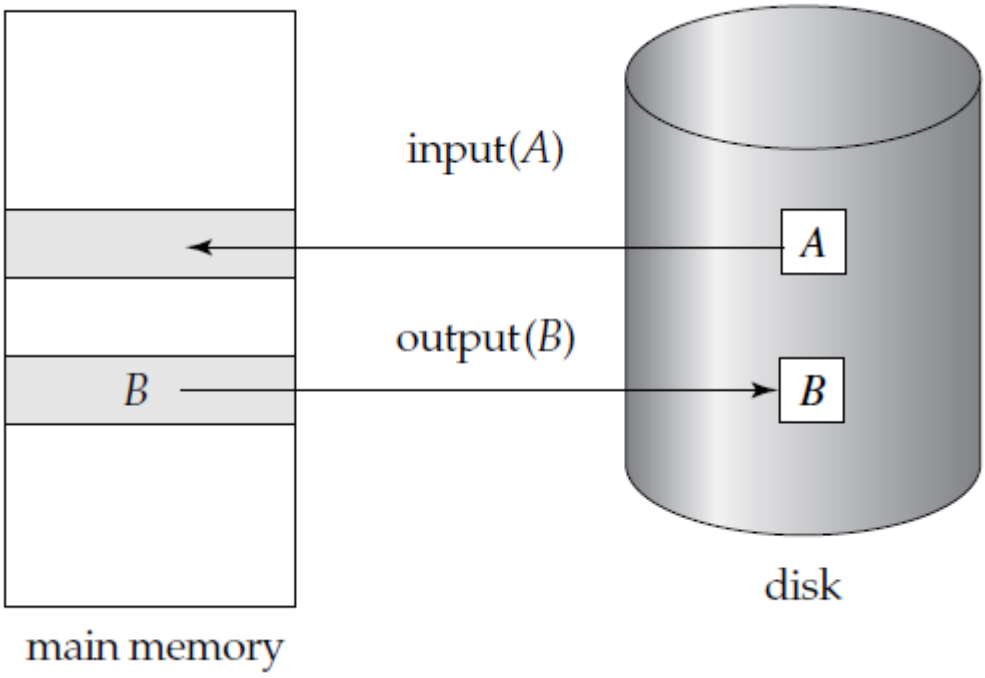


Figure: Block Storage operations

6.3 Recovery and Atomicity

Consider simplified banking system and 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 taken place, 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; thus, we could invoke one of two possible recovery procedures:

• **Reexecute** *Ti*. This procedure will result in the value of *A* becoming $900, rather than $950. Thus, the system enters an inconsistent state.

• **Do not reexecute** *Ti*. The current system state has values of $950 and $2000 for *A* and *B*, respectively. Thus, the system enters an inconsistent state. In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by *Ti*. However, if *Ti* performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made,

but before all of them are made.
To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself. As we shall see, this procedure will allow us to output all the modifications made by a committed transaction, despite failures.

6.4 Log-Based Recovery

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** is the unique identifier of the transaction that performed the write operation.

• **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.

• **Old value** is the value of the data item prior to the write.

• **New value** is the value that the data item will have 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. We denote the various types of log records as:

• <*Ti* start>. Transaction *Ti* has started.

• <*Ti*, *Xj*, *V*1, *V*2>. Transaction *Ti* has performed a write on data item *Xj* . *Xj* had value *V*1 before the write, and will have value *V*2 after the write.

• <*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 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.

6.5 Deferred Database Modification

The **deferred-modification technique** ensures transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits. Recall that a transaction is said to be partially committed once the final action of the transaction has been executed. The version of the deferred-modification technique that we describe in this section assumes that transactions are executed serially.
When a transaction partially commits, the information on the log associated with the transaction is used in executing the deferred writes. If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction *Ti* proceeds as follows. Before *Ti* starts its execution, a record <*Ti* start> is written to the log. A write(*X*) operation by *Ti* results in the writing of a new record to the log. Finally, when *Ti* partially commits, a record <*Ti* commit> is written to the log. When transaction *Ti* partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state. Observe that only the new value of the data item is required by the deferredmodification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field.

To illustrate, reconsider 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*).

