www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 4 April 2015, Page No. 11482-11486

Database Recovery Techniques: A Review

Pradip R. Patel

Information Technology Department, Government Polytechnic,
Sector-26, Gandhinagar-382026, Gujarat, India.
patel_pradip_r@yahoo.com

Abstract: ***Database system has evolved from a specialized computer application to a central component of a modern computing environment, and, as a result, it has become an essential part of computer science. 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, we must 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. Recovery processes vary depending on the type of failure that occurred, the structures affected, and the type of recovery that you perform. Recovery refers to the various strategies and procedures involved in protecting the database against data loss and reconstructing the database after any kind of failure. This paper presents a conceptual level description of various issues and techniques related to recovery.***

Keywords:

1. Introduction

Database [1,2,5] recovery is the process of restoring data that has been lost, accidentally deleted, corrupted or made inaccessible for some reason. The basic unit of recovery in a database system is the transaction [1,2]. It is a unit of program execution that accesses and possibly updates various data items. It is initiated by a user program written in a high-level programming language. A database system must ensure proper execution of transactions despite failures - either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. To ensure integrity of the data, database system must maintain the ACID properties [1] of the transactions. A computer system is subject to failure due to various reasons like disk crash, power outage, software error, a fire in the machine room, even sabotage. Transactions may also fail for various reasons, such as violation of integrity constraints or deadlocks. In event of failure information concerning the database system is lost. Recovery[6] scheme detects failures and restores the database to a state that existed before the occurrence of the failure.

2. Failure Classification

Several types of failure [1] can halt the normal operation of a database. For some of these failures, recovery is automatic and requires little or no action on the part of the database user or database administrator. Each of these failures needs to be dealt with in a different manner. There are mainly three types of failures: transaction failures, system failures, and media failures.

2.1. Transaction Failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Logical error and System error may cause a transaction to fail.

- Logical error: where a transaction cannot complete because of it has some code error or any internal error condition.
- System error: where the database system itself terminates an active transaction because database is not able to execute it or it has to stop because of some system condition. Deadlock is example of system error.

2.2. System Crash

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supplies, failure of underlying hardware or software failure. Examples may include operating system errors.

2.3. Disk Failure

A disk failure occurs when any part of the stable storage is destroyed. Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

In order to determine how system should recover from failures, we need to identify failure modes of data storage devices. We must also consider how these failure modes affect the contents of the database. We can then propose recovery algorithms [3,6] to ensure database consistency and transaction atomicity despite failures.

3. Storage Media and Access

Various data items in the database [2] may be stored and accessed in a number of different storage media [1]. We must gain a better understanding of these storage media and their access methods.

3.1. Storage Types

Storage media[1] can be distinguished by their relative speed, capacity, and resilience to failure. They are classified as volatile, non-volatile and stable storage.

- Volatile storage: As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.
- Non-volatile storage: These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include, hard disks, magnetic tapes, flash memory, non-volatile (battery backed up) RAM.
- Stable storage: Information residing in stable storage is never lost. Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.

3.2. Data Access

The database is stored in non-volatile storage, and is partitioned into fixed-length storage units called blocks[1]. Blocks are the basic units of data transfer to and from disk. It may contain several data items. Transactions input information from the disk to main memory, and then output the information back onto the disk. The input and output operations are done in block units. The blocks on the disk are referred to as physical blocks. The temporarily stored blocks in main memory are known as buffer blocks.

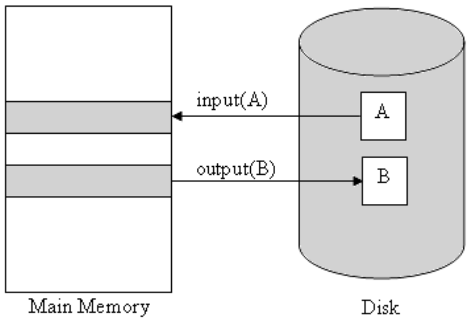

Figure 1: Block Storage Operations.

As depicted in Fig. 1, Blocks are moved between disk and main memory are done through the following two operations:

- input(B) transfers the physical block B to main memory.
- output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

Each transaction has a private work area in which copies of all the data items accessed and updated are kept. The system creates this work area when the transaction is initiated; the system removes it when the transaction either commits or aborts. Each transaction interacts with the database system by transferring data to and from its work area to the system buffer using two operations: read and write.

4. Recovery and Atomicity

Consider example of banking system and transaction T_i that transfers Rs.100 from account X to account Y, with initial values of A and B being Rs.500 and Rs.1000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the outcome of the transaction; thus, we could invoke one of two possible recovery procedures:

- Reexecute Ti. This procedure will result in the value of X becoming Rs.300, rather than Rs.400.
- Do not reexecute Ti. The current system state has values of Rs.400 and Rs.1000 for X and Y, respectively.

