Chapter 13(1)
Data recovery

Recovery is part of the more general task of *transaction management.* it means recovering the database itself, i.e., restoring the database to a state that is known to be correct after some failure has made the current state inconsistent.

The underlying principle of such recovery is simply redundancy *at the physical level.* This is particularly true for system failures, when a disk crashes. In this case, a backup copy (a *dump*) has to be reloaded, then we have to use a *log file* to redo all the transactions issued since the dump was taken.

Transactions

A *transaction* is a logical unit of work. Consider the following example: suppose the part relation `P` includes an additional attribute `TotQty`, representing the total shipment quantity for the part in question, and consider the following segment:

```
Begin Transaction;
 Exec SQL Insert into SP
 Values ('S5', 'P1', 1000);
 If any error occurred Go To UNDO;
 Exec SQL Update P
 Set TotQty=TotQty+1000
 WHERE P#='P1';
 If any error occurred GO TO UNDO;
 COMMIT;
 Go To FINISH;
 UNDO:
   ROLLBACK;
 FINISH: return;
```

What could go wrong?

The above example tried to insert another shipment by `S5`, and is involved with two update operations: one Insert, followed by an Update.

The important thing is that the database is not consistent *between* those two operations: it temporarily violates the constraint that the value of `TotQty` for part `P1` be equal to the sum of all `QTY` values for P1.

This example shows that a logic unit of work could be a *sequence* of several such operations, that transforms a consistent state of the database into another, w/o preserving consistency at all intermediate points.

What should we do?

Obviously, the important thing is that it must not be allowed for one of the updates to be executed, and the others not. In other words, we want to have some mechanism to guarantee that either both or none of the updates be executed.

A system that supports *transaction management* makes sure that if the transaction executes some updates and then a failure occurs before the transaction reaches its planned termination, then those updates will be undone. Thus, the unit will become *atomic.*

A transaction either executes in its entirety or is totally canceled, via two system operations: COMMIT, and ROLLBACK.

The COMMIT operation signals a successful end of a transaction, and tells the transaction manager that a logic unit of work has been successfully completed, the database is in a consistent sate again, thus all of the updates made can now be made permanent out of a buffer.

On the other hand, the ROLLBACK operation signals an unsuccessful end of a transaction, and tells the manager that something has gone wrong, the database might be in an inconsistent state, and all of the updates made so far must be undone, i.e., the temporary results thrown out of the buffer.

In both cases, the manager might send some message back. E.g., “Shipment added” if the COMMIT is reached, or “Error-shipment not added” otherwise.

One of two ways out

A transaction begins with a successful execution of a BEGIN TRANSACTION statement, and ends with a successful execution of either a COMMIT or a ROLLBACK statement.

COMMIT establishes another *commit point,* and ROLLBACK rolls the database back to the state it was in at BEGIN TRANSACTION, i.e., the last commit point.

When a commit point is established, all updates made by the executing program since the previous commit point are committed, i.e., made permanent.

What to do after a failure?

If the system crashes after the COMMIT has been honored, but before the updates have been physically written to the database, the restart procedure will still install those updates in the database by checking out a log file to find out the data to be inserted into a table.

Thus, the transaction is actually the unit of recovery, as well.

The ACID requirements

*A*tomicity, of course, refers to the fact that transactions are atomic.

*C*onsistency refers to the fact that a transaction transforms database from a consistent state to another consistent state.

*I*solation means that although in general, there can be many transactions running concurrently, any given transaction's updates are concealed from all the rest, until that transaction commits, when its result will be revealed.

*D*urability means that once a transaction commits, its updates survive in the database, even if there is a subsequent system crash.

System recovery

A system must be prepared to recover, not only from a local failure such as an overflow within an individual transaction, but also from more global failures such as a power outage. A global failure affects all of the transactions in progress at the time of the failure, and hence has significant system-wide implications.

We consider two kinds of global failures: *system failure,* which affects all the transactions currently in progress but do not physically damage the database; and *media failures,* e.g., disk crash, which do cause damage to the database, or to some portion of it, and affect at least that transaction currently using that portion.

System failures

The key point regarding for this category of failure is that the contents of main memory, particularly, the database buffers, are lost. The precise state of any transaction that was in progress at the time of the failure is therefore no longer known.

Thus, those transactions must be undone, when the system restarts. Also, it might be necessary to redo certain transactions at the restart time that did successfully complete prior to the crashes but did not manage get their updates transferred from the database buffers to the physical database.

Question: How does the system know at restart time what to undo, and what to redo?

Answer: The log file.

A bit more details

At certain prescribed period, e.g., whenever certain number of entries have been written to the log, the system automatically takes a checkpoint, i.e., physically write the contents of the database buffers out to the physical database, and write a special checkpoint record out to the physical log, as well.

This log file provides a list of all the transactions that were in progress at the time the point was taken, and can be used to redo and/or undo whatever that are needed.

Media recovery

Recovery from such a failure typically involves reloading the database from a backup copy,(a *dump*),and then using the log, both active and archive portions, to redo all transactions that completed since that backup copy was taken.

This clearly implies the need for a dump/restore utility.

Check out page 4 and 5 in the `MySQL` Lab notes for the operational details as how to dump and restore database files in `MySQL`.

Two phase commit

Two-phase commit is important whenever a given transaction can interact with several independent "resource managers", each manages its own "recoverable sources", and keeps its own recovery log files.

