RECOVERY SYSTEM

* 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 system must also provide high availability i.e. it must minimize the time for which the database is not usuable after a crash.

Failure Classification

* The simplest type of failure is one that does not result in the loss of information in the system.
* The failures that are more difficult to deal with are those that result in loss of information
* We consider the following types of failure:

1) Transaction Failure

There are two types of errors that may cause a transaction to fail:

(i) Logical Error

Transaction cannot complete due to some internal error (wrong input, data not found, overflow, etc.)

(ii) System Error

The system has entered an undesirable state (for eg: deadlock), as a result of which transaction cannot continue with its normal execution.

2) System Crash

There is a hardware malfunction or a bug in the database software or the o/s that causes the loss of the content of volatile storage, and brings transaction processing to a halt.

3) Disk failure

Any kind of disk failure destroys all or part of disk storage.

Recovery and Atomicity

* Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
* We consider the transaction T_i that transfers $50 from account A to B. The goal is either to perform all database modifications made by a transaction Ti or none at all. (atomicity)
* We assume that the transactions run serially, i.e one after the other.
* To ensure recovery despite failures, we first output information describing the modifications to stable storage (log records) without modifying the database itself.

Log-Based Recovery

* The most widely used structure for recording database modifications is the log.
* The log is a sequence of log records, that records 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 the following fields:
  - Transaction Identifier: Unique identifier of the transaction that performed the write operation.
  - Data-Item Identifier: Unique identifier of the data item written.
  - Old Value: Value of the data item prior to write
  - New Value: Value of the data item will have after the write.

* The various types of log records are denoted as:
  - $\langle T_i, \text{start} \rangle$: Transaction T_i has started
  - $\langle T_i, X_j, V1, V2 \rangle$: Transaction T_i has performed a write on data item X_j. X_j has value V1 before the write and will have a value V2 after the write. (Update log Record)
  - $\langle T_i, \text{commit} \rangle$: Transaction T_i has committed.
  - $\langle T_i, \text{abort} \rangle$: Transaction T_i has aborted.
* Whenever a transaction performs a write, it is essential that the log record for that write to be created before the database is modified.
* Also, we have the ability to undo the modification that has already been done in the database by using the old-value field of the log records.
* For the log records to be useful for recovery from system and disk failure, the log must reside in stable storage.
* The log contains a complete record of all database activity. As a result, the volume of data stored in the log may be unreasonably large.

Log-Based Recovery Approaches

There are two recovery approaches using logs:

1) Deferred Database Modification
2) Immediate Database Modification

1) Deferred Database Modification

* The deferred modification technique ensures the transaction atomicity by recording all database modifications in the log, but deferring the execution of all write operations of a transaction until the transaction partially commits (final action has been executed)
* When transaction Ti partially commits, the records associated with it in the log are used in executing the deferred writes.
* Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written to stable storage. Once the log records have been written, the actual updating takes place and the transaction enters the committed state.
* Only the new value of the data item is required by the deferred modification technique. The update-log-record structure can be ~~omitted~~ updated by omitting the old-value field.

transaction partially commit
↓
log records written to stable storage
↓
write operations of transaction
↓
transaction commits

Example

We consider the example of the banking system. Let To be a transaction that transfers $50 from account A to account B:

```
To: read(A);
    A := A - 50;
    write(A);
    read(B);
    B := B + 50;
    write(B)
```

Let T1 be a transaction that withdraws $100 from account C:

```
T1: read(C);
    C := C - 100;
    write(C)
```

Suppose the transactions are executed serially, in the order To followed by T1 and the initial balance in the accounts be A = $1000, B = $2000 and C = $700 respectively.

State of the database and the log corresponding to To and T1 are:

log	Database
<To, start>	
<To, A, 950>	
<To, B, 2050>	
<To commit>	A = 950 B = 2050
<T1 start>	
<T1, C, 600>	
<T1 commit>	C = 600

* The recovery scheme uses the following recovery procedure:

- redo (T_i) sets the value of all data items updated by transaction T_i to the new values

* After a failure, the recovery subsystem consults the log to determine which transactions needs to be redone.