In either case, the database is left in an inconsistent [4] state, and thus this simple recovery scheme does not work. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made. To achieve our goal of atomicity, we must first output information describing the modifications to stable storage, without modifying the database itself.

5. Log-Based Recovery

The log [1,7] is widely used for storing database modifications. It is a sequence of log records, recording all the update activities in the database. Among several types of log records[1], an update log record describes a single database write. It has following fields:

- Transaction identifier – It uniquely identifies the transaction that performed the write operation.
- Data-item identifier – It uniquely identifies the data item written.
- Old value - It is the value of data item before write operation.
- New value - It is the value of data item after write operation.

Various log records used to record major events during transaction processing, such as the start of a transaction and the commit or abort of a transaction are as following.

- < Ti start>. Transaction Ti has started.
- < Ti commit>. Transaction Ti has committed.
- < Ti, Xj, V1, V2>. Transaction Ti has performed a write on data item Xj . Xj had value V1 before the write, and will have value V2 after the write.
- < Ti abort>. Transaction Ti has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo[6,14] a modification that has already been output to the database. We undo it by using the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.

5.1. Deferred Database Modification

The deferred-modification technique[1,8] ensures atomicity of transaction by recording all database modifications in the log, but postponing (or deferring) the execution of all write operations of a transaction until the transaction partially commits (i.e. the final action of the transaction has been executed). When a transaction partially commits, the information on the log is used in executing the deferred writes.

If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.

The execution of transaction T_i proceeds as follows. Before T_i starts its execution, a record <T_i start> is written to the log. A write(X) operation by T_i results in the writing of a new record to the log. Finally, when T_i partially commits, a record <T_i commit> is written to the log. When transaction T_i partially commits, the records associated with it in the log are used in executing the deferred writes. Since a failure may occur while this updating is taking place, we must ensure that, before the start of these updates, all the log records are written out to stable storage. Once they have been written, the actual updating takes place, and the transaction enters the committed state. Observe that only the new value of the data item is required by the deferred modification technique. Thus, we can simplify the general update-log record structure that we saw in the previous section, by omitting the old-value field.

Let's reconsider our simplified banking system. Let T_0 be a transaction that transfers Rs.100 from account X to account Y. And T_1 be a transaction that withdraws Rs.50 from account Z.

```
T0:  read(X);
     X := X - 100;
     write(X);
     read(Y);
     Y := Y + 100;
     write(Y).

T1:  read(Z);
     Z := Z -  50;
     write(Z).
```

Suppose that these transactions are executed serially, in the order T_0 followed by T_1, and that the values of accounts X, Y, and Z before the execution took place were Rs.500, Rs.1000, and Rs.1500, respectively. The portion of the log containing the relevant information on these two transactions appears in Figure 2.

```
<T0 start>
<T0, X, 400>
<T0, Y, 1100>
<T0 commit>
<T1 start>
<T1, Z, 1450>
<T1 commit>
```

Figure 2: Database log corresponding.

There are various orders of execution of T_0 and T_1 in which the actual outputs can take place to both the database system and the log. One such order appears in Figure 3.

```
    Log           Database
<T0 start>
<T0, X, 400>
<T0, Y, 1100>
<T0 commit>
                   X = 400
                   Y = 1100
<T1 start>
<T1, Z, 1450>
<T1 commit>
                   Z = 1450
```

Figure 3: System Log and Database

Note that the value of A is changed in the database only after the record <T_0, X, 400> has been placed in the log. Using the log, the system can handle any failure that results in the loss of information on volatile storage. The recovery scheme[14] uses the following recovery procedure:

- redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

The set of data items updated by T_i and their respective new values can be found in the log. After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction T_i needs to be redone if and only if the log contains both the record <T_i start> and the record <T_i commit>. Thus, if the system crashes after the transaction completes its execution, the recovery scheme uses the information in the log to restore the system to a previous consistent state after the transaction had completed.

5.2. Immediate Database Modification

The immediate-modification technique [1] allows database modifications to be output to the database while the transaction is still in the active state. In the event of a crash or a transaction failure, the system must use the old-value field of the log records to restore the modified data items to the value they had prior to the start of the transaction. Since the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage. We therefore require that, before execution of an output(Y) operation, the log records corresponding to Y be written onto stable storage. As an illustration, let us reconsider our simplified banking system, with transactions T_0 and T_1 executed one after the other in the order T_0 followed by T_1. The portion of the log containing the relevant information concerning these two transactions appears in Figure 4.

```
<T0 start>
<T0, X, 500, 400>
<T0, Y, 1000, 1100>
<T0 commit>
<T1 start>
<T1, Z, 1500, 1450>
<T1 commit>
```