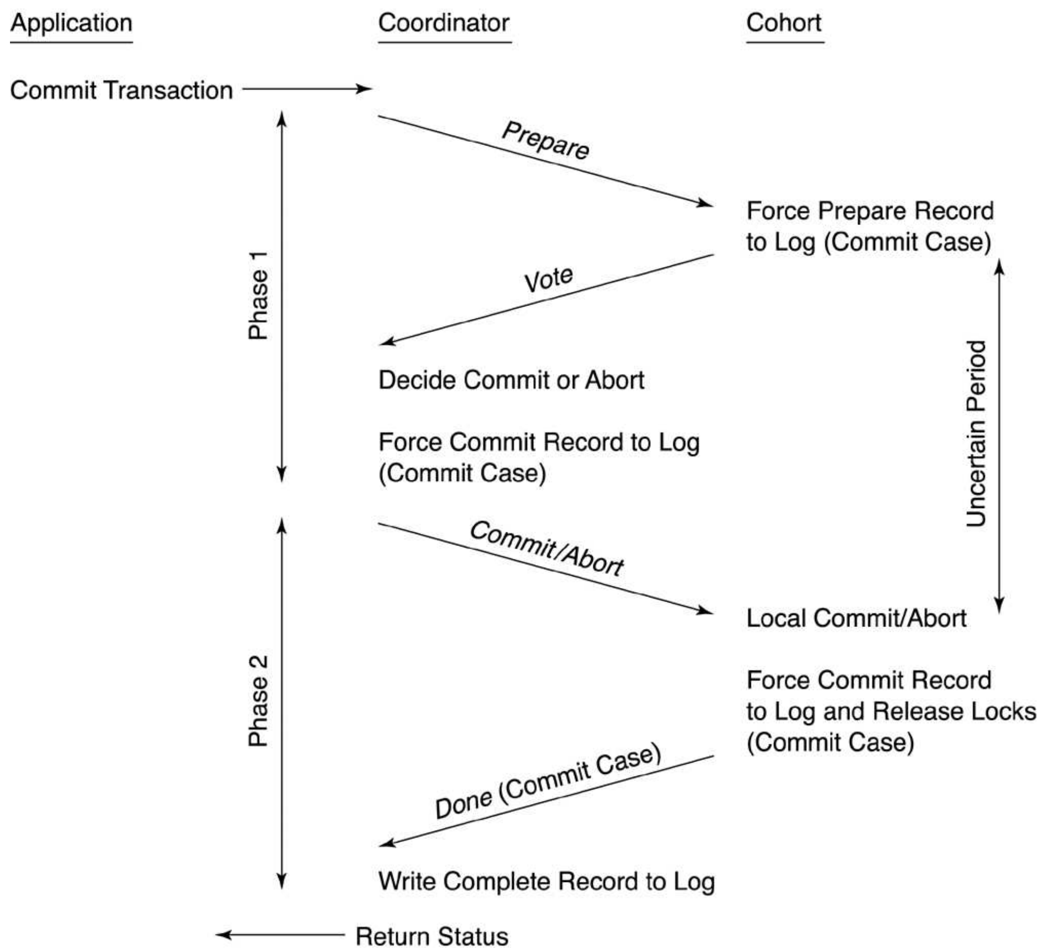
For example, consider a transaction running on an IBM mainframe that updates both an IMS database, and a DB2 database. If the transaction completes successfully, then all of its updates, to both the IMS and the DB2 databases must be committed; conversely, if it fails, then all of its updates must be rolled back.

Thus, it follows that it must not issue a COMMIT to IMS and a ROLLBACK to DB2.

Instead, a transaction must issue a single *system-wide* COMMIT (or ROLLBACK) which is then handled by a system coordinator, using "two-phase commit", to guarantee that both resource manages commit or roll back the updates they are responsible for, and also provide that guarantee even if the system fails in the middle of the process.

Picture first

Assume for simplicity that the transaction has completed its database processing successfully, so that a COMMIT is in order.



How does it work?

When a coordinator receives a request to issue a system wide commit, it goes through the following two-phase process:

1. It instructs all resource managers to get ready to "go either way" on the transaction, i.e., each participant in the process must force all the log entries for local resources be written out to a physical log record.

When all the writings are successfully completed, the resource manger replies "OK" to the coordinator; otherwise, replies "NOT OK." This is referred to as a "vote" in the picture.

2. When the coordinator has received replies from all the participants, the coordinator makes a decision. If all replies were “OK”, that entry will be “commit”; otherwise, “rollback”.

The coordinator also sends this decision out to all the local managers and forces an entry to its own physical log, recording its decision regarding the transaction.

Now, if the system fails at some point during the overall process, the restart procedure will look for the decision record in the coordinator’s log. Once found, then the two -phase commit process can pick up where it left off. If it does not find the log, a rollback will be carried out.

SQL facilities

SQL's support for transaction-based recovery follows the general outline described in the previous slides. Particularly, SQL does support the usual COMMIT and ROLLBACK statements. These statements force a CLOSE for every open cursor, thus causing all database positioning to be lost.

One syntactical difference is that an SQL segment does not include any explicit *BEGIN TRANSACTION statement.* A transaction starts implicitly whenever a "transaction-initiating" statement is encountered.

Also, a special `SET TRANSACTION` statement is used to define certain characteristics, such as *access mode* and the *isolation level*, of the next transaction to be initialed.

The access mode can be either READ ONLY, or READ WRITE, with the latter being the default.

The isolation level includes READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE, and will be further discussed in the next chapter.

Homework: Do some research about MySQL's support to transaction-based recovery and send in a non-trivial report on your finding.