Checkpoint

Hi! Today you will learn about check point and then only you will understand how important check point is? You will also learn the mechanisms of checkpoint. This lecture is covering check point in great depth. Though this topic is huge, introduction and basic to checkpoint is sufficient to understand. But if you want to understand checkpoint from DBA's point of view then go through the entire lecture.

1. INTRODUCTION

Most large institutions have now heavily invested in a data base system. In general they have automated such clerical tasks as inventory control, order entry, or billing. These systems often support a worldwide network of hundreds of terminals. Their purpose is to reliably store and retrieve large quantities of data. The life of many institutions is critically dependent on such systems, when the system is down the corporation has amnesia.

This puts an enormous burden on the implementers and operators of such systems. The systems must on the one hand be very high performance and on the other hand they must be very reliable

SYSTEM CHECKPOINT LOGIC

System checkpoints may be triggered by operator commands, timers, or counters such as the number of bytes of log record since last checkpoint. The general idea is to minimize the distance one must travel in the log in the event of a catastrophe. This must be balanced against the cost of taking frequent checkpoints. Five minutes is a typical checkpoint interval.

Checkpoint algorithms that require a system quiesce should be avoided because they imply that checkpoints will be taken infrequently thereby making restart expensive.

The checkpoint process consists of writing a BEGIN_CHECKPOINT record in the log, then invoking each component of the system so that it can contribute to the checkpoint, and then writing an END_CHECKPOINT record in the log. These records bracket the checkpoint records of the other system components. Such a component may write one or more log records so that it will be able to restart from the checkpoint. For example, buffer manager will record the names of the buffers in the buffer pool, file manager might record the status of files, network manager may record the network status, and transaction manager will record the names of all transactions

active at the checkpoint.

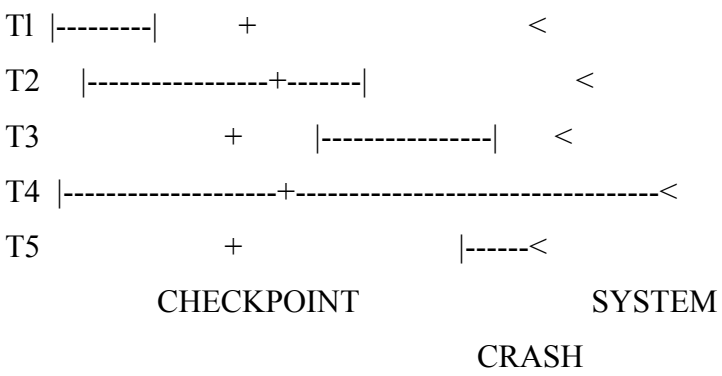
After the checkpoint log records have been written to non-volatile storage, recovery manager records the address of the most recent checkpoint in a warm start file. This allows restart to quickly locate the checkpoint record (rather than sequentially searching the log for it.) Because this is such a critical resource, the restart file is duplexed (two copies are kept) and writes to it are alternated so that one file points to the current and another points to the previous checkpoint log record.

At system restart, the programs are loaded and the transaction manager invokes each component to re-initialize itself. Data communications begins network-restart and the database manager reacquires the database from the operating system (opens the files).

Recovery manager is then given control. Recovery manager examines the most recent warm start file written by checkpoint to discover the location of the most recent system checkpoint in the log. Recovery manager then examines the most recent checkpoint record in the log. If there was no work in progress at the system checkpoint and the system checkpoint is the last record in the log then the system is in restarting from a shutdown in a quiesced state. This is a warm start and no transactions need be undone or redone. In this case, recovery manager writes a restart record in the log and returns to the scheduler, which opens the system for general use.

On the other hand if there was work in progress at the system checkpoint, or if there are further leg records then this is a restart from a crash (emergency restart).

The following figure will help to explain emergency restart logic:



Five transaction types with respect to the most recent system checkpoint and the crash point. Transactions T1, T2, and T3 have committed and must be redone. Transactions T4 and T5 have not committed and so must be undone. Let's call transactions like T1, T2 and T3 winners and lets call transactions like T4 and T5 losers. Then the restart logic is:

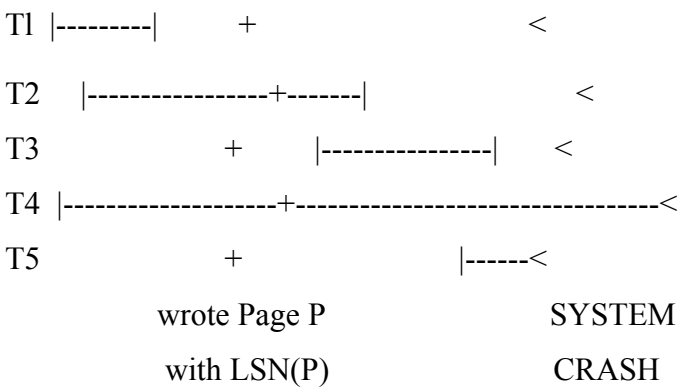
```
RESTART: PROCEDURE;
    DICHOTOMIZE WINNERS AND LOSERS;
    REDO THE WINNERS;
    UNDO THE LOSERS;
    END RESTART;
```

It is important that the REDOs occur before the UNDO (Do you see why (we are assuming page-locking and high-water marks from log-sequence numbers?)

