**ST. XAVIER’S COLLEGE**

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**Database Management System Theory Assignment #9**

**Submitted by:**

Sazjan Neupane

013BSCCSIT036

**Submitted to:**

|  |  |
| --- | --- |
| Er. Sanjay Kumar Yadav  Lecturer, St. Xavier’s College |  |

**Date of Submission:** 1st October, 2015

**Database Recovery**

Database Recovery is the process of restoring the database and the data to a consistent state. This may include restoring lost data up to the point of the event (e.g. system crash). A major responsibility of the database administrator is to prepare for the possibility of hardware, software, network, process, or system failure. If such a failure affects the operation of a database system, you must usually recover the database and return to normal operation as quickly as possible. Recovery should protect the database and associated users from unnecessary problems and avoid or reduce the possibility of having to duplicate work manually.

**Major types of Errors and Failures**

To see where the problem has occurred, we generalize a failure into various categories, as follows −

**Transaction failure**

A transaction has to abort when it fails to execute or when it reaches a point from where it can’t go any further. This is called transaction failure where only a few transactions or processes are hurt.

* Reasons for a transaction failure could be −
  + Logical errors − Where a transaction cannot complete because it has some code error or any internal error condition.
  + System errors − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

**System Crash**

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

**Disk Failure**

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, un-reach ability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

**Storage Structure**

* **Volatile storage:**
  + does not survive system crashes
  + examples: main memory, cache memory
* **Nonvolatile storage:**
  + survives system crashes
  + examples: disk, tape, flash memory,
  + non-volatile (battery backed up) RAM
* **Stable storage:**
  + a mythical form of storage that survives all failures
  + approximated by maintaining multiple copies on distinct
  + nonvolatile media

**Database Buffer Management**

Database maintains an in-memory buffer of data blocks.

When a new block is needed, if buffer is full an existing block needs to be removed from buffer. If the block chosen for removal has been updated, it must be output to Disk. If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first (Write ahead logging).

No updates should be in progress on a block when it is output to disk can be ensured as follows. Before writing a data item, transaction acquires exclusive lock on block containing the data item. Lock can be released once the write is completed. Such locks held for short duration are called latches. Before a block is output to disk, the system acquires an exclusive latch on the block ensures no update can be in progress on the block.

Database buffer can be implemented either in an area of real main-memory reserved for the database, or in virtual memory.

Implementing buffer in reserved main-memory has drawbacks:

* Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
* Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

## Data updates

Immediate Update: As soon as a data item is modified in cache, the disk copy is updated.

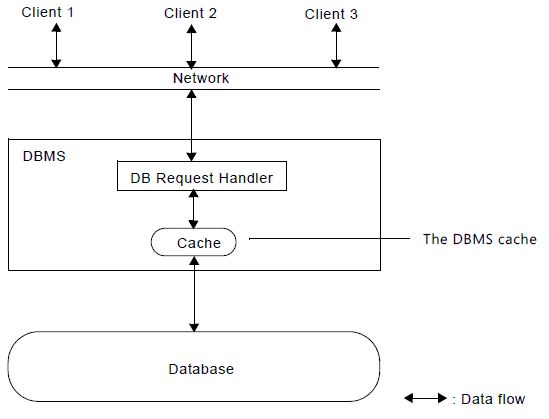
Deferred Update: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.

Shadow update: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.

In-place update: The disk version of the data item is overwritten by the cache version.

## Data caching

The Database Management System (DBMS) is a memory buffer which stores copies of portions of the database that the DBMS is currently using. Reading from memory is much faster than reading from the disk. The DBMS therefore returns a record more quickly if it is already stored in cache. As long as the required data is stored in cache, the data is immediately available. When the required data is not stored in cache, it must be copied from the disk and then stored in cache.



**Transaction logs**

The transaction log is a critical component of the database and, if there is a system failure, the transaction log might be required to bring your database back to a consistent state. The transaction log should never be deleted or moved unless you fully understand the ramifications of doing this.

The transaction log must be truncated on a regular basis to keep it from filling up. However, some factors can delay log truncation, so monitoring log size is important. Some operations can be minimally logged to reduce their impact on transaction log size. Physically, a log is a [file](https://en.wikipedia.org/wiki/Computer_file) listing changes to the database, stored in a stable storage format.

