ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**Database Management System Assignment #9**

**Submitted by:**

Binod Marikhu  
013BSCCSIT013

**Submitted to:**

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

**Date of Submission: October 2, 2015**

* **Database Recovery**

In every database system, the possibility of a system failure is always present. Should system failure occur, we must recover the database as quickly, and with as little detrimental impact on users, as possible. Data recovery is the process of restoring data that has been lost, accidentally deleted, corrupted or made inaccessible for any reason.

Recovering from any type of system failure requires the following:

1. Determining which data structures are intact and which ones need recovery.

2. Following the appropriate recovery steps.

3. Restarting the database so that it can resume normal operations.

4. Ensuring that no work has been lost nor incorrect data entered in the database.

The goal is to return to normal as quickly as possible while insulating database users from any problems and the possibility of losing or duplicating work.

The recovery process varies depending on the type of failure and the files of the database affected by the failure.

1. **Purpose of Data Recovery**

As a backup administrator, your principal duty is to devise, implement, and manage a backup and recovery strategy. In general, the purpose of a backup and recovery strategy is to protect the database against data loss and reconstruct the database after data loss. Typically, backup administration tasks include the following:

• Planning and testing responses to different kinds of failures

• Configuring the database environment for backup and recovery

• Setting up a backup schedule

• Monitoring the backup and recovery environment

• Troubleshooting backup problems

• Recovering from data loss if the need arises

1. **Types of Failure**

Failures may be:

|  |  |
| --- | --- |
| Transaction | Caused by errors within the transaction processes. |
| System | Caused by failure of network or operating system or physical threats to the system as a whole. |
| Media | Failure of hard disk, out of memory errors, out of disk space errors. |

Failure may be caused by a number of things.

|  |  |
| --- | --- |
| A System Crash | A hardware, software or network error causes the transaction to fail. |
| Transaction or System error | Some operation in the transaction may cause the failure or the user may interrupt the transaction. |
| Local Errors or Exceptions | Conditions occur during the transaction that results in transaction cancellation. |
| Concurrency Control Enforcement | Several transactions may be in deadlock so the transaction may be aborted to be restarted later. |
| Disk Failure | Read Write error on the physical disk. |
| Physical Problems | This can be any range of physical problems, such as power failure, mounting wrong disk or tape by operator, wiring problems etc |
| Catastrophe Situations | Large scale threats to the system and the data for example fire, cyclone, security breaches etc. |

Transaction errors, system errors, system crashes, concurrency problems and local errors or exceptions are the more common causes of system failure.  The system must be able to recover from such failures without loss of data.

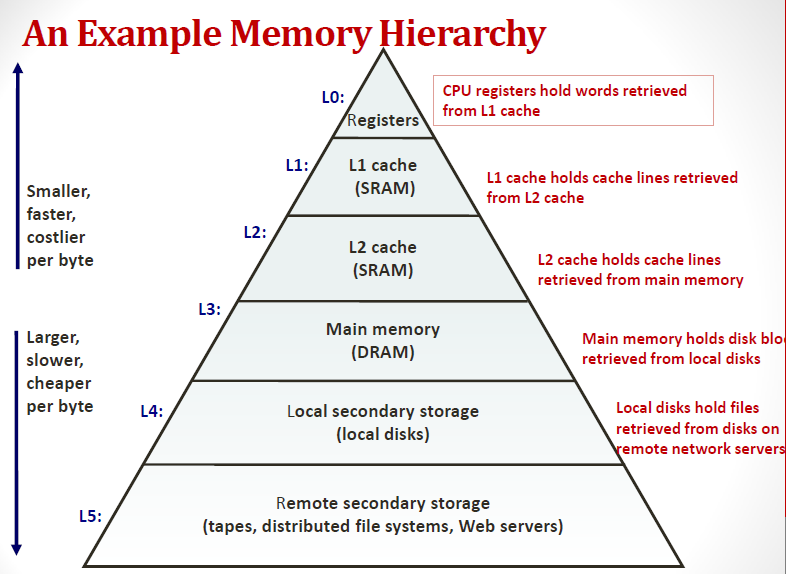
1. **Storage Hierarchy**

Databases are stored in file formats, which contain records. At physical level, the actual data is stored in electromagnetic format on some device. These storage devices can be broadly categorized into three types −

* **Primary Storage** − The memory storage that is directly accessible to the CPU comes under this category. CPU's internal memory (registers), fast memory (cache), and main memory (RAM) are directly accessible to the CPU, as they are all placed on the motherboard or CPU chipset. This storage is typically very small, ultra-fast, and volatile. Primary storage requires continuous power supply in order to maintain its state. In case of a power failure, all its data is lost.
* **Secondary Storage** − Secondary storage devices are used to store data for future use or as backup. Secondary storage includes memory devices that are not a part of the CPU chipset or motherboard, for example, magnetic disks, optical disks (DVD, CD, etc.), hard disks, flash drives, and magnetic tapes.
* **Tertiary Storage** − Tertiary storage is used to store huge volumes of data. Since such storage devices are external to the computer system, they are the slowest in speed. These storage devices are mostly used to take the back up of an entire system. Optical disks and magnetic tapes are widely used as tertiary storage.