As it stands, this implies reading every log record ever written because redoing the winners requires going back to redo almost all transactions ever run.

Much of the sophistication of the restart process is dedicated to minimizing the amount of work that must be done, so that restart can be as quick as possible, (We are describing here one of the more trivial workable schemes.) In general restart discovers a time T such that redo log records written prior to time T are not relevant to restart.

To see how to compute the time T, we first consider a particular object: a database page P. Because this is a restart from a crash, the most recent version of P may or may not have been recorded on non-volatile storage. Suppose page P was written out with high water mark LSN(P). If the page was updated by a winner "after" LSN(P), then that update to P must be redone. Conversely, if P was written out to nonvolatile storage with a loser's update, then those updates must be undone. (Similarly, message M may or may rot have been sent to its destination.) If it was generated by a loser, then the message should be canceled. If it was generated by a committed transaction but not sent then it should be retransmitted.) The figure below illustrates the five possible types of transactions at this point: T1 began and committed before LSN(P), T2 began before LSN(P) and ended before the crash, T3 began after LSN(P)and ended before the crash, T4 began before LSN(P) but its COMMIT record does not appear in the log, and T5 began after LSN(P) and apparently never ended. To honor the commit of T1, T2 and T3 requires that their updates be added to page P (redone). But T4, T5, and T6 have not committed and so must be undone.



Five transactions types with respect to the most recent write of page P and the crash point,

Notice that none of the updates of T5 are reflected in this state so T5 is already undone. Notice also that all of the updates of T1 are in the state so it need not be redone. So only T2, T3, and T4 remain. T2 and T3 must be redone from LSN(P) forward. The updates of the first half of T2 are already reflected in the page P because it has log sequence number LSN(P). On the other hand, T4 must be undone from LSN(P) backwards. (Here we are

skipping over the following anomaly: if after LSN(P), T2 backs up to a point prior to the LSN(P)then some undo work is required for T2. This problem is not difficult, just annoying.)

Therefore the oldest redo log record relevant to P is at or after LSN(P). (The write-ahead-log protocol is relevant here.) At system checkpoint, data manager records MINLSN, the log sequence number of the oldest page not yet written (the minimum LSN(P)of all pages, P, not yet written.) Similarly, transaction manager records the name of each transaction active at the checkpoint. Restart chooses T as the MINLSN of the most recent checkpoint.

Restart proceeds as follows: It reads the system checkpoint log record and puts each transaction active at the checkpoint into the loser set.

It then scans the log forward to the end. If a COMMIT log record is encountered, that transaction is promoted to the winners set. If a BEGIN_TRANSACTION record is found, the transaction is tentatively added to the loser set. When the end of the log is encountered, the winners and losers have been computed. The next thing is to read the log forwards from MINLSN, redoing the winners. Then it starts from the end of the log, read the log backwards undoing the losers.

This discussion of restart is very simplistic. Many systems have added mechanisms to speed restart by:

- Never write uncommitted objects to non-volatile storage (stealing) so that undo is never required.
- Write committed objects to secondary storage at phase 2 of commit (forcing), so that redo is only rarely required (this maximizes “MINLSN).
- Log the successful completion of a write to secondary storage. This minimizes redo.
- Force all objects at system checkpoint, thereby maximizing MINLSN.

5.8.4.5. MEDIA FAILURE LOGIC

In the event of a hard system error (one non-volatile storage integrity), there must be minimum of lost work. Redundant copies of the object must be maintained, for example

on magnetic tape that is stored in a vault. It is important that the archive mechanism have independent failure modes from the regular storage subsystem. Thus, using doubly redundant disk storage would protect against a disk head crash, but wouldn't protect against a bug in the disk driver routine or a fire in the machine room. The archive mechanism periodically writes a checkpoint of the data base contents to magnetic tape, and writes a redo log of all update actions to magnetic tape, Then recovering from a hard failure is accomplished by locating the most recent surviving version on tape, loading it back into the system, and then redoing all updates from that point forward using the surviving log tapes.

While performing a system checkpoint causes relatively few disk writes, and takes only a few seconds, copying the entire database to tape is potentially a lengthy operation. Fortunately there is a (little used) trick: one can take a fuzzy dump or an object by writing it to archive with an idle task. After the dump is taken, the log generated during the fuzzy dump is merged with the fuzzy dump to produce a sharp dump. The details of this algorithm are left as an exercise for the reader.