Figure 4: System Log Corresponding

Figure 5 shows one possible order in which the actual outputs took place in both the database system and the log as a result of the execution of T_0 and T_1.

```
      Log                 Database
<T0 start>
<T0, X, 500, 400>
<T0, Y, 1000, 1100>
                          X = 400
                          Y = 1100
<T0 commit>
<T1 start>
<T1, Z, 1500, 1450>
                          Z = 1450
<T1 commit>
```

Figure 5: State of System Log and Database

Using the log, the system can handle any failure that does not result in the loss of information in nonvolatile storage. The recovery scheme uses two recovery procedures:

- undo(Ti) restores the value of all data items updated by transaction Ti to the old values.
- redo(Ti) sets the value of all data items updated by transaction Ti to the new values.

After a failure has occurred, the recovery scheme consults the log to determine which transactions need to be redone, and which need to be undone. Transaction Ti needs to be undone if the log contains the record <T_i start>, but does not contain the record <T_i commit>. T_i needs to be redone if the log contains both the record <T_i start> and the record <T_i commit>.

5.3. Checkpoints

After system failure we must search entire log to determine transactions that need to be redone and undone. This search process becomes time consuming. Second, most of the transactions that need to be redone have already written their updates into the database. It will cause recovery to take longer. Checkpoints[1, 11] reduce these problems. The system periodically writes into log a <checkpoint> record. It means all the previous logs are removed and stored permanently in storage disk. Checkpoint declares a point before which the DBMS was in consistent state and all the transactions were committed.

During recovery, recovery scheme examine the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place by searching the log backward, from the end of the log, until it finds the first <checkpoint> record; then it continues the search backward until it finds the next <T_i start> record. This record identifies a transaction T_i. The redo and undo operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i. Let us denote these transactions by the set T. The remainder of the log can be ignored and removed. For the immediate-modification technique, recovery operations are: For all transactions T_k in T if <T_k commit> record is not in the log, execute undo(T_k) and if <T_k commit> appears in the log, execute redo(T_k). Obviously, the undo operation does not need to be applied for deferred-modification technique.

6. Shadow Paging

An alternative to log-base recovery is to use a system of shadow paging[11]. This is where the database is divided into pages that may be stored in any order on the disk. In order to identify the location of any given page, we use something called a page table as in Figure 6. The page table[1] has multiple entries - one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on. Figure 6 shows that the logical order of database pages does not need to correspond to the physical order in which the pages are placed on disk.

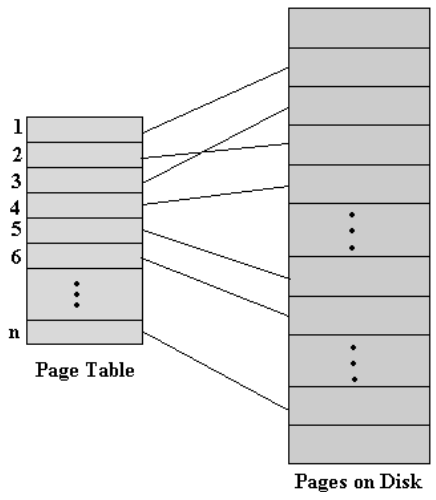

Figure 6: Sample Page Table.

During the life of a transaction two page tables are maintained, one called a shadow page table and current page table. When a tranasaction begins both of these page tables are identical. During the lifetime of a transaction the shadow page table doesn't change at all. However during the lifetime of a transaction changes may be made to current page table.

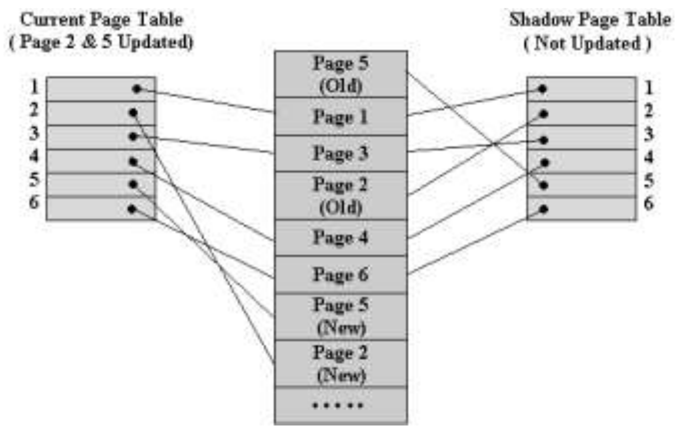

Figure 7: Shadow and Current Page Tables.

