Transaction Management

Defn: A transaction (Tx) is a process whose execution preserves the consistency of a database...retrieval, insertion, deletion, and update.

Transaction properties:

a. Atomicity: either the entire Tx is completed successfully or none of it is.
b. Consistency: A Tx must take a database from one consistent state to another.
c. Permanency. Once a change is made, it should not be lost because of a subsequent error.
d. Isolation: the results of a Tx should not be visible to another Tx until it is completed.
e. Serializibility. If a number of Tx's are being performed simultaneously in an interleaved manner, then the net result should be the same as if the Tx's had been performed sequentially.

Transaction management is difficult for two reasons:
  System failure
  Concurrency control.

Examples (using ad hoc notation). transfer $50 from acct A to acct B

Read A
A = A - 50
write A
read B
B = B+50
(system failure)
write B

Tx_1 is going to give everyone a 10% raise.  
Tx_2 is going to give Smith a $10,000 raise.

Tx_1                      Tx_2
read Smith.salary        read Smith.salary
s = 1.10*s              s = s + 10000
(time out)              write s
write s                 finish
finish
Part One. Recovery from failures.

Failure types:

1. System Crash...power failure...
2. System errors...for some reason, the system has entered an illegal state, integer overflow, user interrupt etc.
3. Local errors...attempt to access a non-existing record, integrity violation, etc.
4. Errors due to concurrency control...a Tx may have to be aborted to avoid deadlock.
5. Disk crash.
6. Physical catastrophe

The last two aren't strictly speaking database problems but should be the concern of any D.B.A.

Solution: Periodically make a back-up copy of the database and the system logs on secondary storage. In case of a failure the backup database can be recovered up to the last backup and the system log can be used to recover Tx's that have happened since the last backup. Tx's that are not in the system log are lost. Obviously the backup copies should be kept on different disks and or at different sites than the database.

<INGRES NOTE> the Ingres backup procedure is called "ckpdb" and the system log is called the "journal".

Basic File Processing Operations.

Input(X), transfers the physical block containing item X to main memory.
Output(X), transfers the buffer block containing item X to permanent storage

Basic Tx processes.( Capital letters stand for data items, lower case letters for local variables)

read(X,x):
if the page containing X is not in main memory,
then execute Input(X)
perform some computations
x = X;
Write(X,x):
if the page containing X is not in main memory
then Input(X)
perform some computations
X = x;
Etc....
Note that Output operations usually take place independently of any Tx's, they may be timed out or may happen as the result of paging. A process is said to force output of a buffer block if it issues an Output(X) command. Write (X,x) usually does not force output.

**Terminology:**

A Tx which completes its execution successfully, and any changes that it made, are made permanent (i.e. written to disk) is said to be committed, otherwise it is said to be aborted. An aborted transaction must be rolled back (i.e. its changes undone) in order to restore the database to a previous consistent state.

At any given instant, a Tx must be in one of the following states:
- **Active**
  - Partially committed. The last statement has been executed, but the Tx has not necessarily been committed.
- **Committed**
- **Failed**
  - Aborted = Failed and Rolled Back.
- **Terminated** = Committed or Aborted.

Depending on the type of failure a Tx may either be restarted or killed.

IRON CLAD RULE: No Tx can enter the committed state unless the probability of being aborted is 0.

Another example:

```
Tx_1     Tx_2
read(X,x) read(X,x)
X = x- 50  Time out
write(X,x) Print(x)
Time out finish.
Read(Y,y)
ERROR
```

What do you do?

We'll construct a "rollback" strategy using two primitive operations:
- **undo(Op)** which places the database in state that it was in prior to execution of Op.
- **Redo(Op)** which places the database in the state it was in after Op executed.

It is important that both of these processes be "idempotent", ie the result of executing the process twice is the same as the result of executing once.
The basic tool for constructing a recovery system is a "log file". It must be kept on disk, preferably a disk other than the one on which the data is kept. An active Tx will place various records in this log:

Start(Tx_id) where T_id is a unique Tx id which is assigned by the database. This may be a sequential integer or maybe just the time that it started.

Write(Tx_id,X,o,u) where X is a data item and o and u are respectively the old and new values of x.

Read(Tx_id,X,o)

Commit(Tx_id) which actually means that the Tx is partially committed ( ie it has finished its operations and is eligible to be committed )

Another kind of record may also be placed in the log.

Checkpoint. A checkpoint record GUARANTEES that all changes made since the start of the log file or since the last checkpoint have been permanently entered on the disk.

Recovery Strategies

Recovery from system failure.

Several Possible strategies:

A. Logging with deferred update. A database is updated only after the Tx which has performed the "writes", has written its <Commit, Tx> record to the log file. (A demon process, which reads the log, may actually do the updating).

In case of a system error, the recovery strategy is simple: redo all the Tx's in the log, specifically

1. Read the log backwards making a list of all the Tx's that have a < Commit, Tx> record in the log.
2. Continue reading backwards until you come to the beginning of the log or to a checkpoint. In case you find a checkpoint, you have to continue reading backwards until you find the <Start, Tx> record for all the Tx's on your list.
3. Redo all the Tx's on your list.

NB. If Tx does not have a <Commit, Tx> record, we are guaranteed that the writes in this Tx were never made permanent...it is as if the Tx never happened. Such Tx's may have to be restarted.

Example

log.
<Start, T0>
<Read, T0, A, 1000>
<Read, T0, B, 25>
<Write, T0, A, 1000, 950>
  Crash 1
<Write, T0, B, 25, 75>
<Commit, T0 >
<Start, T1 >
<Read, T1, C, 600>
<Write, T1, C, 600, 500 >
  Crash 2
<Commit, T1 >,
  Crash 3

Crash 1: No recovery possible, restart T0, T1
Crash 2: redo(T0), restart T1
Crash 3: redo(T0), redo(T1)

B. Logging with immediate update...sometime called the write ahead log protocol

A Tx may actually update the database before it partially commits.
Before a Tx makes a change to the database, it must put a "write" record in the log
and the log must be force-output to permanent storage.
A Tx cannot partially commit until all of its reads and writes are in the log, and
the log is on permanent storage.

Recovery strategy

(Single-user Environment)

1. Read the log backwards as before.
2. If the log contains both a <Start, Tx> record and a <Commit, Tx> record, then
   perform redo(Tx)
3. If the log contains only a <Start, Tx> record, perform undo(Tx)

Same example:
Crash 1: Undo(T0)
Crash 2: Redo(T0) and Undo(T1)
Crash 3: Redo(T0) and Redo(T1)

(Multi-user Environment)

1. Make two lists of transactions maintained by the system: the committed
   transactions since the last checkpoint, List_1 and the active transactions,
   List_2
2. If a Tx on List_1 has read a value written by a Tx in our List_2, move it onto
   List_2.
3. Undo all the Tx’s in List_2. The operations should be done in the reverse of the order in which they were written into the log.
4. Redo all the Tx’s in List_1. The operations should be done in the order in which they were written into the log.

This second technique is the one Ingres uses. The log file is called

II_Log_File:[Ingres.Log]dbname.log

The recovery process is called dmfrcp