Suppose that these transactions are executed serially, in the order *T*0 followed by *T*1, and that the values of accounts *A*, *B*, and *C* before the execution took place were $1000, $2000, and $700, respectively. The portion of the log containing the relevant information on these two transactions appears in Figure There are various orders in which the actual outputs can take place to both the database system and the log as a result of the execution of *T*0 and *T*1.

<*T*0 start>
<*T*0 , *A*, 950>
<*T*0 , *B*, 2050>
<*T*0 commit>
<*T*1 start>
<*T*1 , *C*, 600>
<*T*1 commit>

Figure: Portion of the database log corresponding to *T*0 and *T*1.

One such order appears in below Figure . Note that the value of *A* is changed in the database only after the record <*T*0, *A*, 950> has been placed in the log. Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme uses the following recovery procedure:

• redo(*Ti*) sets the value of all data items updated by transaction *Ti* to the new values.

The set of data items updated by *Ti* and their respective new values can be found in the log. The redo operation must be **idempotent**; that is, executing it several times must be equivalent to executing it once. This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process. After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction *Ti* needs to be redone if and only if the log contains both the record <*Ti* start> and the record <*Ti* commit>. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

Log Database

A = 950
B = 2050
C = 600
<*T*0 start>
<*T*0 , *A*, 950>
<*T*0 , *B*, 2050>
<*T*0 commit>
<*T*1 start>
<*T*1 , *C*, 600>
<*T*1 commit>

Figure State of the log and database corresponding to *T*0 and *T*1.

As an illustration, let us return to our banking example with transactions $T0$ and $T1$ executed one after the other in the order $T0$ followed by $T1$. Figure shows the log that results from the complete execution of $T0$ and $T1$. Let us suppose that the

<$T0$ start>
<$T0$, A, 950>
<$T0$, B, 2050>
<$T0$ start>
<$T0$, A, 950>
<$T0$, B, 2050>
<$T0$ commit>
<$T1$ start>
<$T1$, C, 600>
<$T0$ start>
<$T0$, A, 950>
<$T0$, B, 2050>
<$T0$ commit>
<$T1$ start>
<$T1$, C, 600>
<$T1$ commit>
(a) (b) (c)

Figure The same log as that in Figure 17.3, shown at three different times.

recovery technique restores the database to a consistent state. Assume that the crash occurs just after the log record for the step

write(B)

of transaction $T0$ has been written to stable storage. The log at the time of the crash appears in Figure 17.4a. When the system comes back up, no redo actions need to be taken, since no commit record appears in the log. The values of accounts A and B remain \$1000 and \$2000, respectively. The log records of the incomplete transaction $T0$ can be deleted from the log.
Now, let us assume the crash comes just after the log record for the step

write(C)

of transaction $T1$ has been written to stable storage. In this case, the log at the time of the crash is as in Figure 17.4b. When the system comes back up, the operation redo($T0$) is performed, since the record

<$T0$ commit>

appears in the log on the disk. After this operation is executed, the values of accounts A and B are \$950 and \$2050, respectively. The value of account C remains \$700. As before, the log

records of the incomplete transaction *T*1 can be deleted from the log. Finally, assume that a crash occurs just after the log record

<*T*1 commit>
is written to stable storage. The log at the time of this crash is as in Figure When the system comes back up, two commit records are in the log: one for *T*0 and one for *T*1. Therefore, the system must perform operations redo(*T*0) and redo(*T*1), in the order in which their commit records appear in the log. After the system executes these operations, the values of accounts *A*, *B*, and *C* are $950, $2050, and $600, respectively. Finally, let us consider a case in which a second system crash occurs during recovery from the first crash. Some changes may have been made to the database as a result of the redo operations, but all changes may not have been made. When the system comes up after the second crash, recovery proceeds exactly as in the preceding examples. For each commit record
<*Ti* commit>
found in the log, the the system performs the operation redo(*Ti*). In other words, it restarts the recovery actions from the beginning. Since redo writes values to the database independent of the values currently in the database, the result of a successful second attempt at redo is the same as though redo had succeeded the first time.

