Clarification

- The log entry $[T_i.x,v]$ contains the identity of the transaction ($T_i$), the identity of the data item being written to ($x$), and the data being written into $x$ ($v$).
- I.e. we store only **after-images** in the log.
- Before images are determined by scanning the log backwards and finding
  - the first entry for $x$ made by a committed txn, or
  - the initial value of $x$ (that we assume is saved at the beginning of the log).
Implementation

• Since only a small part of a page may actually be changed, we store only the change and the offset – this is called partial data item logging.
• The abort, commit, and active lists are incorporated into the log.
• With smaller entries, writing each log entry to stable storage is inefficient ➔ buffer log entries in main memory.
• This improves I/O performance but potentially violates the undo and redo rules.

Redo Rule

• Redo rule: txn may commit without having its changes or the log entries on stable storage!
• Simple fix: flush buffer before commitment (after adding $T_i$ to commit list in log).
• This may oppose the use of the buffer – we may flush when the buffer is largely empty.
• Solution: Delay commits until buffer is reasonably full.
• Also called Group commit.
Undo Rule

• Undo rule can be violated if an updated page in the cache is flushed (to the stable DB) before the corresponding log records are flushed (to stable log).
• Trickier to solve.
• Associate with
  – each log entry, a log sequence number (LSN); and
  – cache slot, an LSN field.
• The LSN of a log entry uniquely identifies it in the log. LSN always increase as time passes.

Undo Rule (contd.)

• When a cache slot is written to, the LSN filed is updated to the LSN of the log entry that is writing before the slot is unpinned.
• Before the CM flushes a cache slot, it ensures that the log upto and including the slot’s LSN is stable.
Details of Log Records - Update

• Update:
  – The name of the txn that is writing
  – The name of the data item written
  – The offset, and length of the portion of data updated
  – The old value of the portion of data item that was updated (before-image)
  – The new value of the portion of data item that was updated (after-image)
  – A pointer (LSN) to the previous update record for this txn.

Details of Log Records

• Commit: just contains the ID of the committing txn.
• Abort: just contains the ID of the aborting txn.
• Checkpoint (Assume Fuzzy checkpointing): indicates the completion of a checkpoint (CKPT):
  – A list of active txns at the time of CKPT,
  – A list of the data items that were in dirty cache slots, along with the stable-LSNs of these slots, at the time of the CKPT.
Stable LSNs

• **Stable-LSN**:  
  – associated with each cache slot (like the LSN).  
  – this is the LSN value of the last record in the log buffer when the cache slot was last fetched or flushed.  
  
• It represents a point in time (LSN effectively measure time for us), when the cache slot, and its copy on stable storage are guaranteed to hold the same values.  
• Also, the stable copy must contains all updates upto the stable-LSN.

Checkpoint Procedure

1. Stop RM from processing operations; wait for active operations to finish.  
2. Flush all dirty cache slots that have not be flushed since the previous checkpoint (CKPT’). Flush those slots whose stable-LSN is smaller than the LSN of the previous checkpoint record, update the stable-LSN accordingly.  
3. Create a CKPT record and add to log.  
4. Ack the end of CKPT, and start processing operations.
Restart

- **Two passes**: back scan – undo; fwd scan – redo.
- Begin **backward scan**.
- Scan backwards from end of log; $CL = \{\}, \ AL = \{\}$
- If commit (abort) add $T_i$ to $CL$ ($AL$).
- If update $[T_i, x, v]$:
  - If $T_i$ is in $CL$; ignore
  - If $T_i$ not in $CL$ or $AL$, add to $AL$.
  - If $T_i$ now in $AL$, restore $x$’s before-image; if this is the first update by $T_i$ (prev LSN is null), then remove $T_i$ from $AL$. 

No slots to be flushed – write CKPT record

**History**: $w_1[a] w_1[x] w_2[b] w_3[c] w_4[d] a \ c_2 CKPT w_3[e] w_4[f] a_3 w_4[x] w_4[c] CKPT$

**Stable Log**: 

<table>
<thead>
<tr>
<th>LSN</th>
<th>Log Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0</td>
</tr>
<tr>
<td>$b$</td>
<td>1</td>
</tr>
<tr>
<td>$c$</td>
<td>1</td>
</tr>
<tr>
<td>$d$</td>
<td>1</td>
</tr>
<tr>
<td>$e$</td>
<td>0</td>
</tr>
<tr>
<td>$f$</td>
<td>1</td>
</tr>
<tr>
<td>$x$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Cache**:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Dirty</th>
<th>LSN</th>
<th>S-LSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>$x$</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>$c$</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>10</td>
</tr>
</tbody>
</table>
Restart (contd.)

