Recovery System

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage.

In any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions.

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**;

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.

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.

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.

Recovery algorithms have two parts:

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

- **Volatile storage**
- **Nonvolatile storage**
- **Stable storage**

Stable-Storage Implementation

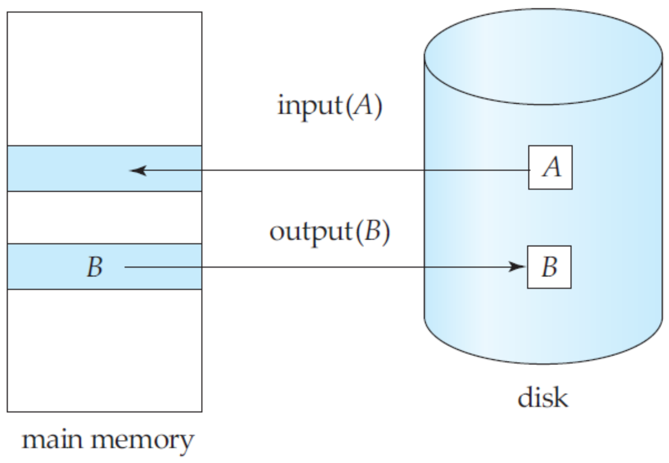
Data Access

- The database 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.)
- 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.



Conceptually, each transaction *Ti* has a private work area in which copies of data items accessed and updated by *Ti* are kept.

The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts.

Each data item *X* kept in the work area of transaction *Ti* is denoted by *xi*.

Transaction *Ti* interacts with the database system by transferring data to and from its work area to the system buffer.

We transfer data by these two operations:

1. read(*X*) assigns the value of data item *X* to the local variable *xi*. It executes this operation as follows:

a. If block B_X on which *X* resides is not in main memory, it issues input(B_X).
b. It assigns to *xi* the value of *X* from the buffer block.

2. write(*X*) assigns the value of local variable *xi* to data item *X* in the buffer block. It executes this operation as follows:

a. If block B_X on which *X* resides is not in main memory, it issues input(B_X).
b. It assigns the value of xi to X in buffer B_X.

Note that both operations may require the transfer of a block from disk to main memory.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to *B* on the disk.

We shall say that the database system performs a **force-output** of buffer *B* if it issues an output(*B*).

When a transaction needs to access a data item *X* for the first time, it must execute read(*X*). The system then performs all updates to *X* on *xi*. At any point during its execution a transaction may execute write(*X*) to reflect the change to *X* in the database itself; write(*X*) must certainly be done after the final write to *X*.

The output(*BX*) operation for the buffer block *BX* on which *X* resides does not need to take effect immediately after write(*X*) is executed, since the block *BX* may contain other data items that are still being accessed.
Thus, the actual output may take place later.

If the system crashes after the write(*X*) operation was executed but before output(*BX*)was executed, the new value of *X* is never written to disk and, thus, is lost.

the database system executes extra actions to ensure that updates performed by committed transactions are not
lost even if there is a system crash.

Recovery and Atomicity
Consider again our simplified banking system and a transaction

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 taken place, but before output(*BB*)was executed, where *BA* and *BB* denote the buffer blocks on which *A* and *B* reside.

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.

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.

Represent an update log record as<Ti , Xj , V1, V2>,

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.

~~We shall introduce several other types of log records later.~~
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.

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.

{
In Section
16.5, we shall see when it is safe to relax this requirement so as to reduce the
overhead imposed by logging. Observe that the log contains a complete record of
all database activity. As a result, the volume of data stored in the log may become
unreasonably large. In Section 16.3.6, we shall show when it is safe to erase log
information.
}

Database Modification

a transaction creates a log record prior to modifying the database.

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.

To understand the role of these log records in recovery, 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.

We say a transaction *modifies the database* if it performs an update on a disk buffer, or on the disk itself; updates to the private part of main memory do not count as database modifications.

If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique.

If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique.

Deferred modification has the overhead that transactions need to make local copies of all updated data items;
further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

The recovery algorithms we describe in this chapter support immediate modification

A recovery algorithm must take into account a variety of factors, including:

• The possibility that a transaction may have committed although some of its database modifications exist only in the disk buffer in main memory and not in the database on disk.

• The possibility that a transaction may have modified the database while in the active state and, as a result of a subsequent failure, may need to abort.

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.

SHADOW COPIES AND SHADOW PAGING

In the **shadow-copy** scheme, a transaction that wants to update the database first creates a complete copy of the database.

All updates are done on the new database copy, leaving the original copy, the **shadow copy**, untouched.

If at any point the transaction has to be aborted, the system merely deletes the new copy.The old copy of the database has not been affected.

The current copy of the database is identified by a pointer, called db-pointer, which is stored on disk.

If the transaction partially commits (that is, executes its final statement) it is committed as follows:

First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.

After the operating system has written all the pages to disk, the database system updates the pointer db pointer
to point to the new copy of the database; the new copy then becomes the current copy of the database.

The old copy of the database is then deleted. The transaction is said to have been *committed* at the point where the updated db-pointer is written to disk.

The implementation actually depends on the write to db-pointer being atomic; that is, either all its bytes are written or none of its bytes are written.

Disk systems provide atomic updates to entire blocks, or at least to a disk sector.

Shadow copy schemes are commonly used by text editors (saving the file is equivalent to transaction commit, while quitting without saving the file is equivalent to transaction abort).

Shadow copying can be used for small databases, but copying a large database would be extremely expensive.

A variant of shadow copying, called **shadow-paging**, reduces copying as follows:

the scheme uses a page table containing pointers to all pages; the page table itself and all updated pages are copied to a new location.

Any page which is not updated by a transaction is not copied, but instead the new page table just stores a pointer to the original page.

Any page which is not updated by a transaction is not copied, but instead the new page table just stores a pointer to the original page.

Shadow paging unfortunately does not work well with concurrent transactions and is not widely used in databases.