6.6 Immediate Database Modification

The **immediate-modification technique** allows database modifications to be output to the database while the transaction is still in the active state. Data modifications written by active transactions are called **uncommitted modifications**. In the event of a crash or a transaction failure, the system must use the old-value field of the log records described in Section 17.4 to restore the modified data items to the value they had prior to the start of the transaction. The undo operation, described next, accomplishes this restoration.

Before a transaction *Ti* starts its execution, the system writes the record <*Ti* start> to the log. During its execution, any write(*X*) operation by *Ti* is *preceded* by the writing of the appropriate new update record to the log. When *Ti* partially commits, the system writes the record <*Ti* commit> to the log. Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the correspond ing log record is written out to stable storage.We therefore require that, before execution of an output(*B*) operation, the log records corresponding to *B* be written onto stable storage As an illustration, let us reconsider our simplified banking system, with transactions *T*0 and *T*1 executed one after the other in the order *T*0 followed by *T*1. The portion of the log containing the relevant information concerning these two transactions appears in Figure

Figure shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of *T*0 and *T*1. Notice that
<*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 Portion of the system log corresponding to *T*0 and *T*1.

Log	Database
$<T_0$ start>	
$<T_0, A, 1000, 950>$	
$<T_0, B, 2000, 2050>$	
	$A = 950$
	$B = 2050$
$<T_0$ commit>	
$<T_1$ start>	
$<T_1, C, 700, 600>$	
	$C = 600$
$<T_1$ commit>	

State of system log and database corresponding to T_0 and T_1.

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

- undo(*Ti*) restores the value of all data items updated by transaction *Ti* to the old values.
- redo(*Ti*) sets the value of all data items updated by transaction *Ti* to the new values.

The set of data items updated by *Ti* and their respective old and new values can be found in the log. The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process. After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone:

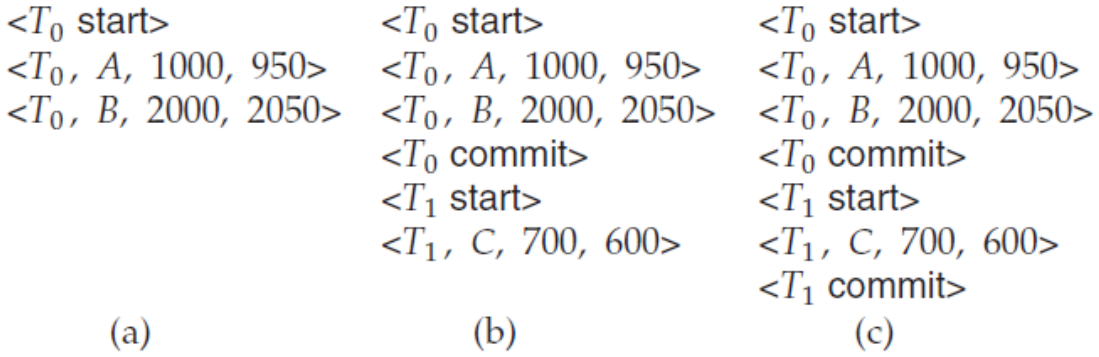
- Transaction *Ti* needs to be undone if the log contains the record <*Ti* start>, but does not contain the record <*Ti* commit>.

• Transaction *Ti* needs to be redone if the log contains both the record <*Ti* start> and the record <*Ti* commit>.

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. We shall consider three cases. The state of the logs for each of these cases appears in Figure

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 17.7a). When the system comes back up, it finds the record <*T*0 start> in the log, but no corresponding <*T*0 commit> 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 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>. 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. Note that the undo(*T*1) operation is performed before the redo(*T*0). In this example, the same outcome would result if the order were reversed. However, the order of doing undo operations first, and then redo operations, is important for the recovery algorithm.

Finally, let us assume that the crash occurs just after the log record
<*T*1 commit>
has been written to stable storage 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.

6.7 Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. 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 nevertheless cause recovery to take longer. To reduce these types of overhead, we introduce checkpoints.

