Transactional Recovery

Introduction

Data in a database resides on disk

Memory:
- Volatile storage
- Nonvolatile storage

Buffer disk pages are read into memory buffer(s) so the data will be available for random access by processes.

The system makes attempt to keep a page in the buffer in the hope that it will be referenced again in the near future – it reduces the number of I/O operations.

To support buffering - **lookaside approach**

- Disk page read request – hash the page address to any entry in the lookaside table:
- If the entry is found – the page is already in the buffer
- If not – it must be read from disk

LRU buffering (least recently used): database pages remain in buffer until a new page being read in from disk requires space, and all buffer pages are occupied.

Some page must be dropped from the buffer among those that has not been referenced for the longest time (LRU)

We also don't want to have to write out page back to disk every time it is updated by some transaction

This is one of the most important optimizations for transactional applications

Conclusion - popular pages remain in buffer until:

- They become less popular and drift out of buffer because of LRU
- The system forces them to be written back to disk after some period of time

A page in a buffer is said to be *dirty* if it has been updated by some transaction since the last time it was written back to disk.

Dirty pages remain in buffer long after the transaction that dirtied them has committed.

Problem: suppose we suddenly lose power or have a system crash. Some of the disk pages are out of date because they were very popular and they haven't been written out from buffer during the last thousand updates that took places. All these updates existed only in memory!

How can we handle this problem and be able to recover these lost updates without going back to the approach of writing out every update as soon as it happens?

As a tuple update occurs the system write a note, a *log entry*, into a memory area known as a *log buffer*.

At appropriate times the log buffer is written out to disk into
A sequential file called – *log file*.

A log file contains all log entries created for some interval of time into the past. If memory is lost, the recovery process will be able to use the log file to recover updates of tuples that are out of date on disk.

This log method is preferable to writing out each tuple update as it happens because it is much more efficient with respect of I/Os.

Even if all disk page updates were written out to disk as soon as they occurred at no cost, this would not be sufficient to perform recovery!

2 Log Formats

Consider the following execution:

r_1 = T1:r(a, 50) T1:w(a, 20) T2:r(c, 100) T2:w(c,50)
T2:c T1:r(b, 50) **T1:w(b, 80)** T1:c.....

Suppose the system crashed after T1:c. Due to LRU buffering scheme the data items A, B, and C might not be written out to disk in the order in which the updates occur in the system.

a = 50
b = 80
c = 100

data are inconsistent

consistency constraint is: a +b = 100

Conclusion: while we depend only on LRU buffering scheme to write out updates, we cannot expect consistent data to be on disk.

Assume that writes to disk are performed as soon as the page is updated in buffer.

Assume that the crash occurred after the operation T2:c

a = 20
b = 80
c = 50

Data are inconsistent

Conclusion: the system that performs updates to disk as soon as they occur does'nt solve the problem.

Transaction:

- atomicity
- durability

A procedure known as *database recovery* is performed after a crash that uses the log entries written earlier and brings the disk-resident database to a state where it will reflect either all or none of the tuple updates for transactions that were in progress at the time of the crash.

Basic assumption of recovery:
Transactional system, after being restarted following a crash, will never remember the intentions of the transaction logic that was running when memory was lost.

What is the proper consistent state in the case of the system crash?
Transaction T2 committed; T1 rolled back. Thus, the proper consistent state after the crash would be:

a = 50
b = 50
c = 50

Consistent data

Transaction execution and corresponding log entries

Operation	Log entry
T1:r(a, 50)	(S, 1) – start transaction T1 log entry. No log entry is written for a read operation, but this operation is the start of T1
T1:w(a, 20)	(W, 1, a, 50, 20) – T1 write log for update of a. The value 50 is the before image, 20 – is the after image of a
T2:r(c, 100)	(S, 2) – start transaction T2 log entry
T2:w(c, 50)	(W, 2, c, 100, 50) – T2 write log for update of b. The value 100 is the before image, 50 – is the after image of b
T2:c	(C, 2) – commit T2 log entry – **(write log buffer to log file)**
T1:r(b, 50)	No log entry
T1:w(b, 80)	(W, 1, b, 50, 80) – T1 write log for update of b. The value 50 is the before image, 80 – is the after image of b
T1:c	(C, 1) – commit T1 log entry – **(write log buffer to log file)**

Log entries also appear for *insert* and *delete* operations.