**Memory Hierarchy**

A computer system has a well-defined hierarchy of memory. A CPU has direct access to it main memory as well as its inbuilt registers. The access time of the main memory is obviously less than the CPU speed. To minimize this speed mismatch, cache memory is introduced. Cache memory provides the fastest access time and it contains data that is most frequently accessed by the CPU.



The memory with the fastest access is the costliest one. Larger storage devices offer slow speed and they are less expensive, however they can store huge volumes of data as compared to CPU registers or cache memory.

1. **Buffer Management**
   * We need to use disk storage for the database, and to transfer blocks of data between MM and disk. We also want to minimize the number of such transfers, as they are time-consuming. One way is to keep as many blocks as possible in MM.
   * Usually, we cannot keep all blocks in MM, so we need to manage the allocation of available MM space.
   * The **buffer** is the part of MM available for storage of **copies** of disk blocks.
   * The subsystem responsible for the allocation of buffer space is called the **buffer manager**.
   * The buffer manager handles all requests for blocks of the database.
   * If the block is already in MM, the address in MM is given to the requestor.
   * If not, the buffer manager must read the block in from disk (possibly displacing some other block if the buffer is full) and then pass the address in MM to the requestor.

The buffer manager must use some sophisticated techniques in order to provide good service:

* + **Replacement Strategy** - When there is no room left in the buffer, some block must be removed to make way for the new one. Typical operating system memory management schemes use a ``least recently used'' (**LRU**) method. (Simply remove the block least recently referenced.) This can be improved upon for database applications.
  + **Pinned Blocks** - For the database to be able to recover from crashes, we need to restrict times when a block maybe written back to disk. A block not allowed to be written is said to be **pinned**. Many operating systems do not provide support for pinned blocks, and such a feature is essential if a database is to be ``crash resistant''.
  + **Forced Output of Blocks** - Sometimes it is necessary to write a block back to disk even though its buffer space is not needed. (Called the **forced output** of a block.) This is due to the fact that MM contents (and thus the buffer) are lost in a crash, while disk data usually survives.

**Replacement Strategy:** Goal is minimization of accesses to disk. Generally it is hard to predict which blocks will be referenced. So operating systems use the history of past references as a guide to prediction.

* 1. **General Assumption:** Blocks referenced recently are likely to be used again.
  2. **Therefore:** if we need space, throw out the least recently referenced block. (LRU replacement scheme)

LRU is acceptable in **operating systems**, however, a database system is able to predict future references more accurately.

1. **Transaction Log**

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction log (also transaction journal, database log, binary log or audit trail) is a history of actions executed by a database management system to guarantee ACID properties over crashes or hardware failures. Physically, a log is a file listing changes to the database, stored in a stable storage format.

If, after a start, the database is found in an inconsistent state or not been shut down properly, the database management system reviews the database logs for uncommitted transactions and rolls back the changes made by these transactions. 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 and durability of transactions.

The database can be modified using two approaches −

* **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
* **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

1. **Data Update**

**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.

1. **Data Caching**

Caching is the practice of storing data in and retrieving data from a high-performance store (usually memory) either explicitly or implicitly. Memory is faster to access than a file, a remote URL (usually), a database or any other external store of information. Cashing is generally used in case of database access and manipulation

Consider the following:

A web page request gets to the web server, which passes the request on to the application server, which executes some code that renders the page, which needs to turn to the database to dynamically retrieve data.

This model does not scale well, because as the number of requests for the page goes up, the server has to do the same thing over and over again, for every request.

This becomes even more of an issue if web server, application server, and database are on different hardware and communicate over the network with each other.

If you have a large number of users hitting this page, it makes sense to not go all the way through to the database for every request. Instead, you resort to caching at different levels.

**Resultset Cache**

Resultset caching is storing the results of a database query along with the query in the application. Every time a web page generates a query, the applications checks whether the results are already cached, and if they are, pulls them from an in-memory data set instead. The application still has to render the page.

**Component Cache**

A web page is comprised of different components - pagelets, or whatever you may want to call them. A component caching strategy must know what parameters were used to request the component. For instance, a little "Latest News" bar on the site uses the user's geographical location or preference to show local news. Consequently, if the news for a location is cached, the component does not need to be rendered and can be pulled from a cache.

**Page Cache**

One strategy for caching entire pages is to store the query string and/or header parameters along with the completely renderered HTML. The file system is fast enough for this - it is still way less expensive for a web server to read a file than to make a call to the application server to have the page rendered. In this case, every user who sends the same query string will get the same cached content.

