ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**DBMS Lab Report #9**

**Submitted by:**

Pankaj Niroula  
013BSCCSIT025

**Submitted to:**

|  |  |
| --- | --- |
| Er. Sanjay K. Yadav  Lecturer Department of Computer Science |  |

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

**Database Recovery**

1. **Purpose of Data Recovery**

As a backup administrator, one 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

As a backup administrator, you may also be asked to perform other duties that are related to backup and recovery:

* Data preservation, which involves creating a database copy for long-term storage
* Data transfer, which involves moving data from one database or one host to another

1. **Types of Failure**

* **Transaction Failure**

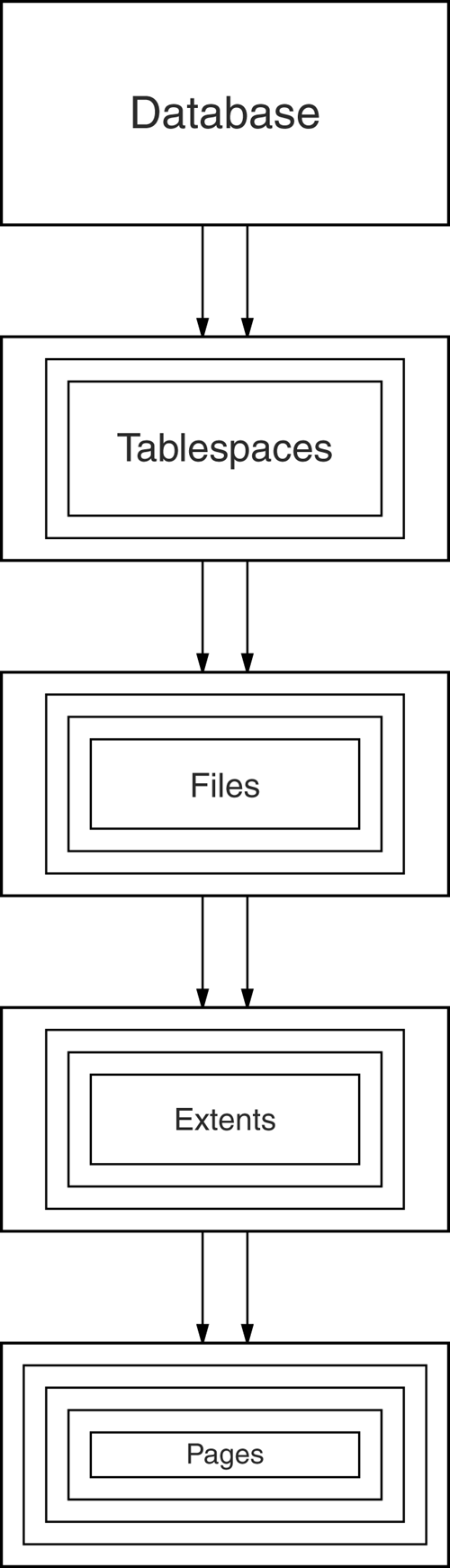
A transaction has to abort when it fails to execute or when it reaches a point from where it can’t go any further. This is called transaction failure where only a few transactions or processes are hurt.

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, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

1. **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.

**Pages**

Depending on the DBMS, a page is also called a data block, a block, a blocking unit, a control interval, and a row group.

A page is a fixed-size hopper that stores rows of data. Pages have four common characteristics, which are not true by definition but are always true in practice.

**Extents**

An extent is a group of contiguous pages. Extents exist to solve the allocation problem. The allocation problem is that, when a file gets full, the DBMS must increase its size. If the file size increases by only one page at a time, waste occurs because: the operating system must update the file allocation tables. The amount of updating is about the same whether the addition is one page or eight pages.

**Files**

A file is a group of contiguous extents. And that's about it. Surprisingly, a file is not a physical representation of a table. It could be, but usually it isn't because of one of the following: most DBMSs allow mixing of extents. That is, the first extent in a file can contain pages for Table1, and the second extent in the same file can contain pages for Table2. This is true even for those DBMSs that do not allow mixing within a page.