If, after a start, the database is found in an [inconsistent](https://en.wikipedia.org/wiki/Consistency_(database_systems)) state or not been shut down properly, the database management system reviews the database logs for [uncommitted](https://en.wikipedia.org/wiki/Commit_(data_management)) transactions and [rolls back](https://en.wikipedia.org/wiki/Rollback_(data_management)) the changes made by these [transactions](https://en.wikipedia.org/wiki/Database_transaction). Additionally, all transactions that are already committed but whose changes were not yet materialized in the database are re-applied. Both are done to ensure [atomicity](https://en.wikipedia.org/wiki/Atomicity_(database_systems)) and [durability](https://en.wikipedia.org/wiki/Durability_(computer_science)) of transactions.

**The redo log and rolling forward**

The redo log is a set of operating system files that record all changes made to any database buffer, including data, index, and rollback segments, whether the changes are committed or uncommitted. Each redo entry is a group of change vectors describing a single atomic change to the database. The redo log protects changes made to database buffers in memory that have not been written to the data files.

The first step of recovery from an instance or disk failure is to roll forward, or reapply all of the changes recorded in the redo log to the data files. Because rollback data is also recorded in the redo log, rolling forward also regenerates the corresponding rollback segments. This is called I.

Rolling forward proceeds through as many redo log files as necessary to bring the database forward in time. Rolling forward usually includes online redo log files and may include archived redo log files.

After roll forward, the data blocks contain all committed changes. They may also contain uncommitted changes that were either saved to the data files before the failure, or were recorded in the redo log and introduced during roll forward.

**Rollback Segments and Rolling Back**

Rollback segments record database actions that should be undone during certain database operations. In database recovery, rollback segments undo the effects of uncommitted transactions previously applied by the rolling forward phase.

After the roll forward, any changes that were not committed must be undone. After redo log files have reapplied all changes made to the database, then the corresponding rollback segments are used. Rollback segments are used to identify and undo transactions that were never committed, yet were either saved to the data files before the failure, or were applied to the database during the roll forward. This process is called rolling back or transaction recovery*.*

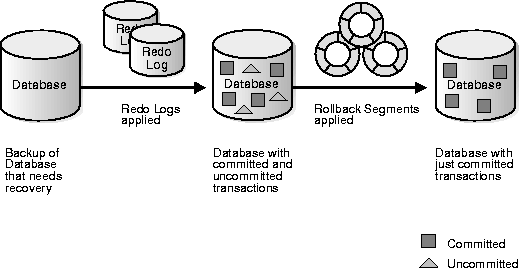


Fig. illustrates rolling forward and rolling back, the two steps necessary to recover from any type of system failure.

**Check pointing**

When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Problems in recovery procedure as discussed earlier :

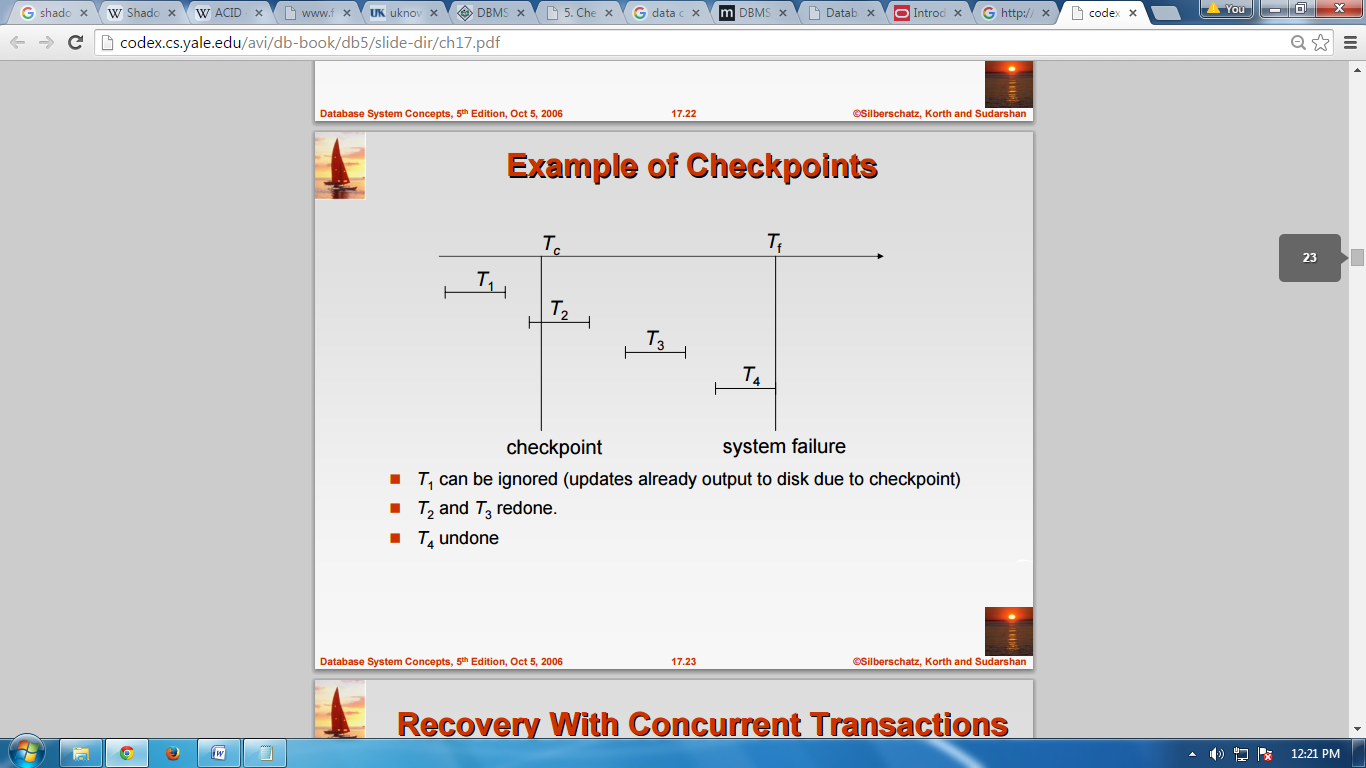
* Searching the entire log is time-consuming
* We might unnecessarily redo transactions which have already
* Output their updates to the database.