In addition, the system periodically performs **checkpoints**, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction *Ti* that committed prior to the checkpoint. For such a transaction, the <*Ti* commit> record appears in the log before the <checkpoint> record. Any database modifications made by *Ti* must have been written to the database either prior to the checkpoint or as part of the checkpoint itself.
Thus, at recovery time, there is no need to perform a redo operation on *Ti*. This observation allows us to refine our previous recovery schemes. (We continue to assume that transactions are run serially.) After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction *Ti* that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final <checkpoint> record in the log); then it continues the search backward until it finds the next <*Ti* start> record. This record identifies a transaction *Ti*. Once the system has identified transaction *Ti*, the redo and undo operations need to be applied to only transaction *Ti* and all transactions *Tj* that started executing after transaction *Ti*. Let us denote these transactions by the set *T*. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on the modification technique being used. For the immediate-modification technique, the recovery operations are:

• For all transactions *Tk* in *T* that have no <*Tk* commit> record in the log, execute undo(*Tk*).

• For all transactions *Tk* in *T* such that the record <*Tk* commit> appears in the log, execute redo(*Tk*).

6.8 Shadow Paging

An alternative to log-based crash-recovery techniques is shadow paging. The database is partitioned into some number of fixed-length blocks, which are referred to as **pages**. The term *page* is borrowed from operating systems, since we are using a paging scheme for memory management. Assume that there are *n* pages, numbered 1 through *n*. (In practice, *n* may be in the hundreds of thousands.) These pages do not need to be stored in any particular order on disk.
However, there must be a way to find the *i*th page of the database for any given *i*.We use a **page table**, as in Figure for this purpose. The page table has *n* entries—one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on. The example in Figure shows that the logical order of database pages does not need to correspond to the physical order in which the pages are placed on disk.

The key idea behind the shadow-paging technique is to maintain *two* page tables during the life of a transaction: the **current page table** and the **shadow page table**. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk. Suppose that the transaction *Tj* performs a write(*X*) operation, and that *X* resides on the *i*th page. The system executes the write operation as follows:

1. If the *i*th page (that is, the page on which *X* resides) is not already in main memory, then the system issues input(*X*).

2. If this is the write first performed on the *i*th page by this transaction, then the system modifies the current page table as follows:

a. It finds an unused page on disk. Usually, the database system has access
to a list of unused (free) pages