Combining these caching strategies intelligently is the only way to create really scalable web apps for large numbers of concurrent users. As you can easily see, the potential risk here is that if a piece of content in the cache cannot be uniquely identified by it's key, people will start to see the wrong content. This can get pretty complicated, particularly when users have sessions and there is security context.

1. **Transaction Roll Back (Undo) and Roll Forward**

The transaction log Includes information helpful to the recovery process such as: A transaction identifier, the date and time, the user running the transaction, before images and after images

Before Image: A copy of the table record (or data item) before it was changed by the transaction.

After Image: A copy of the table record (or data item) after it was changed by the transaction.

**Rollback:** Undo any partially completed transactions (ones in progress when the crash occurred) by applying the before images to the database.

**Rollforward:** Redo the transactions by applying the after images to the database. This is done for transactions that were committed before the crash.

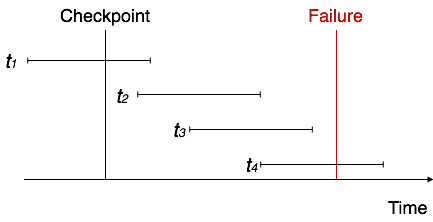
transaction and to over-write the changed value once again to ensure the consistency.

1. **Check Pointing, Shadow Paging**

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

**Recovery**

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



* The recovery system reads the logs backwards from the end to the last checkpoint.
* It maintains two lists, an undo-list and a redo-list.
* If the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>, it puts the transaction in the redo-list.
* If the recovery system sees a log with <Tn, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs

In computer science, shadow paging is a technique for providing atomicity and durability (two of the ACID properties) in database systems. A page in this context refers to a unit of physical storage (probably on a hard disk), typically of the order of 210 to 216 bytes.

Shadow paging is a copy-on-write technique for avoiding in-place updates of pages. Instead, when a page is to be modified, a shadow page is allocated. Since the shadow page has no references (from other pages on disk), it can be modified liberally, without concern for consistency constraints, etc. When the page is ready to become durable, all pages that referred to the original are updated to refer to the new replacement page instead. Because the page is "activated" only when it is ready, it is atomic.

1. **Recovery Schemess (WAL: Write Ahead Logging Protocol)**

**Write Ahead Logging (WAL)** is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems. In a system using WAL, all modifications are written to a log before they are applied. Usually both redo and undo information is stored in the log.

The main goals of WAL are:

* 1. Provide high-availability (fast repair) and consistency in the presence of failures
  2. Transaction failures

1. Logical errors: internal error condition, bad input, data not found, resource unavailable, etc. These correspond to software faults
2. System errors: deadlock or other system problem that prevents the execution of a transaction
3. rollback: explicit call to fail a transaction by an application. We will often use the term rollback to describe all of these conditions, which all cause the log to “rollback”
   1. System crash
4. bug in the OS or the DB
5. hardware problem
6. environmental fault (power)
   1. We will be taking a single transaction view of recovery at this point.
7. While I will discuss multiple transactions, we will not consider isolation.
8. So WAL is a protocol for atomicity and durability iii. What about consistency?
9. **Failure with Loss of Non-Volatile Storage (General Concept)**

A volatile storage like RAM stores all the active logs, disk buffers, and related data. In addition, it stores all the transactions that are being currently executed. What happens if such a volatile storage crashes abruptly? It would obviously take away all the logs and active copies of the database. It makes recovery almost impossible, as everything that is required to recover the data is lost.

Following techniques may be adopted in case of loss of volatile storage −

* We can have **checkpoints** at multiple stages so as to save the contents of the database periodically.
* A state of active database in the volatile memory can be periodically**dumped** onto a stable storage, which may also contain logs and active transactions and buffer blocks.
* <dump> can be marked on a log file, whenever the database contents are dumped from a non-volatile memory to a stable one.

1. **Recovery in Multidatabase System**

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager,**or **coordinator,**is needed to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables).

The coordinator usually follows a protocol called the **two-phase commit protocol,**whose two phases can be stated as follows:

• **Phase 1:**When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message "prepare for commit" to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a "ready to commit" or "OK" signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a "cannot commit" or "not OK" signal to the coordinator. If the coordinator does not receive a reply from a database within a certain time out interval, it assumes a "not OK" response.

• **Phase 2:**If *all*participating databases reply "OK," and the coordinator’s vote is also "OK," the transaction is successful, and the coordinator sends a "commit" signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases or the coordinator have a "not OK" response, the transaction has failed, and the coordinator sends a message to "roll back" or UNDO the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before Phase 1 usually requires the transaction to be rolled back, whereas a failure during Phase 2 means that a successful transaction can recover and commit.