• If checkpoint record:
  – If this is the first CKPT record, ignore
  – Upon reading CKPT’ (penultimate):
    • Examine the list of active txns stored in CKPT’ – add any of these not already in AL or CL to AL.
    • Continue backwards, ignoring all records except updates of txns in AL. For each of these restore the before-image; if this is the first update for that txn, remove it from AL.
    • Stop when AL is empty.

This is the end of the backward scan.

Restart (contd.)

• After the end of the backward scan, we know that
  – For all items whose last committed value was written before CKPT’, x has the after-image of that write.
  – Before image of all aborted txns have been restored.
Restart (contd.)

• The forward scan begins at CKPT’.
• For each update of a committed txn, restore the after-image in the cache.
• Update records of txns not in CL are ignored.
• At the end of the log, the DB is in a stable state.
• Restart is idempotent.

Optimizations

• To reduce the amount of work restart has to do, we can avoid certain undo/redo operations.
• During backward scan for \([T_i,x,v]\) in AL, need not undo this operation if:
  – A1: \(T_i\)’s abort lies between CKPT and CKPT’, but \(x\) is not among the dirty items at CKPT.
  – A2: \(T_i\)’s abort record lies between CKPT and CKPT’, and \(x\) was in a dirty cache slot at CKPT, but its stable-LSN (saved in CKPT) is greater than the LSN of \(T_i\)’s abort record.
Optimizations

- In the forward scan, $[T_p, x, v]$, where $T_i$ is in $CL$, need not be redone if:
  - C1: $T_i$’s update record lies between CKPT and CKPT’, but $x$ is not in the list of dirty cache slots at CKPT; or
  - C2: $T_i$’s update record lies between CKPT and CKPT’, $x$ is in the list of data items in dirty cache slots at CKPT, but its stable-$LSN$ is greater than the $LSN$ of the update record at hand.

- RM can improve performance in case of multiple failures by appending two CKPT records at the

Logical Logging

- Logical logging can significantly reduce the amount of storage needed for the log, e.g. *add entry 5 to B-tree, adding a record to a file.*
- We must be able to log, undo, and redo each logical operation.
- However, *multiple undo/redo* of logical operations are not equivalent – we must be careful!!
- Could be solved by implementing *undo/redo* such that they are *idempotent* – not always possible.
Alternative 2

• Another alternative is to save a copy of the stable DB at the last checkpoint.
• Restart essentially replays operations from the checkpoint:
  – It begins with the checkpoint DB state, and
    • Undoes all updates that precede the CKPT, but were by txns that were active at CKPT and did not commit;
    • Redoes update records that follow CKPT by committed txns.
  – From strictness this is exactly what we want.
• IBM system R used this with shadowing.

Alternative 3

• Another solution is to save LSNs in the stable DB.
• Each data item on stable DB saves the LSN of the last update applied to it by an active or committed txn.
• For performance, we chain back the updates for each data item in the log (as well as for each txn).
• New algo: LSN-based logical logging.
• Assume logical logging with fuzzy checkpointing; strict executions;
LSN-based Logical Logging

- **RM-Write:**
  - Create an update record, $U$
  - save current $LSN(x)$ in $U$
  - Update $x$, set $LSN(x) \leftarrow LSN(U)$.

- **RM-Abort:**
  - Upon undo for record $U$ restore $LSN(x) \leftarrow prev LSN(x)$ saved in $U$.

- Restart scans back from the end of the log for undo, and fwd from CKPT’ to redo, as before.

---

Backward Scan

- In backward scan of restart, when dealing with an update record $U$ by $T_i$ (aborted) for $x$:
  - Fetch $x$ and examine $LSN(x)$
  - If $LSN(x)=LSN(U)$ ($x$ is in the same state after $U$ was applied), then undo $U$, set $LSN(x)$ to the value stored in $U$.
  - If $LSN(x) < LSN(U)$ ($x$ does not contain $U$’s update) – do not undo $U$.
  - If $LSN(x) > LSN(U)$: $x$ contains a later update ($V$) – which was not undone, so $V$ follows $U$ and must have committed (o/w $LSN(x)$ must have been set to $LSN(U)$ as above). By strictness, $U$ must have been undone before $V$ was applied, thus no need to undo $U$. 
Forward Scan

- Backward scan ends as with physical logging.
- Forward scan begins at CKPT’, and processes each update record $U$ of committed txns.
  - If $LSN(x) < LSN(U)$, then $U$ hasn’t been applied \(\rightarrow\) redo $U$
  - If $LSN(x) = LSN(U)$, then $U$ has been applied, need not redo.
  - If $LSN(x) > LSN(U)$, then a later committed (cannot be an aborted or active txn – why?) update has been applied, so no need to redo.
- $LSNs$ in stable DB can help with physical logging too.