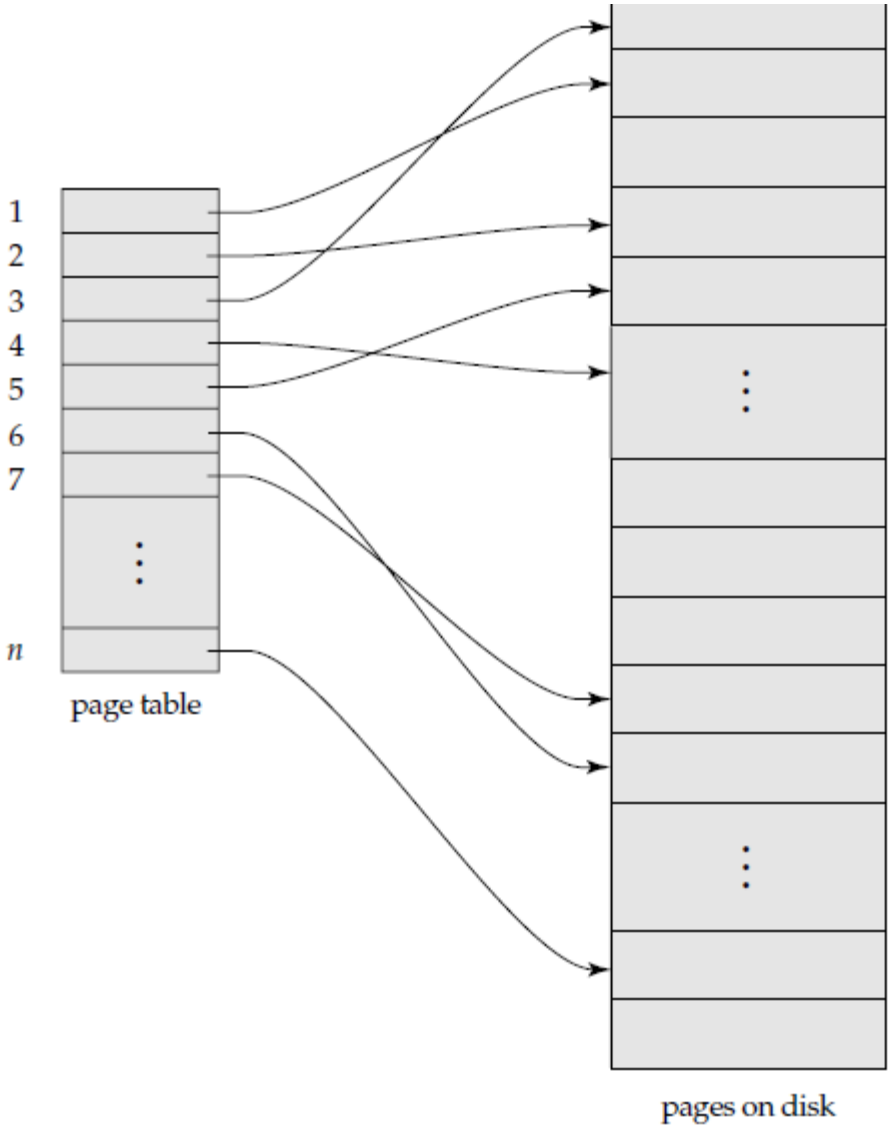


Figure: Shadow page table

b. It deletes the page found in step 2a from the list of free page frames; it copies the contents of the *i*th page to the page found in step 2a. **c.** It modifies the current page table so that the *i*th entry points to the page found in step 2a.
3. It assigns the value of *xj* to *X* in the buffer page.

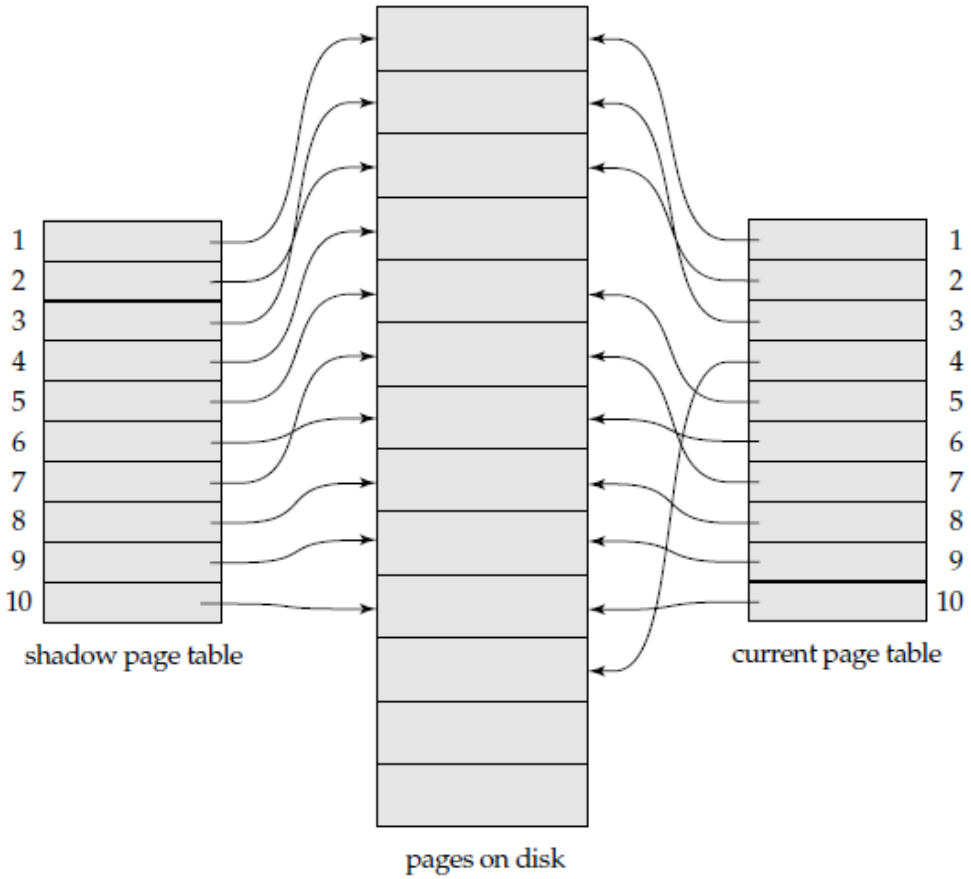


Figure: Shadow and current page tables.