5.8.4.6. COLD START LOGIC

Cold start is too horrible to contemplate. Since we assumed that the log never fails, cold start is never required. The system should be cold started once: when the implementers create its first version. Thereafter, it should be restarted. In particular moving to new hardware or adding to a new release of the system should not require a cold start. (i.e. all data should survive.) Note that this requires that the format of the log never change, it can only be extended by adding new types of log records.

5.8.5. LOG MANAGEMENT

The log is a large linear byte space. It is very convenient if the log is write-once, and then read-only. Space in the log is never re-written. This allows one to identify log records by the relative byte address of the last byte of the record.

A typical (small) transaction writes 500 bytes of log. One can run about one hundred such transactions per second on current hardware. There are almost 100,000 seconds in a day. So the log can grow at 5 billion bytes per day. (more typically, systems write four log

tapes a day at 50 megabytes per tape.) Given those statistics the log addresses should be about 48 bits long (good for 200 years on current hardware.)

Log manager must map this semi-infinite logical file (log) into the rather finite files (32 bit addresses) provided by the basic operating system. As one file is filled, another is allocated and the old one is archived. Log manager provides other resource managers with the operations:

WRITE_LOG: causes the identified log record to be written to the log. Once a log record is written. It can only be read. It cannot be edited. WRITE_LOG is the basic command used by all resource managers to generate log records. It returns the address of the last byte of the written log record.

FORCE-LOG: causes the identified log record and all prior log records to be recorded in nonvolatile storage. When it returns, the writes have completed.

OPEN-LOG: indicates that the issuer wishes to read the log of some transaction, or read the entire log in sequential order. It creates a read cursor on the log.

SEARCH-LOG: moves the cursor a designated number of bytes or until a log record satisfying some criterion is located.

READ-LOG: requests that the log record currently selected by the log cursor be read.

CHECK-LOG: allows the issuer to test whether a record has been placed in the non-volatile log and optionally to wait until the log record has been written out.

GET-CURSOR: causes the current value of the write cursor to be returned to the issuer. The RBA (relative byte address) returned may be used at a later time to position a read cursor.

CLOSE-LOG: indicates the issuer is finished reading the log.

The write log operation moves a new log record to the end of the current log buffer. If the buffer fills, another is allocated and the write continues into the new buffer.

When a log buffer fills or when a synchronous log write is issued, a log daemon writes the buffer to nonvolatile storage. Traditionally, logs have been recorded on magnetic tape

because it is so inexpensive to store and because the transfer rate is quite high. In the future disk, CCD (nonvolatile?) or magnetic bubbles may be attractive as a staging device for the log. This is especially true because an on-line version of the log is very desirable for transaction undo and for fast restart.

It is important to doubly record the log. If the log is not doubly recorded, then a media error on the log device will produce a cold start of the system. The dual log devices should be on separate paths so that if one device or path fails the system can continue in degraded mode (this is only appropriate for applications requiring high availability.)

The following problem is left as an exercise for the reader: We have decided to log to dedicated dual disk drives. When a drive fills it will be archived to a mass storage device. This archive process makes the disk unavailable to the log manager (because of arm contention.) Describe a scheme which:

- minimizes the number of drives required, .
- always has a large disk reserve of free disk space, and
- always has a large fraction of the recent section of the log on line.

5.8.5.1. LOG ARCHIVE AND CHANGE ACCUMULATION

When the log is archived, it can be compressed so that it is convenient for media recovery. For disk objects, log records can be sorted by cylinder, then track then sector then time. Probably, all the records in the archived log belong to completed transactions. So one only needs to keep redo records of committed (not aborted) transactions. Further only the most recent redo record (new value) need be recorded. This compressed redo log is called a change accumulation log. Since it is sorted by physical address, media recover becomes a merge of the image dump of the object and its change accumulation tape.

```
FAST_MEDIA_RECOVERY: PROCEDURE (IMAGE,
CHANGE_ACCUMULATION_LOG);
    DO WHILE ( ! END_OF_FILE IMAGE);
        READ IMAGE PAGE;
        UPDATE WITH REDO RECORDS FROM
```

```
CHANGE_ACCUMULATION_LOG;
        WRITE IMAGE PAGE TO DISK;
        END
    END;
```

This is a purely sequential process (sequential on input files and sequential on disk being recovered) and so is limited only by the transfer rates of the devices.

The construction of the change accumulation file can be done off-line as an idle task.

If media errors are rare and availability of the data is not a critical problem then one may run the change accumulation utilities when needed. This may save building change accumulation files that are never used.

The same topic check point is dealt in detail as tutorials which are added at the last of these lectures. For further reference, refer those tutorials.

Review Question

1. What is Checkpoint and why it is important?