The log buffer is written out to the log file under only two circumstances in our scheme:

- when some transaction commits
- when the log buffer becomes too full to hold more entries (double-buffered disk write)

Is there enough data in these log entries to permit the system to recover after a system crash?

Two kinds of problems may occur.

- we may have written to disk page updates of transactions that ever completed (UNDO)
- we may find that some page updates of transactions that have committed never got to disk (REDO)

Assume that a system crash occurs immediately after the operation T1:w(b, 80) has completed ((W, 1, b, 50, 80) has been placed in the log buffer).

The log buffer was written out to disk with the log entry (C, 2). Transaction T2 has committed while transaction T1 has not. After the system is reinitialized and the system operator gives a command that initiates recovery (RESTART command)

It means that all updates performed by T2 must be on disk and all updates performed by T1 must be rolled back on disk. The final values of data items after recovery should be:

a = 50
b = 50
c = 50

Consistent data

The process of recovery takes place in two phases:
- ROLLBACK
- ROLL FORWARD

In the ROLLBACK phase, the entries in the sequential log file are read in reverse order back to system starup, when all data items activity began. During the ROLLBACK step, recovery performs UNDO of all the updates that should not have occurred, because the transactions that made them did not commit.

In the ROLL FORWARD phase, the entries in the sequential log file are read forward again to the last entry. During the ROLL FORWARD step, recovery performs REDO of all the updates that should have occurred, because they were legal updates by transactions that have committed.

Log entry	ROLLBACK action performed
1. (C, 2)	Put T2 into the committed list
2. (W, 2, c, 100, 50)	Since T2 is on the committed list, do nothing
3. (S, 2)	Transaction T2 is no longer active
4. (W, 1, a, 50, 20)	Transaction T1 has never committed. Therefore, the system performs UNDO of this update by writing the before image value (50) into data item a. Put T1 into the uncommitted list
5. (S, 1)	T1 is no longer active. Now that no transactions were active, we can end the ROLLBACK phase

Log entry	ROLL FORWARD action performed
6. (S, 1)	No action required
7. (W, 1, a, 50, 20)	T1 is uncommitted – no action required
8. (S, 2)	No action required
9. (W, 2, c, 100, 50)	Since T2 is on the committed list, we REDO this update by writing after image value (50) into data item c
10. (C, 2)	No action required
11.	We have rolled forward through all log entries and terminate recovery

Comments:

3 Correctness of recovery

How can we be certain that all the log entries needed for proper recovery are out on disk?

- Writes to disk are carried out in an atomic manner - „read after write"
- Transaction commit + write out of the log buffer to the log file – atomic action

Correctness of ROLLBACK i ROLL FORWARD procedures.

1. **ROLL FORWARD – REDO operation**

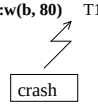
A commit log entry is a trigger for writing out the log buffer and the transaction is not considered to have successfully completed until this log buffer write has been successful. Since the commit log must get out to disk, all earlier write log entries for that transaction are also out to disk, so we are assured that the REDO task can be performed successfully.

2. **ROLLBACK – UNDO operation**

We have to UNDO all updates of transactions that have not committed in the log file. We have not provided any guarantee that all log entries for updates of uncompleted transactions will get out to disk. Could any problems arise as a result of this? YES.

This may happen if the page updated by uncommitted transaction drifted out to disk through the LRU buffering before the log buffer containing the associated log entry was written to the log file.

r_1 = T1:r(a, 50) T1:w(a, 20) T2:r(c, 100) T2:w(c,50)
T2:c T1:r(b, 50) **T1:w(b, 80)** T1:c.....



crash

Two feasible solutions:

Solution A: subvert LRU page replacement policy

All updated pages will not drifted out to disk until transactions that have written on them have committed – dirty pages remain in the buffer until transactions commit. No UNDO processing is ever needed at al during recovery, and therefore before images are not needed in the logs.
Drawback - ?

Solution B: modify LRU page replacement policy

The problem – dirty page drift out to disk through the LRU before the associated log buffer is written out to disk.

The solution - the associated log buffer is written out to disk prior to dirty buffer pages reaching disk.