Intuitively, the shadow-page approach to recovery is to store the shadow page table in nonvolatile storage, so that the state of the database prior to the execution of the transaction can be recovered in the event of a crash, or transaction abort. When the transaction commits, the system writes the current page table to nonvolatile storage. The current page table then becomes the new shadow page table, and the next transaction is allowed to begin execution. It is important that the shadow page table be stored in nonvolatile storage, since it provides the only means of locating database pages. The current page table may be kept in main memory (volatile storage).We do not care whether the current page table is lost in a crash, since the system recovers by using the shadow page table.

6.9 Buffer Management

The system stores the database in nonvolatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block $B1$ in main memory when another block $B2$ needs to be brought into memory. If $B1$ has been modified, $B1$ must be output prior to the input of $B2$.

The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of block $B2$ causes block $B1$ to be chosen for output, all log records pertaining to data in $B1$ must be output to stable storage before $B1$ is output. Thus, the sequence of actions by the system would be:

- Output log records to stable storage until all log records pertaining to block $B1$ have been output.
- Output block $B1$ to disk.
- Input block $B2$ from disk to main memory.

6.9.1 Log-Record Buffering:

we have assumed that every log record is output to stable storage at the time it is created. This assumption imposes a high overhead on system execution for several reasons: Typically, output to stable storage is in units of blocks. In most cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level. Also the output of a block to stable storage may involve several output operations at the physical level. The cost of performing the output of a block to stable storage is sufficiently high that it is desirable to output multiple log records at once. To do so, we write log records to a log buffer in main memory, where they stay temporarily until they are output to stable storage. Multiple log records can be gathered in the log buffer, and output to stable storage in a single output operation. The order of log records in the stable storage must be exactly the same as the order in which they were written to the log buffer. As a result of log buffering, a log record may reside in only main memory (volatile storage) for a considerable time before it is output to stable storage. Since such log records are lost if the system crashes, we must impose additional requirements on the recovery techniques to ensure transaction atomicity:

- Transaction Ti enters the commit state after the $<Ti$ commit$>$ log record has been output to stable storage.

- Before the $<Ti$ commit$>$ log record can be output to stable storage, all log records pertaining to transaction Ti must have been output to stable storage.

- Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage. This rule is called the **write-ahead logging (WAL)** rule.

6.9.2 Database Buffering:

The system stores the database in nonvolatile storage (disk), and brings blocks of data into main memory as needed. Since main memory is typically much smaller than the entire database, it may be necessary to overwrite a block *B*1 in main memory when another block *B*2 needs to be brought into memory. If *B*1 has been modified, *B*1 must be output prior to the input of *B*2.

The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of block *B*2 causes block *B*1 to be chosen for output, all log records pertaining to data in *B*1 must be output to stable storage before *B*1 is output. Thus, the sequence of actions by the system would be:

- Output log records to stable storage until all log records pertaining to block *B*1 have been output.
- Output block *B*1 to disk.
- Input block *B*2 from disk to main memory.

It is important that no writes to the block *B*1 be in progress while the system carries out this sequence of actions. We can ensure that there are no writes in progress by using a special means of locking: Before a transaction performs a write on a data item, it must acquire an exclusive lock on the block in which the data item resides. The lock can be released immediately after the update has been performed. Before a block is output, the system obtains an exclusive lock on the block, to ensure that no transaction is updating the block. It releases the lock once the block output has completed.

6.9.3 Operating System Role in Buffer Management:

We can manage the database buffer by using one of two approaches:

1.The database system reserves part of main memory to serve as a buffer that it, rather than the operating system, manages. The database system manages data-block transfer in accordance with the requirements. This approach has the drawback of limiting flexibility in the use of main memory. The buffer must be kept small enough that other applications have sufficient main memory available for their needs. However, even when the other applications are not running, the database will not be able to make use of all the available memory. Likewise, nondatabase applications may not use that part of main memory reserved for the database buffer, even if some of the pages in the database buffer are not being used.