Files can be split across more than one drive, either because the file's simply too big or to enhance the advantage of partitioning. (Partitioning is the process of splitting a database object—usually a tablespace, table, or index—into two or more physical locations or partitions)

**Partitions**

Partition is a group of contiguous extents. Often a partition is a file, but it doesn't have to be. Partitions are bad if there are few extents and few database users. Why? Because the fundamental principle is that rows should be crammed together. That helps caching and reduces the number of page writes.

Partitions are good if—and only if—there are many extents and many database users. Why? Two reasons, actually. The first is that in a multiuser environment, partitioning reduces contention. The second is that in a multidisk-drive environment, partitioning increases parallelism.

1. **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.

1. **Transaction Log**

Transaction logs are a vital yet often overlooked component of database architecture. They are often forgotten because they are not something actively maintained like the schema contained within a database. In this article we’ll examine how transaction logs are used in Microsoft SQL Server, maintenance and potential problems with them, how they can be used to restore a database, and finally, optimizing them for performance.

Transaction logs can present problems because they are often forgotten about until an issue occurs. The log continues to grow as operations are performed within the database. While the log continues to grow, the available disk space decreases. Unless routine action is taken to prevent it, the transaction log will eventually consume all available space allocated to it. If the log is configured to grow indefinitely as is the default, it will grow to consume all available physical disk space where it is stored. Either scenario causes the database to stop functioning.

A technique often used to perform recovery is the transaction log or journal.

* records information about the progress of transactions in a log since the last consistent state.
* the database therefore knows the state of the database before and after each transaction.
* every so often database is returned to a consistent state and the log may be truncated to remove committed transactions.
* when the database is returned to a consistent state the process is often referred to as `checkpointing'.

1. **Data Updates**

**Deferred Update**

Deferred update, or NO-UNDO/REDO, is an algorithm to support ABORT and machine failure scenarios.

* While a transaction runs, no changes made by that transaction are recorded in the database.
* On a commit:
* The new data is recorded in a log file and flushed to disk
* The new data is then recorded in the database itself.
* On an abort, do nothing (the database has not been changed).
* On a system restart after a failure, REDO the log.

**Immediate Update**

Immediate update, or UNDO/REDO, is another algorithm to support ABORT and machine failure scenarios.

* While a transaction runs, changes made by that transaction can be written to the database at any time. However, the original and the new data being written must both be stored in the log BEFORE storing it on the database disk.
* On a commit:
* All the updates which have not yet been recorded on the disk is first stored in the log file and then flushed to disk.
* The new data is then recorded in the database itself.
* On an abort, REDO all the changes which that transaction has made to the database disk using the log entries.
* On a system restart after a failure, REDO committed changes from log.

**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**

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.

1. **Transaction Roll back (Undo) and Roll forward**

The process of undoing changes done to the disk under immediate update is frequently referred to as rollback.

* Where the DBMS does not prevent one transaction from reading uncommitted modifications (a `dirty read') of another transaction (i.e. the uncommitted dependency problem) then aborting the first transaction also means aborting all the transactions which have performed these dirty reads.
* as a transaction is aborted, it can therefore cause aborts in other dirty reader transactions, which in turn can cause other aborts in other dirty reader transaction. This is referred to as `cascade rollback'.

It is also possible to keep a separate journal of all modifications to a database management system. (sometimes called after images). This is not required for rollback of failed transactions but it is useful for updating the database management system in the event of a database failure, so some transaction-processing systems provide it. If the database management system fails entirely, it must be restored from the most recent back-up. The back-up will not reflect transactions committed since the back-up was made. However, once the database management system is restored, the journal of after images can be applied to the database (roll forward) to bring the database management system up to date. Any transactions in progress at the time of the failure can then be rolled back. The result is a database in a consistent, known state that includes the results of all transactions committed up to the moment of failure.

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.



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.

1. **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.

1. **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.

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.