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 consistent 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 when a disk crashes and we lose all the data.

In this case, a backup copy (a dump) of consistency has to be reloaded, then we have to use a log file to redo all the transactions issued since the dump was taken.
Transactions

Suppose the Part table includes an additional attribute TotQty, representing the total shipment quantity for the parts, and consider the following transaction:

Begin Transaction;
   exec SQL Insert into SupplyPart
   values ('S5', 'P1', 1000);
   If any error occurred Go To UNDO;
   //What happens if something occurs here....
   exec SQL Update Part
   set TotQty=TotQty+1000
   WHERE PId='P1';
   If any error occurred GO TO UNDO;
   COMMIT; //I am done.
   Go To FINISH;
   UNDO:
      ROLLBACK; //Throw out every change.
   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 into the SupplyPart table, followed by an Update of the Part table.

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 reiterates that, a transaction, i.e., 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 the intermediate points.

**Question:** Do we still remember ACID? 😊
What should we do?

With the previous segment, we mean either both or none of the updates be executed. Namely, the unit has to be *atomic*.

In general, a transaction either executes in its entirety or it is canceled, via two system operations: COMMIT, and ROLLBACK.

This is essentially what the additional exception capturing segment does in the previous example.
What do they do?

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 state again, thus all of the updates made can now be made permanent out of a buffer into the tables.

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. As a result, all of the updates made so far must be undone, i.e., the temporary results thrown out of the buffer.

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

**Question:** How?

**Answer:** By checking out a log file to find out the data to be inserted into a table.

Thus, a transaction, besides a unit of data operation, is actually a unit of recovery, as well.
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

When this happens, the content of the main memory, particularly, the database buffers, are lost. (Remember the difference between RAM and disk?) The precise state of any transaction that was in progress at the time of the failure is therefore no longer known.

When the system restarts, we have to *undo* some of the transactions to go back to a previous consistent state, and then *redo* certain transactions that did successfully complete prior to the crashes but did not manage to 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

Whenever certain number of entries have been written to the log file, 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 file, as well.

This log file provides a list of all the transactions that were in progress at the time when the point was taken, particularly, it makes a record as which transactions generate those updates.

Such a record can be used to redo and/or undo whatever that are needed later on.
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 6 in the *MySQL* Lab notes for the operational details as how to dump database files in *MySQL*, which can then be restored when a need arises.
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.

The key is that everybody has to do the same thing as told by this transaction, to maintain a consistency.

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; otherwise, if it fails, then all of its updates must be rolled back.
What should happen?

The transaction monitor must not issue a COMMIT to IMS and a ROLLBACK to DB2. In other words, the system should not send out the COMMIT command to the two systems separately, as it could fail for either of these two DBMSes.

Instead, a transaction must issue a single system-wide COMMIT (or ROLLBACK) which is then handled by a system coordinator, using “two-phase commit”.

This latter policy is to guarantee that both resource managers commit or rollback the updates they are responsible for, and also provides that guarantee even if the system fails in the middle of the process.
Looks better?

<table>
<thead>
<tr>
<th>Application</th>
<th>Coordinator</th>
<th>Cohort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit Transaction</td>
<td></td>
<td>Force Prepare Record to Log (Commit Case)</td>
</tr>
<tr>
<td></td>
<td>Prepare</td>
<td>Vote</td>
</tr>
<tr>
<td>Phase 1</td>
<td>Decide Commit or Abort</td>
<td>Force Commit Record to Log (Commit Case)</td>
</tr>
<tr>
<td></td>
<td>Commit/Abort</td>
<td>Local Commit/Abort</td>
</tr>
<tr>
<td>Phase 2</td>
<td>Done (Commit Case)</td>
<td>Force Commit Record to Log and Release Locks (Commit Case)</td>
</tr>
<tr>
<td></td>
<td>Write Complete Record to Log</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Return Status</td>
<td></td>
</tr>
</tbody>
</table>
How does it work?

Assume that the transaction has completed its database processing successfully, so that a COMMIT is in order. On receiving this request, the coordinator 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 manager replies “OK” to the coordinator; otherwise, replies “NOT OK.”
2. When the coordinator has received replies from all the participants, the coordinator makes a decision and sends it out to all the local managers, which will then send back an acknowledgment. The coordinator then forces an entry to its own physical log, recording its decision regarding the transaction.

If all replies were “OK”, that entry will be “commit”; otherwise, “rollback”.

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, then a rollback will be carried out.