2. The database system implements its buffer within the virtual memory provided by the operating system. Since the operating system knows about the memory requirements of all processes in the system, ideally it should be in charge of deciding what buffer blocks must be force-output to disk, and when. But, to ensure the write-ahead logging requirements the operating system should not write out the database buffer pages itself, but instead should request

the database system to force-output the buffer blocks. The database system in turn would force-output the buffer blocks to the database, after writing relevant log records to stable storage. Unfortunately, almost all current-generation operating systems retain complete control of virtual memory. The operating system reserves space on disk for storing virtual-memory pages that are not currently in main memory; this space is called **swap space**. If the operating system decides to output a block *Bx*, that block is output to the swap space on disk, and there is no way for the database system to get control of the output of buffer blocks. Therefore, if the database buffer is in virtual memory, transfers between database files and the buffer in virtual memory must be managed by the database system.

6.10 ARIES

ARIES uses a number of techniques to reduce the time taken for recovery, and to reduce the overheads of checkpointing. In particular, ARIES is able to avoid redoing many logged operations that have already been applied and to reduce the amount of information logged. The price paid is greater complexity; the benefits are worth the price.

The major differences between ARIES and our advanced recovery algorithm are that ARIES:

1. Uses a **log sequence number** (LSN) to identify log records, and the use of LSNs in database pages to identify which operations have been applied to a database page.
2. Supports **physiological redo** operations, which are physical in that the affected page is physically identified, but can be logical within the page. For instance, the deletion of a record from a page may result in many other records in the page being shifted, if a slotted page structure is used.With physical redo logging, all bytes of the page affected by the shifting of records must be logged. With physiological logging, the deletion operation can be logged, resulting in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.
3. Uses a **dirty page table** to minimize unnecessary redos during recovery. Dirty pages are those that have been updated in memory, and the disk version is not up-to-date.
4. Uses fuzzy checkpointing scheme that only records information about dirty pages and associated information, and does not even require writing of dirty pages to disk. It flushes dirty pages in the background, continuously, instead of writing them during checkpoints.

6.10.1 Data Structures

Each log record in ARIES has a **log sequence number** (**LSN**) that uniquely identifies the record. The number is conceptually just a logical identifier whose value is greater for log records that occur later in the log. In practice, the LSN is generated in such a way that it can also be used to locate the log record on disk. Typically, ARIES splits a log into multiple log files, each of which has a file number. When a log file grows to some limit, ARIES appends further log records to a new log file; the new log file has a file number that is higher by 1 than the previous log file. The LSN then consists of a file number and an offset within the file.
Each page also maintains an identifier called the **PageLSN**. Whenever an operation (whether physical or logical) occurs on a page, the operation stores the LSN of its log record in the

PageLSN field of the page. During the redo phase of recovery, any log records with LSN less than or equal to the PageLSN of a page should not be executed on the page, since their actions are already reflected on the page. In combination with a scheme for recording PageLSNs as part of checkpointing, which we present later, ARIES can avoid even reading many pages for which logged operations are already reflected on disk. Thereby recovery time is reduced significantly.

Each log record also contains the LSN of the previous log record of the same transaction. This value, stored in the PrevLSN field, permits log records of a transaction to be fetched backward, without reading the whole log. There are special redo-only log records generated during transaction rollback, called **compensation log records** (**CLR**s) in ARIES. The CLRs have an that records the LSN of the log that needs to be undone next, when the transaction is being rolled back.extra field, called the UndoNextLSN, The **DirtyPageTable** contains a list of pages that have been updated in the database buffer. For each page, it stores the PageLSN and a field called the RecLSN which helps identify log records that have been applied already to the version of the page on disk.

6.10.2 Recovery Algorithm

ARIES recovers from a system crash in three passes:

• **Analysis pass**: This pass determines which transactions to undo, which pages were dirty at the time of the crash, and the LSN from which the redo pass should start.

• **Redo pass**: This pass starts from a position determined during analysis, and performs a redo, repeating history, to bring the database to a state it was in before the crash.

• **Undo pass**: This pass rolls back all transactions that were incomplete at the time of crash.

