**ST. XAVIER’S COLLEGE**

**Maitighar, Kathmandu**



**DATABASE MANAGEMENT SYSTEM**

**Theory Assignment #9**

**Submitted by**

Sushant Gautam

013BSCCSIT043

**Submitted to**

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

**Date of Submission:**

**Thursday, October 01, 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).

**Purpose of Database Recovery**

There are many situations in which a transaction may not reach a commit or abort point. Some of them are given below:

* To bring the database into the last consistent state which existed prior to the failure.
* To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).
* There are many situations in which a transaction may not reach a commit or abort point.
* An operating system crash can terminate the DBMS processes
* The DBMS can crash
* The system might lose power
* A disk may fail or other hardware may fail.
* Human error can result in deletion of critical data.

Example:

If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

**Types of Failure**

The database may become unavailable for use due to

**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 −

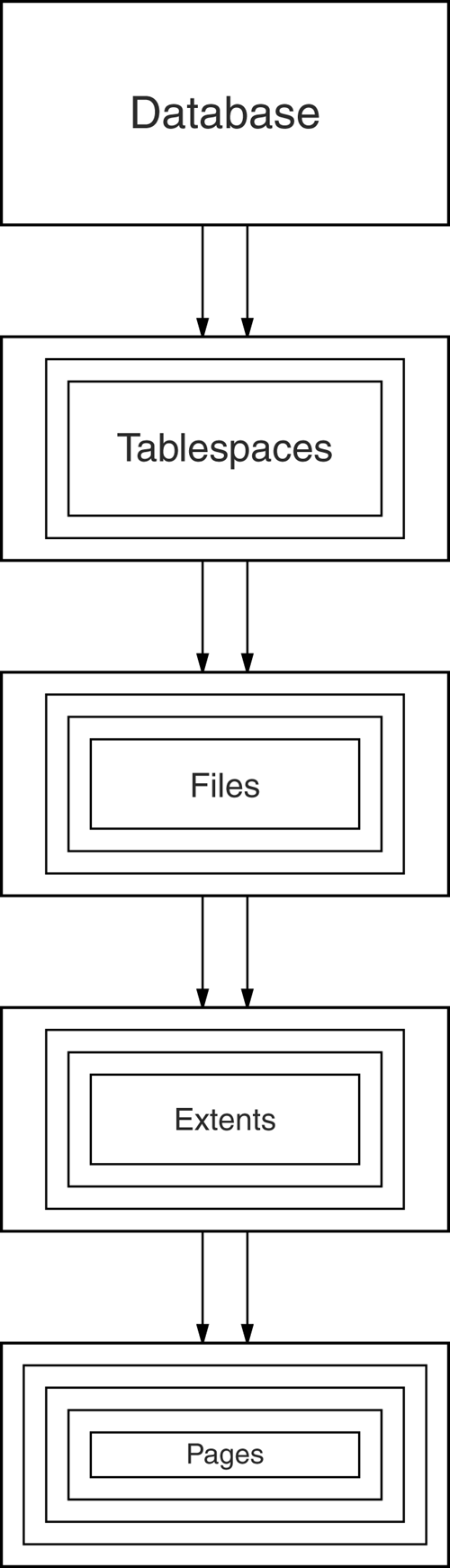
* **Logical errors** − Where a transaction cannot complete because it has some code error or any internal error condition.
* **System errors** − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.
* **System failure**: System may fail because of addressing error, application error, operating system fault, RAM failure, etc. 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.

* **Media failure**: Disk head crash, power disruption, etc.

**The Storage Hierarchy**

You can doubtless think of many examples of storage hierarchies in ordinary life. For example, people live in neighborhoods, which are in towns, which are in regions, countries, continents, and so on up the line. The relations are generally many-to-one, although there are occasional one-to-one correspondences (e.g., Australia is both a country and a continent), and occasional exceptions (e.g., a person can straddle a city boundary).



The figure in the side shows the storage hierarchy—the physical constructs of a database. The hierarchy of physical objects suggests that—with occasional one-to-one correspondences or exceptions—data rows live in pages, which are in extents, which are in files, table spaces, and databases. There is a reason for each level of grouping. To see what the reason is, we'll go through each of those objects in order, up the line.

**Buffer Management**

Buffer management is a key component in achieving this efficiency. The buffer management component consists of two mechanisms: the buffer manager to access and update database pages, and the buffer cache (also called the buffer pool), to reduce database file I/O.

A buffer is an 8-KB page in memory, the same size as a data or index page. Thus, the buffer cache is divided into 8-KB pages. The buffer manager manages the functions for reading data or index pages from the database disk files into the buffer cache and writing modified pages back to disk. A page remains in the buffer cache until the buffer manager needs the buffer area to read in more data. Data is written back to disk only if it is modified. Data in the buffer cache can be modified multiple times before being written back to disk.

The buffer manager only performs reads and writes to the database. Other file and database operations such as open, close, extend, and shrink are performed by the database manager and file manager components.

**Transaction Log:**

* For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
* These values and other information is stored in a sequential file called Transaction log. A sample log is given below. Back P and Next P point to the previous and next log records of the same transaction.

**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.

**Data Caching**

* Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk.
* The flushing is controlled by **Modified** and **Pin-Unpin** bits.
* **Pin-Unpin**: Instructs the operating system not to flush the data item.
* **Modified**: Indicates the AFIM of the data item.

**Transaction Roll-back (Undo) and Roll-Forward (Redo)**

* To maintain atomicity, a transaction’s operations are redone or undone.
* **Undo**: Restore all BFIMs on to disk (Remove all AFIMs).
* **Redo**: Restore all AFIMs on to disk.
* Database recovery is achieved either by performing only Undo’s or only Redo’s or by a combination of the two. These operations are recorded in the log as they happen.

**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.



The AFIM does not overwrite its BFIM but

recorded at another place on the disk. Thus,

at any time a data item has AFIM and BFIM

(Shadow copy of the data item) at two different

places on the disk.

|  |
| --- |
| X and Y: Shadow copies of data items  X' and Y': Current copies of data items |

To manage access of data items by concurrent transactions two directories (current and shadow) are used.

**Recovery Schemes (WAL: Write Ahead Logging Protocol)**

When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging (WAL)** protocol. WAL states that

* + **For Undo**: Before a data item’s AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).
  + **For Redo**: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

**Failure with loss of Non-volatile Storage (General Concepts)**

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.

**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.