## Primitive DB Operations of Transactions

- `INPUT(X)` = copy the disk block containing database element $X$ to a memory buffer.
- `READ(X,t)` = if the block containing database element $X$ is not in a memory buffer then `INPUT(X)`. Next, assign the value of $X$ to local variable $t$.
- `WRITE(X,t)` = if the block containing database element $X$ is not in a memory buffer then `INPUT(X)`. Next, copy the value of $t$ to $X$ in the buffer.
- `OUTPUT(X)` = copy the buffer containing $X$ to disk.

**Example**

- $A$, $B$ are database values; constraint $A = B$ must hold.
- Transaction $T =$

```
A := A*2;
B := B*2;
```

- Execution of $T$ involves reading $A$, $B$ from disk, performing arithmetic in memory, and writing new $A$, $B$ to disk.

| Action | $t$ | Mem $A$ | Mem $B$ | Disk $A$ | Disk $B$ |
|---|---|---|---|---|---|
| `READ(A,t)` | 8 | 8 | | 8 | 8 |
| `t := t*2` | 16 | 8 | | 8 | 8 |
| `WRITE(A,t)` | 16 | 16 | | 8 | 8 |
| `READ(B,t)` | 8 | 16 | 8 | 8 | 8 |
| `t := t*2` | 16 | 16 | 8 | 8 | 8 |
| `WRITE(B,t)` | 16 | 16 | 16 | 8 | 8 |
| `OUTPUT(A)` | 16 | 16 | 16 | 16 | 8 |
| `OUTPUT(B)` | 16 | 16 | 16 | 16 | 16 |

- Problem: what happens if there is a system failure just before `OUTPUT(B)`?

## Undo Logging

Create a *log* of all important actions (due to Hansel and Gretel, 782 AD; improved to *durable* undo logging in 784).

- `<START` $T$`>` = transaction $T$ started.
- <$T$,$X$,$v$> = database element $X$ was modified; it used to have value $v$.
- `<COMMIT` $T$`>` = transaction $T$ has completed, and all its changes have been output to the database.

## Intention

If there is a crash before transaction finishes, the log will tell us how to restore old values for any DB elements changed on disk.

## Difficulties

- If the log isn't on disk, it too can be lost.
- If we have to write every log entry to disk, we do a *lot* of disk I/O.

## Undo (Write-Ahead) Logging

- Create a log record for every action.
- Log records for DB element $X$ must be on disk (or other nonvolatile storage) *before* any database modification to $X$ appears on disk.
- Before commit record appears on disk, all database modifications of the transaction must appear on disk.
  - ✦ *Flush log* = write any log entries to disk if they are not already there.

## Example

| `Action` | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | Log |
|---|---|---|---|---|---|---|
| `READ(A,t)` | 8 | 8 | | 8 | 8 | `<START` $T$`>` |
| `t := t*2` | 16 | 8 | | 8 | 8 | |
| `WRITE(A,t)` | 16 | 16 | | 8 | 8 | <$T, A, 8$> |
| `READ(B,t)` | 8 | 16 | 8 | 8 | 8 | |
| `t := t*2` | 16 | 16 | 8 | 8 | 8 | |
| `WRITE(B,t)` | 16 | 16 | 16 | 8 | 8 | <$T, B, 8$> |
| `FLUSH LOG` | | | | | | |
| `OUTPUT(A)` | 16 | 16 | 16 | 16 | 8 | |
| `OUTPUT(B)` | 16 | 16 | 16 | 16 | 16 | `<COMMIT` $T$`>` |
| `FLUSH LOG` | | | | | | |

## Abort Actions

Sometimes a transaction cannot complete, e.g.:

1. It detects an error condition such as faulty data.
2. It gets involved in a deadlock, competing for resources or data with other transactions.

If so, the transaction *aborts*; it does not write any of its DB modifications to disk, and it issues an `<ABORT` $T$`>` record to the log.

## Recovery With Undo Logging

Suppose there is a system crash, say just before `OUTPUT(B)`. Do the following:

1. Examine the log to identify all transactions $T$ such that `<START` $T$`>` appears in the log, but neither `<COMMIT` $T$`>` nor `<ABORT` $T$`>` does.

   ✦ Call such transactions *incomplete.*

2. Examine each log entry `<`$T, X, v$`>` from most recent to earliest.

   a) If $T$ is not an incomplete transaction, do nothing.

   b) If $T$ is incomplete, do `WRITE(X,v); OUTPUT(X)`.

3. For each incomplete transaction $T$ add `<ABORT` $T$`>` to the log, and flush the log.

## Checkpointing

Problem: in principle recovery requires looking at entire log. Simple solution: occasional *checkpoint* operation during which we:

1. Stop accepting new transactions.
2. Wait until all current transactions commit or abort.
3. Flush log to disk and all memory buffers to disk.
   - ✦ Should have occurred anyway in common log methods.
4. Enter a `<CHECKPOINT>` record in the log and flush to disk.

At this point, transactions may resume.

- If recovery is necessary, we know that all transactions prior to a recorded checkpoint have committed and need not be undone.