* Transaction T_i needs to be redone if and only if and only if the log ~~const~~ contains both the records $\langle T_i, start \rangle$ and the record $\langle T_i . commit \rangle$

* If the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction has completed.

State of log at three different times:

$\langle T_0$ start$\rangle$ $\langle T_0, A, 950\rangle$ $\langle T_0, B, 2050\rangle$ ←	$\langle T_0$ start$\rangle$ $\langle T_0, A, 950\rangle$ $\langle T_0, B, 2050\rangle$ $\langle T_0$, commit$\rangle$ $\langle T_1$, start$\rangle$ $\langle T_1, C, 600\rangle$ ←	$\langle T_0$ start$\rangle$ $\langle T_0, A, 950\rangle$ $\langle T_0, B, 2050\rangle$ $\langle T_0$ commit$\rangle$ $\langle T_1$ start$\rangle$ $\langle T_1, C, 600\rangle$ $\langle T_1$ commit$\rangle$ ←
- Crash occurs after the step write(B) - No redo action needed as no commit record appears in the log. - Log record of incomplete transaction T_0 is deleted from log. - A = \$1000 B = \$2000	- Crash occurs just after the step write(C) - The operation redo(T_0) is performed as $\langle T_0$, commit$\rangle$ appears in the log. - A = \$950 B = \$2050 - Log of T_1 is deleted.	- Crash occurs after $\langle T_1$ commit$\rangle$ - Two commit records are there in the log - one for T_0 and one for T_1. - redo(T_0) and redo(T_1) are performed. - A = \$950 C = \$600 B = \$2050

2) Immediate Database Modifications

* The immediate modification technique allows database modifications to be output to the database while the transaction is still in the active state.
* Data modifications done by active transactions are called uncommitted ~~transactions~~ modifications.
* In the event of a crash or transaction failure, the system must use the old-value field of the log records to restore the modified data items to the value they had prior to the start of the transaction.
* The information in the log is used in reconstructing the state of the database. So, the actual update in the database in not allowed to take place before the corresponding log record is written to stable storage.

Example

* We reconsider the simplified banking system, with transactions T0 and T1 executed one after the other in the order T0 followed by T1.
* The portion of the log corresponding to transactions T0 and T1 are:

<T0 start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
<T0 commit>
<T1 start>
<T1, C, 700, 600>
<T1 commit>

State of the system log and database corresponding to T0 and T1:

Log	Database
<T0 start>	
<T0, A, 1000, 950>	
<T0, B, 2000, 2050>	A = 950 B = 2050
<T0 commit>	
<T1 start>	
<T1, C, 700, 600>	C = 600
<T1 commit>	

* The immediate database modification technique uses two recovery procedures:
  - undo(Ti): restores the value of all data items updated by transaction (Ti) to the old values
  - redo(Ti): sets the value of all data items updated by transaction Ti to the new values
* After a failure has occurred, the recovery scheme consults the log to determine which transactions needs to be redone and which needs to be undone:
  - Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain the record <Ti commit>
  - Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>

Same log shown at three different times

(I)	(II)	(III)
<To start>	<To start>	<To start>
<To, A, 1000, 950>	<To, A, 1000, 950>	<To, A, 1000, 950>
<To, B, 2000, 2050>	<To, B, 2000, 2050>	<To, B, 2000, 2050>
	<To commit>	<To commit>
	<T1 start>	<T1 start>
	<T1, C, 700, 600>	<T1, C, 700, 600>
		<T1 commit>

Initially,
A = $1000
B = $2000
C = $700

Case I: Suppose the crash occurs just after the step write(B) of To. Since in the log, <To start> is there but no corresponding <To commit> record, To must be undone, so undo(To) is performed.