Streamline recovery procedure by periodically performing

* Output all log records currently residing in main memory onto stable storage.
* Output all modified buffer blocks to the disk.
* Write a log record < checkpoint> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the check point, and transactions that started after Ti.

* Scan backwards from end of log to find the most recent <checkpoint> record
* Continue scanning backwards till a record <Ti start> is found.
* Need only consider the part of log following above start record.
* Earlier part of log can be ignored during recovery, and can be erased whenever desired.
* For all transactions (starting from Ti or later) with no <Ti commit>, execute undo(Ti).
* Scanning forward in the log, for all transactions starting from Ti or later with a <Ti commit>, execute redo(Ti).



*Fig. example of check points*

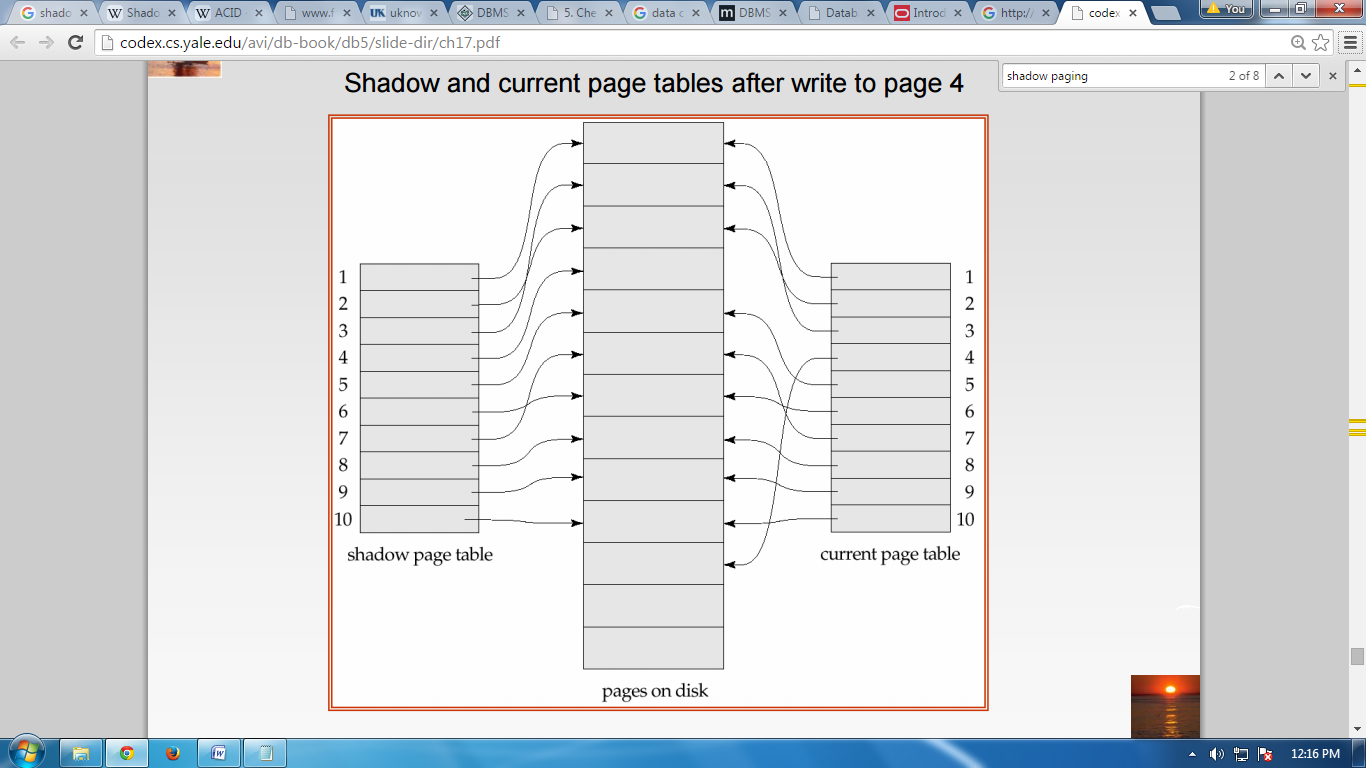
**Shadow Paging**

Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially.

* Idea: maintain two page tables during the lifetime of a transaction –the current page table, and the shadow page table
* Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.

Shadow page table is never modified during execution; to start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction. Whenever any page is about to be written for the first time

* A copy of this page is made onto an unused page.
* The current page table is then made to point to the copy.
* The update is performed on the copy.



*Fig. shadow paging*

**Failure with loss of non volatile storage**

During recovery we need to consider only the most recent transaction Ti that started before the check point, and transactions that started after Ti.

* Scan backwards from end of log to find the most recent <checkpoint> record
* Continue scanning backwards till a record <Ti start> is found.
* Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
* For all transactions (starting from Ti or later) with no <Ti commit>, execute undo(Ti). (Done only in case of immediate modification.)
* Scanning forward in the log, for all transactions starting from Ti or later with a <Ti commit>, execute redo(Ti).

**WAL: Write ahead Logging Protocol**

Write-ahead logging (WAL) is a family of techniques for providing [atomicity](https://en.wikipedia.org/wiki/Atomic_(computer_science)) and [durability](https://en.wikipedia.org/wiki/Durability_(database_systems)) (two of the [ACID](https://en.wikipedia.org/wiki/ACID) properties) in [database systems](https://en.wikipedia.org/wiki/Database_system). In a system using WAL, all modifications are written to a [log](https://en.wikipedia.org/wiki/Database_log) before they are applied. Usually both redo and undo information is stored in the log.

