Checkpointing, Redo, Undo/Redo Logging

CS157B

Chris Pollett

Apr.20, 2005.

Outline

- Checkpointing
- Redo Logging
- Undo/redo Logging

Checkpointing

- So far recovery requires that the entire log file be looked at.
- Even if a transaction has written a COMMIT to the log one might still need to look at its operations during a recovery because several other transaction might be running at the same time.
- To simplify the issue we can *checkpoint* the log periodically.
- This involves doing the following:
  1. Stop accepting new transactions
  2. Wait until all current transactions commit or abort and have written the COMMIT or ABORT to the log.
  3. Flush the log to disk.
  4. Write a log record <CKPT> and flush the log again.
  5. Resume accepting transactions.
- If checkpoints are used then there is no need to look in the log file prior to the last checkpoint.

Nonquiescent Checkpointing

- The problem with checkpointing is that we effectively stop the system until all current operations have committed or aborted.
- *Nonquiescent checkpointing* is a technique that avoids this bottleneck. The to do this steps are to:
  1. Write a record <START CKPT(T1…Tk)> and flush log. Here T1,…Tk are the active transactions.
  2. Wait until all of T1, … Tk commit or abort, but allow new transactions to start.
  3. When all of T1,…Tk have written COMMIT or ABORT then write <END CKPT> to the log.
- If this kind of checkpoint is being used and a crash occurs, then we look backwards through the log for the first <START CKPT(T1…Tk)> or <END CKPT>. If we see an <END CKPT>, we know we only have to consider after this. If we see a <START CKPT(T1…Tk)> but no <END CKPT>, then we need to only consider after the transactions T1, .. Tk began.

Redo Logging

- A problem with undo logging is that we cannot commit a transaction without first writing all its changed data to disk. This might cost us I/Os.
- This requirement can be avoided using redo logging.
- The principle differences between undo and redo logging are:

1. Undo logging cancels the effects of incomplete transactions and ignores committed ones; redo logging ignores incomplete transactions and redoes committed ones as necessary.
2. Undo logging requires changed DB elements to be written to disk prior to the commit log record being written to disk; redo logging requires the COMMIT appear on disk before any changed values reach the disk.
3. We had rules U1 and U2 for undo logging to guarantee undo logging worked. We will have a redo log rule R1 which replaces these two rules.

The Redo Logging Rule

(R1) [Also called the write-ahead logging (WAL) rule] Before modifying X on disk, all log records pertaining to this modification of X (that is, both the update record <T, X, v> and the <COMMIT T> record) must appear on disk.

- Here v is now the new value not the old value as in undo logging.

Recovery with Redo Logging

- To recover when using redo logging, we:
  1. Identify the committed transactions.
  2. Scan the log forward from the beginning. For each log record <T, X,v> encountered:
     a) If T is not a committed transaction, do nothing.
     b) If T committed, write the value for database element X.
  3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

Checkpointing a Redo Log

- Checkpointing can also be done with redo logs. The steps are:
  1. Write a log record <START CKPT (T1,…Tk)>, where T1,…Tk are the active transactions.
  2. Write to disk all database elements that were written to buffer but not yet committed when the START CKPT began.
  3. Write an <END CKPT> record.
- To recover with a checkpointed redo log, look for last record of type <END CKPT> .
  – Only need to redo Ti's or transaction commited after the corresponding <START CKPT>.

Undo/Redo Logging

- Both undo and redo logging have disadvantages.
- An example disadvantage of redo logging is that all modified blocks must be kept in buffers until the transaction commits and the log records have been flushed.
- Both cause problems if one has a block in memory that was modified by a transaction that is ongoing and modified by a transaction that has committed.
- We can try to combine the best of both worlds and use undo/redo logging.
- In this set up we have the following new rules:

(UR1) Before modifying X on disk because of T, it is necessary that the update record <T,X,v,w> appear on disk. (v is old value; w is new)

(UR2) A <COMMIT T> must be flushed to disk as soon as it appears in the log.

- To recover when using undo/redo logging, we:

1. Redo all the committed transactions in the order earliest first, and
2. Undo all the incomplete transactions in the order latest first.

Checkpointing an Undo/Redo Log

- The steps this time are:
  1. Write a <START CKPT(T1, …Tk)> to the log
  2. Write all dirty buffers to disk.
  3. Write an <END CKPT>, flush the log.
- We require a transaction not write any values even to memory buffers until it is certain whether or not to abort.