This is referred to as the **„write ahead log"** (WAL) guarantee.

Transactional system create the so called log sequence number (LSN), a sequentially increasing integer value associated with every entry written into the log buffer. Moreover, the system keeps track of the smallest LSN of any entry in the log buffer that has been created since the log buffer was last written out to disk – we denote it LSN_BUFFMIN.
Additionally, for every disk page in buffer, the system notes the most recent LSN of an action that has performed an update on a data item in that page, and calls this value LSN_PGMAX.

Modification rule of LRU buffering policy is the following:

> A disk page cannot be written out to disk from the LRU buffer unless its associated LSN_PGMAX < LSN_BUFFMIN.

The rule guarantees that the associated page never gets written out to disk until the log buffer with the relevent log entry has been written out to disk.

4 Checkpoints

The question is how far should we perform the ROLLBACK procedure?
- to the time of system startup
- to the time defined by a user (DBA)

Problem: the length of time to perform recovery grows with the length of the log file that we need to read through:
- ROLLBACK – most of transactions are relatively short-lived update transactions – the ROLLBACK phase is quite efficient
- ROLL FORWARD – we need to REDO all updates of transactions that have committed (most of them). In fact, we have to perform once more all updates – this phase may be very inefficient and takes more time than original transactions.

Checkpoint – new point of time, since which and till which ROLLBACK and ROLL FORWARD are performed.

We may ignore all the transactions and page updates that occurred before the checkpoint.

There are three distinct approaches to creating a recovery checkpoints:

- commit-consistent checkpointing
- cache-consistent checkpointing
- fuzzy checkpointing

The Commit-Consistent Checkpoint

The system enters a "performing checkpoint state" with the following steps.

1. No new transactions can start until the checkpoint is complete.
2. Database operation processing continues until all existing transactions commit, and all their log entries are written out to disk.
3. The current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffer have been written out to disk.
4. When steps 1-3 have been performed, the system writes a special log entry, (CKPT), to disk, and the checkpoint is complete.

This procedure is very similar we use for a shutdown of the system, and the final state is equivalent to what we have been thinking of as system startup.

Example:

- sizes of buffers - 200 MB
- page size - 4KB
- 50 000 pages – assume that half of them are dirty, so we have 25 000 dirty pages that have to be writte out to disk
- each page requires 25 ms to write to disk
- total time required to write of dirty pages to disk is - 625 seconds.
- total time required to perform a checkpoint is much longer!!!

The Cache-Consistent Checkpoint

To avoid problems of halting system operations during a checkpoint procedure, a new type of a checkpoint is defined – the cache-consistent checkpoint.

Transactions remain active during the checkpoint procedure, and we only require that all dirty pages in buffer and associated write log entries be forced out to disk.

Disk buffer space in memory is also known as disk cache $\Rightarrow$ this explains the name of the checkpoint.

While a cache-consistent checkpoint is in process, the active transactions must WAIT, in particular they cannot perform new I/O operations.

Benefit – we do not have to wait until all active transactions (e.g. long-lived) are completed.

The system enters a "performing cache-consistent checkpoint state" with the following steps.

1. No new transactions can start until the checkpoint is complete.
2. Active transactions are not permitted to start any new operations.
3. The current log buffer is written out to the log file, and after this the system ensures that all dirty pages in buffer have been written out to disk.
4. When steps 1-3 have been performed, the system writes a special log entry, (CKPT, List), to disk, and the checkpoint is complete. The List contains a list of active transactions at the time the checkpoint is created.

The recovery procedure when cache-consistent checkpoints are used differs from the procedure when commit-consistent checkpoints are used.

A log file and recovery using a cache-consistent checkpoint

Given the following execution of transactions:

r: T1:r(A, 10) T1:w(A, 1) T1:c T2:r(A, 1) T3:r(B, 2)
T2:w(A, 3) T4:r(C, 5) CKPT T3:w(B, 4) T3:c
T4:r(B, 4) T4:w(C, 6) T4:c **crash**

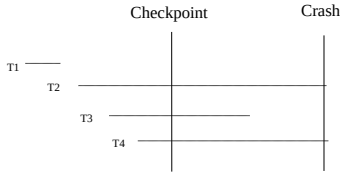
The sequence of log entries resulting from the above execution

(S, 1) (W, 1, A, 10, 1) (C, 1) (S, 2) (S, 3) (W, 2, A, 1, 3) (S, 4) (CKPT, (List = T2, T3, T4)) (W, 3, B, 2, 4) (C, 3) (W, 4, C, 5, 6) **(C, 4) crash**

At the time the cache-consistent checkpoint was created, the following values were written out to disk:

A = 3, B = 2, C = 5

Assume that no other updates were written out to disk before the crash, so the data item values remain the same.