Analysis Pass: The analysis pass finds the last complete checkpoint log record, and reads in the DirtyPageTable from this record. It then sets RedoLSN to the minimum of the RecLSNs of the pages in the DirtyPageTable. If there are no dirty pages, it sets RedoLSN to the LSN of the checkpoint log record. The redo pass starts its scan of the log from RedoLSN. All the log records earlier than this point have already been applied to the database pages on disk. The analysis pass initially sets the list of transactions to be undone, undo-list, to the list of transactions in the checkpoint log record. The analysis pass also reads from the checkpoint log record the LSNs of the last log record for each transaction in undo-list. The analysis pass continues scanning forward from the checkpoint. Whenever it finds a log record for a transaction not in the undo-list, it adds the transaction to undo-list. Whenever it finds a transaction end log record, it deletes the transaction from undo-list. All transactions left in undo-list at the end of analysis have to be rolled back later, in the undo pass. The analysis pass also keeps track of the last record of each transaction in undo-list, which is used in the undo pass. The analysis pass also updates DirtyPageTable whenever it finds a log record for an update on a page. If the page is not in DirtyPageTable, the analysis pass adds it to DirtyPageTable, and sets the RecLSN of the page to the LSN of the log record.

Redo Pass: The redo pass repeats history by replaying every action that is not already reflected in the page on disk. The redo pass scans the log forward from RedoLSN. Whenever it finds an update log record, it takes this action:

1. If the page is not inDirtyPageTable or the LSNof the update log record is less than the RecLSN of the page in DirtyPageTable, then the redo pass skips the log record.
2. Otherwise the redo pass fetches the page from disk, and if the PageLSN is less than the LSN of the log record, it redoes the log record. Note that if either of the tests is negative, then the effects of the log record have already appeared on the page. If the first test is negative, it is not even necessary to fetch the page from disk.

Undo Pass and Transaction Rollback: The undo pass is relatively straightforward. It performs a backward scan of the log, undoing all transactions in undo-list. If a CLR is found, it uses the UndoNextLSN field to skip log records that have already been rolled back. Otherwise, it uses the PrevLSN field of the log record to find the next log record to be undone.

Various Sql databases:

Oracle- An Oracle database is a collection of data treated as a unit. The purpose of a database is to store and retrieve related information. A database server is the key to solving the problems of information management. n general, a server reliably manages a large amount of data in a multiuser environment so that many users can concurrently access the same data. All this is accomplished while delivering high performance. A database server also prevents unauthorized access and provides efficient solutions for failure recovery.
Oracle Database is the first database designed for enterprise grid computing, the most flexible and cost effective way to manage information and applications. Enterprise grid computing creates large pools of industry-standard, modular storage and servers. With this architecture, each new system can be rapidly provisioned from the pool of components. There is no need for peak workloads, because capacity can be easily added or reallocated from the resource pools as needed.

Mysql- is an open-source relational database management system(RDBMS). Its name is a combination of "My", the name of co-founder Michael Widenius's daughter, and "SQL", the abbreviation for Structured Query Language.

MySQL is a database system used on the web

- MySQL is a database system that runs on a server
- MySQL is ideal for both small and large applications
- MySQL is very fast, reliable, and easy to use
- MySQL uses standard SQL
- MySQL compiles on a number of platforms
- MySQL is free to download and use
- MySQL is developed, distributed, and supported by Oracle Corporation

- MySQL is named after co-founder Monty Widenius's daughter: My

The data in a MySQL database are stored in tables. A table is a collection of related data, and it consists of columns and rows.

IBM DB2-IBM Db2 contains database server products developed by IBM. These products all support the relational model, but in recent years some products have been extended to support object-relational features and non-relational structures like JSON and XML.

Historically and unlike other database vendors, IBM produced a platform-specific Db2 product for each of its major operating systems. However, in the 1990s IBM changed track and produced a Db2 common product, designed with a common code base to run on different platforms

Sybase-Sybase is a computer software company that develops and sells databasemanagement system (DBMS) and middleware products. The company was founded in 1984, and the headquarters offices are in Emeryville, CA. Sybase was the first enterprise DBMS for the Linux operating system.