The purpose of this can be illustrated by an example. Imagine a program that is in the middle of performing some operation when the machine it is running on loses power. Upon restart, that program might well need to know whether the operation it was performing succeeded, half-succeeded, or failed. If a write-ahead log is used, the program can check this log and compare what it was supposed to be doing when it unexpectedly lost power to what was actually done. On the basis of this comparison, the program could decide to undo what it had started, complete what it had started, or keep things as they are.

Before a block of data in main memory can be output to the database (in nonvolatile storage), all log records pertaining to data in that block must have been output to stable storage. This rule is called the WAL rule. Strictly speaking, the WAL rule requires only that the undo information in the log have been output to stable storage, and permits the redo information to be written later. The difference is relevant in systems where undo information and redo information are stored in separate log records. Write-ahead logging is employed to flush log records to the persistent log file before data pages are written or at commit time.

**Recovery in Multi database systems**

A multi database consists of a collection of autonomous local databases. Systems used to manage multi databases are called multi database systems (MDBSs). In such a system, global transactions are executed under the control of the MDBS. A computer system is subject to failures. Such failures may provoke loss of information. Hence, MDBSs should be able to react in failure situations in order to restore the multi database to a consistent state, without human intervention, that is, automatically. However, ensuring reliability in MDBSs is a very complex task. In order to make transaction processing in MDBSs resilient to failures, two types of protocols are required. One type of protocol should enforce that, when a given global transaction completes its execution, it has the same state (committed or aborted) at every site it has run. Such protocols ensure what we call commit atomicity. They are called commit protocols. The other type of protocols, denoted recovery protocols, determines the actions to be triggered after failures in a multi database environment. In this work, we will describe a commit and a recovery strategy. They can be implemented to ensure transaction processing reliability in an MDBS which does not use a 2PC protocol. The proposed commit strategy guarantees commit atomicity. In turn, the recovery strategy, denoted ReMT (Recovering Multi database Transactions), enables the MDBS to deal with the specific failures in multi database environments. Our proposal is suitable to a wide variety of multi database applications, such as CAD, CASE, GIS and WFMS.