1 Recovery

Recovery algorithms must take actions during transaction processing to allow recovery should a failure occur. In the event of failure, they must restore database contents to a **consistent** state without compromising transaction properties:

- **atomicity** - a transaction must happen completely or not at all
- **durability** - when a transaction is succesfully completed, no system crash (except HD failure :) results in data loss.

Databases are preferably stored on **stable storage**, such as Redundant Arrays of Independent Disks (**RAID**), which replicate the data and thus greatly reduces the chance of data loss. Data writes to RAIDs are not completed before the data is written in the required number of duplicates.

2 Logging

To ensure **atomicity**, the DBMS writes a log describing the changes to the data, and first when **the log is succesfully written** to stable storage, the changes are performed. In the event of a failure, this allows a **rollback** to the last consistent state before the failure.

The log is a sequence of **log records**:

- An **update log record** has the form

  $$\langle T_i, \text{data item}, \text{old value}, \text{new value} \rangle$$

  , where T_i is the transaction ID, 'data item' is the location on disk of the data item to update, 'old value' is the value of the data item before the write(not used in the 'deferred database modification' scheme), and 'new value' is the value after the write.
- $\langle T_i$ start$\rangle$ signifies that transaction T_i has started
- $\langle T_i$ commit$\rangle$ signifies that transaction T_i has committed
- $\langle T_i$ abort$\rangle$ signifies that transaction T_i has aborted

2.1 Deferred Database Modification

Under the **deferred database modification** recovery scheme, the execution of writes is delayed until the transaction is **partially committed**, i.e. the transaction is completed except for writing the changes to disk.

When a transaction is processed, we start by writing $\langle T_i, \text{start}\rangle$ to the log. When the transaction wants to update values, we write $\langle T_i, \text{data item}, \text{new value}\rangle$ to the log. When the transaction partially commits, we write $\langle T_i, \text{commit}\rangle$ to the log. The log *must* be **flushed** to stable storage **before executing the actual update**.

The following example shows transaction T_0 that transfers 100 from data item A to data item B, and transaction T_1 that adds 1000 to data item C. A, B, and C all have 1000 to begin with. The corresponding database log is shown on the right:

```
T_0: read(A)          <T_0 start>
     A -= 100         <T_0,A,900>
     write(A)         <T_0,B,1100>
     read(B)          <T_0 commit>
     B += 100
     write(B)

T_1: read(C)          <T_1 start>
     C += 1000        <T_1,C,2000>
     write(C)         <T_1 commit>
```

When **recovering** from a failure, the recovery algorithm **scans** the log for transactions that contain both the '**start**' and '**commit**' records. These are **redone**. The redo operation must be **idempotent**, i.e. multiple redo's must be equivalent to one, otherwise a failure *during the recovery* may break consistency.

2.2 Immediate Database Modification

Under the **immediate database modification** recovery scheme, database modifications are allowed during active transactions.

When a transaction is processed, we start by writing $\langle T_i, \text{start}\rangle$ to the log. When the transaction wants to update values, we write $\langle T_i, \text{data item}, \text{old value}, \text{new value}\rangle$ to the log, **and then update on disk**. When the transaction partially commits, we write $\langle T_i, \text{commit}\rangle$ to the log.

When recovering from a failure, the recovery algorithm scans the log and perform 2 different actions:

- if a transaction in the log has the 'start' record but not the 'commit' record, the transaction is **undone**, i.e. the data items are set to 'old value'
- if a transaction in the log has both the 'start' and the 'commit' records, the transaction is **redone**, i.e. the data items are set to 'new value'

2.3 Checkpoints

Of course, the entire database log should not be replayed each time the DBMS restarts, so the DBS periodically does the following:

1. flush all **log records** to stable storage,
2. flush all **modified buffer blocks**, and
3. output ⟨checkpoint⟩ to the log.

No transactions are allowed to update while a checkpoint is being created. In the event of a failure, all the log records before a checkpoint are ignored, as they are already on disk.

3 Buffer Management

To reduce disk access, **logs are buffered** before output to disk. This imposes additional requirements on the recovery algorithm:

- a transaction may not enter the 'commit' state, before the 'commit' record is on stable storage,
- log messages must be output **sequentially** to disk, and
- **logs** must be written to disk **before data**.