Looking at Figure 7 we see how these tables appear during a transaction. As we can see the shadow page table shows the state of the database just prior to a transaction, and the current page table shows the state of the database during or after a transaction has been completed. We now have a system whereby if we ever want to undo the actions of a transaction all we have to do is to recover the shadow page table to be the current page table. As such this makes the shadow page table particularly important, and so it must always be stored on stable storage. On disk we store a single pointer location that points to the address of the shadow page table. This means that to swapt the shadow table for the current page table (commiting the data) we just need to update this single pointer. Figure 7 shows the shadow and current page tables for a transaction performing a write to the second and fifth pages of a database. To commit a transaction, we must do the following: First, ensure that all buffer pages in main memory that have been changed by the transaction are output to disk. Second, output the current page table to disk. Finally, output the disk address of the current page table to the fixed location in stable storage containing the address of the shadow page table. This action overwrites the address of the old shadow page table. Therefore, the current page table has become the shadow page table, and the transaction is committed.

Shadow paging offers several advantages over log-based techniques as described in [1].

- Reduced Overhead: The overhead of log-record output is eliminated.
- Fast Recovery: Recovery [11] from crashes is faster since no undo or redo operations are needed.

However, as explained in [1], there are some drawbacks of this technique:

- Commit overhead: The commit of a single transaction using shadow paging requires multiple blocks to be output—the actual data blocks, the current page table, and the disk address of the current page table. Log-based schemes need to output only the log records, which, for typical small transactions, fit within one block.
- Data fragmentation: Shadow paging causes database pages to change location when they are updated. As a result we lose the locality property of the pages.
- Garbage collection: Each time that a transaction commits, the database pages containing the old version of data changed by the transaction become inaccessible. Such pages are considered garbage, since they are not part of free space and do not contain usable information. Periodically, it is necessary to find all garbage pages, and to add them to the list of free pages. This process, called garbage collection[9,10], imposes additional overhead and complexity on the system.

7. Conclusion

Database system is an important part of computer system that is used to store data. There are mainly three ways that a system fails: transaction failures, system failures, and media failures. Storage media which is used to store database can be classified as volatile, non-volatile and stable storage. Information stored in volatile storage does not usually survive system crashes while information stored in non-volatile storage survives system crashes. Information residing in stable storage is never lost. When failure occurs, information stored in database is lost and database becomes inconsistent. Recovery scheme is an essential part of a database system which can restore the database to the consistent state that existed before the failure. Log-based schemes and Shadow paging are widely used recovery techniques. There are two log-based schemes, deferred-modifications and immediate-modifications. In comparison to log-based schemes, shadow paging technique is fast and has reduced recovery overhead. However, shadow-page technique has several drawbacks like large commit overhead, data fragmentation and creation of garbage data.

References

[1] Silberschatz, A., H.F. Korth and S. Sudarshan. *Database System Concepts*. Boston, MA, McGraw-Hill, 2002.

[2] Philip A. Bernstein and Eric Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann Publishers.

[3] Vijay Kumar and Albert Burger, "*Performance Measurement of Main Memory Database Recovery Algorithms Based on Update-in-Place and Shadow Approaches*", IEEE Transactions on Knowledge and Data Engineering, Vol 4, No. 0, December 1992.

[4] Date, C.J., *Database in Depth*, O'Reilly Publication, 2005.

[5] Paul Beynon, Davies, *Database Systems - Third Edition*. Published by Palgrave Macmillan, 2004.

[6] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, Fellow, IEEE, and S. Sudarshan, "*Detection and Recovery Techniques for Database Corruption*", IEEE Transactions On Knowledge And Data Engineering, Vol. 15, No. 5, September/October 2003.

[7] J. Eliot and B. Moss, "*Log-Based Recovery for Nested Transactions*", Department of Computer and Information Science, University of Massachusetts, Amherst.

[8] Asit Dan, Member, IEEE, Philip S. Yu, Fellow, IEEE, and Anant Jhingran, "*Recovery Analysis of Data Sharing Systems under Deferred Dirty Page Propagation Policies*", IEEE Transactions on Parallel And Distributed Systems, Vol. 8, No. 7, July 1997.

[9] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang And David B. Johnson, "*A Survey Of Rollback-Recovery Protocols In Message-Passing Systems*", ACM.

[10] Jonathan E. Cook, Alexander L. Wolf and Benjamin G. Zorn, "*A Highly Effective Partition Selection Policy for Object Database Garbage Collection*", IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 1, January/February 1998.

[11] Victor F. Nicola, And Johannes M. Van Spanje, "*Comparative Analysis of Different Models of Checkpointing and Recovery*", IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990.

[12] Min-Sheng Lin and Deng-Jyi Chen, "*The Reliability Problem in Distributed Database Systems",* International Conference on Information, Communications and Signal Processing, September 1997.

[13] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[14] An Oracle White Paper, "*Very Large Database (VLDB) Backup & Recovery Best Practices*", July 2008.