ROLLBACK

1 (C, 3)	Transaction T3 is a committed transaction in active list.
2. (W, 3, B, 2, 4)	Committed transaction, wait for ROLL FORWARD
3. (CKPT, (List = T2, T3, T4))	Active transactions T2,T4 not committed
4. (S, 4)	Delete T4 from List of active transactions
5. (W, 2, A, 1, 3)	T2 not committed, perform UNDO; A = 1
6. (S, 3)	Committed transaction, delete T3 from List
7. (S, 2)	Delete T2 from List of active transactions. Since the list of active transactions is empty, Stop ROLLBACK.

Notice that when ROLLBACK encounters the CKPT log entry, the system takes note of the transactions that were active at the time of checkpoint creation. We remove from the list those transactions that have been committed, and receive a list of transactions whose updates we need to UNDO. We continue the ROLBACK phase until we complete all such UNDO actions.

ROLL FORWARD

8. (CKPT, (List = T2, T3, T4))	Skip forward in log file to this entry. The only committed transaction is T3
9. (W, 3, B, 2, 4)	ROLL FORWARD; B = 4
10. (C, 3)	No action. Last entry in the log file, so ROLL FORWARD is complete.

The aim of the ROLL FORWARD procedure is to perform REDO of all updates made by committed transactions that might not have gone out to disk. We can jump forward to the first log entry after the checkpoint, since we known (from the construction of the checkpoint) that all earlier updates were written out to disk.

Recall that the values of data items on disk at the time of crash were:

A = 3, B = 2, C = 5

At the end of recovery, we have set:

A = 1 (step 5)
B = 4 (step 9)
C = 5

It follows from the fact that T2 and T4 are not committed, while T1 and T3 are committed.

The Fuzzy Checkpoint

The aim of a fuzzy checkpoint is to reduce to an absolute minimum the time needed to perform a checkpoint. Since this time is mainly determined by the time necessary to flush all dirty pages to disk, so this approach tries to minimize the effort necessary to write out dirty pages on disk.

Fuzzy checkpoint makes use of two checkpoint events, the most recent two checkpoints that have been recorded to the log file with the CKPT log entries. The idea is the following. During the creation of the checkpoint $CKPT_N$, the system takes note of the set of dirty pages that have accumulated in buffers since the prior checkpoint $CKPT_{N-1}$. The intent is that all of these pages will be written out to disk by the time of next checkpoint is taken - $CKPT_{N+1}$. During the period between checkpoints, there is a good opportunity for most of the pages dirty during the first checkpoint to drift out to disk under the normal operation of the buffer manager. The rest of dirty pages will be forced out to disk when the next checkpoint is taken. Summarizing, all the pages that were dirty at the time $CKPT_{N-1}$ was taken have been forced out to disk by the time we complete $CKPT_N$.

The system enters a “performing fuzzy checkpoint state” with the following steps.

1. Prior to checkpoint start, the remaining pages dirty as of the previous checkpoint are forced out to disk.
2. No new transactions can start until the checkpoint is complete.
3. Active transactions are not permitted to start any new operations.
4. The current log buffer is written out to the log file with an appended log entry, ($CKPT_N$, List), as in the cache-consistent checkpoint procedure.
5. The set of pages in buffer that are dirty since the last checkpoint log, $CKPT_{N-1}$, is noted. Special flags on the buffer directory may accomplish this. The checkpoint is now completed.

The recovery procedure with fuzzy checkpoints differs from the procedure with cache-consistent checkpoints only in that ROLL FORWARD must start with the first log entry following the second-to-last checkpoint log (i.e. it start with $CKPT_{N-1}$, if the last checkpoint that was taken in the system is $CKPT_N$).