∴ A = $1000, B = $2000

Case II: Suppose the crash occurs just after the step write(C) of T1. In this case, two recovery actions need to be taken: redo(To) as the log contains <To start> and <To commit> and undo(T1) as <T1 commit> is not in the log.

∴ A = $950, B = $2050, C = $700

NOTE: undo(T1) is performed before the redo(To). This is important for the recovery algorithm.

Case III: Suppose the crash occurs just after the step <T1 commit>. In this case, both To and T1 needs to be redone, redo(To) and redo(T1).

∴ A = $950, B = $2050, C = $600

Checkpoints

* When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone.
* We need to search the entire log to determine this information.
  - There are two major difficulties with this approach:
    - The search process is time-consuming.
    - The transactions that needs to be redone have already written their updates into the database. Redoing transactions causes no harm, but causes recovery to take longer time.
* To reduce these types of overhead, we introduce checkpoints.
* During execution, the system maintains the log using deferred database modification or immediate database modification technique
* In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

Periodically saves the system state onto a stable storage disk

1) Output onto stable storage all log records currently residing in main memory
2) Output to the disk all modified buffer blocks
3) Output onto stable storage a log record <checkpoint>

* Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

* After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place.

* Such a transaction can be found by searching the log backward from the end of the log, until it finds the first <checkpoint> record (final <checkpoint> record as we are searching back from the end of the log), then it continues the search backward until it find a (T_i start) record.

* Once the system has identified transaction T_i, the redo and undo operations need to be applied to only transaction T_i and all transactions T_j that started executing after T_i.

* The remainder of the log can be ignored.

Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing checkpointing
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage.

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent **<checkpoint>** record
  2. Continue scanning backwards till a record $<T_i$ **start**> is found.
  3. Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no $<T_i$ **commit**>, execute **undo**(T_i). (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit**>, execute **redo**(T_i).

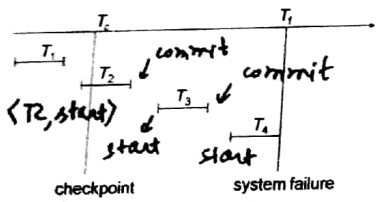
Checkpoints

- Problems in recovery procedure as discussed earlier :
  - searching the entire log is time-consuming
  - we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing checkpointing
  - Output all log records currently residing in main memory onto stable storage.
  - Output all modified buffer blocks to the disk.
  - Write a log record < **checkpoint**> onto stable storage.

Checkpoints (Cont.)

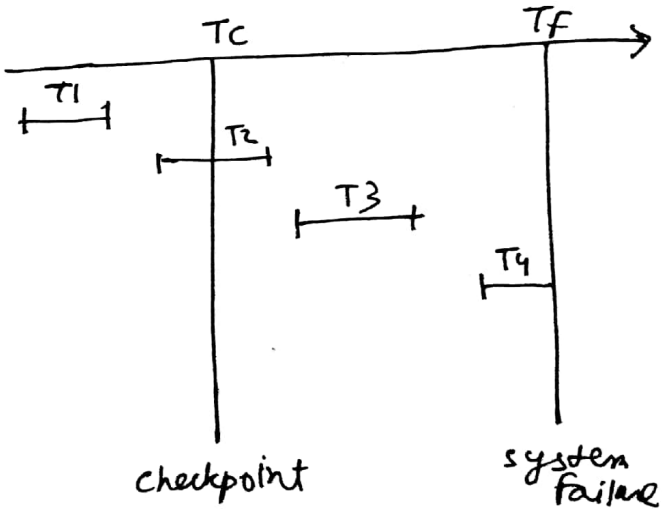
- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint**> record
  - Continue scanning backwards till a record $<T_i$ **start**> is found.
  - Need only consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  - For all transactions (starting from T_i or later) with no $<T_i$, **commit**>, execute **undo**(T_i). (Done only in case of immediate modification.)
  - Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$, **commit**>, execute **redo**(T_i).

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Example



- T1 can be ignored as updates already o/p to disk due to checkpoint
- T2 and T3 are redone
- T4 undone