## Nonquiescent Checkpointing

Problem: we may not want to stop transactions from entering system. Solution:

1. Write `<START CKPT(`$T_1, \ldots, T_k$`)>` record to log, where $T_i$'s are all active transactions.

2. Allow active transactions to commit, but do not prohibit new transactions.

3. Write `END CKPT>` record to log.

## Recovery With Nonquiescent Checkpoints

- If the crash follows `<END CKPT>` we can restrict ourselves to transactions that began after the `<START CKPT>`.

- If the crash occurs between `<START CKPT>` and `<END CKPT>`, we need to undo

  1. All those transactions $T$ with `<START` $T$`>` after the `<START CKPT>` but but no `<COMMIT` $T$`>`.

  2. All transactions $T$ on the list associated with `<START CKPT>` with no `<COMMIT` $T$`>`.

## Redo Logging

- Commit before writing data to disk.
- Redo-log entries contain *new* values:
  - ✦ `<T,X,v>` = "transaction $T$ modified $X$ and the new value is $v$."

## Redo Logging Rules

1. Generate new-value log entry whenever an element is modified (in buffer).
2. Before modifying DB element $X$ on disk, transaction must be committed, and `COMMIT` record written to log.
3. Before modifying DB element $X$ on disk, flush all log entries involving $X$ (including commit) to disk.

## Example

| Action | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | Log |
|---|---|---|---|---|---|---|
| READ(A,t) | 8 | 8 | | 8 | 8 | <START $T$> |
| t := t*2 | 16 | 8 | | 8 | 8 | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <$T, A, 16$> |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | |
| t := t*2 | 16 | 16 | 8 | 8 | 8 | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <$T, B, 16$> |
| | | | | | | <COMMIT $T$> |
| FLUSH LOG | | | | | | |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | |
| FLUSH LOG | | | | | | |

## Recovery for Redo Logging

1. Find set of committed transactions from the log.
   - ✦ Look back to previous checkpoint only.
2. Examine log forward, from earliest to latest. For each <$T, X, v$> in log:

   ```
   WRITE(X,v);
   OUTPUT(X);
   ```

- Notice that no uncommitted transaction can have any effect on the DB.

## Problem

If we use nonquiescent checkpointing with redo logging, how do we simplify recovery.

- Hint: If a transaction doesn't make the "active" list at `START CKPT`, then it not only has committed, but all its changes have been written to the DB's disk.

## Undo/Redo Logging

Problem: both previous methods have some downside:

- Redo requires keeping all modified blocks buffered until after commit.
- Undo can lose effects of transaction that appear (to the user) to have completed.

## Undo/Redo Log

Log entries <$T, X, v, w$>, means transaction $T$ updated DB element $X$ from old value $v$ to new value $w$.

## Undo/Redo Rules

1. Generate a new/old record on the log whenever a DB element is modified (in buffer).
2. Flush the log before updating $X$ on disk.
3. Flush log after writing a <COMMIT $T$> record.

- But there is no constraint about whether DB elements are flushed to disk before or after commit point.

## Example

One possibility:

| `Action` | $t$ | M-$A$ | M-$B$ | D-$A$ | D-$B$ | `Log` |
|---|---|---|---|---|---|---|
| `READ(A,t)` | 8 | 8 | | 8 | 8 | `<START` $T$`>` |
| `t := t*2` | 16 | 8 | | 8 | 8 | |
| `WRITE(A,t)` | 16 | 16 | | 8 | 8 | <$T, A, 8, 16$> |
| `READ(B,t)` | 8 | 16 | 8 | 8 | 8 | |
| `t := t*2` | 16 | 16 | 8 | 8 | 8 | |
| `WRITE(B,t)` | 16 | 16 | 16 | 8 | 8 | <$T, B, 8, 16$> |
| `FLUSH LOG` | | | | | | |
| `OUTPUT(A)` | 16 | 16 | 16 | 16 | 8 | `<COMMIT` $T$`>` |
| `OUTPUT(B)` | 16 | 16 | 16 | 16 | 16 | |
| `FLUSH LOG` | | | | | | |

## Redo/Undo Recovery

1. Find set of problematic transactions:
   - ✦ Go back to previous checkpoint; include all that either started after the checkpoint began or are on the "active" list at `START CKPT`.
2. If a transaction has no `COMMIT` record, undo it.
   - ✦ Must proceed latest to earliest.
3. If the transaction has a `COMMIT` record, redo it.
   - ✦ Must proceed earliest to latest.

## Idempotence

An operation is *idempotent* if the result of repeating it several times is the same as doing it once.

- Example: $f(x)$ defined by "execute `x := x+1`" is not idempotent; $f; f$ does not have the same effect as $f$.
- Example: $g(x)$ defined by "execute `x := 10` is idempotent; $g; g$ has exactly the same effect as $g$.
  - ✦ Thus, the recovery steps recommended for undo, redo, and undo/redo logging are all idempotent.

## Problem

What if the transaction involves an inherently nonidempotent operation, such as spitting out cash from an ATM?

- How would you log withdrawals, and how would you recover in a situation where "spit out cash" can be neither undone nor redone?