

Project Report for  
**Log Based Recovery System**

Prajakta S. Kalekar (04329008)      Shruti P. Mahambre (04429801)  
Anirudha U. Bodhankar (04329003)  
Indian Institute of Technology, Powai, Mumbai -400076

October 25, 2004

## **Abstract**

The purpose of Database Recovery is to bring the database into the last consistent state, which existed prior to the failure. The most widely used technique for Database recovery is Log based recovery.

For recovery from any type of failure data values prior to modification and the new value after modification are required. These values and other information is stored in a sequential file called Transaction log. To maintain atomicity, a transactions operations are redone or undone. Operations of committed transactions are redone whereas operations of uncommitted or aborted transactions are undone. Checkpoints are used to reduce the number of log records that the system must scan when recovering from a crash.

The project aims at building a such a Log based recovery system for a database employing immediate database modification.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                             | <b>4</b>  |
| <b>2</b> | <b>System overview</b>                          | <b>4</b>  |
| 2.1      | Architectural diagram . . . . .                 | 4         |
| 2.2      | Input . . . . .                                 | 4         |
| <b>3</b> | <b>Implementation details</b>                   | <b>6</b>  |
| 3.1      | Data Structures . . . . .                       | 6         |
| 3.2      | Recovery Algorithm . . . . .                    | 6         |
| 3.2.1    | Undo/Redo of Update operation . . . . .         | 7         |
| 3.2.2    | Undo/Redo of Insert operation . . . . .         | 7         |
| 3.2.3    | Undo/Redo of Delete operation . . . . .         | 8         |
| <b>4</b> | <b>Directory structure</b>                      | <b>8</b>  |
| <b>5</b> | <b>Miscellaneous</b>                            | <b>10</b> |
| 5.1      | Integration with the existing code . . . . .    | 10        |
| 5.2      | Performance . . . . .                           | 10        |
| 5.3      | Compilation and execution of the code . . . . . | 10        |
| 5.4      | Tools used . . . . .                            | 10        |
| 5.5      | Assumptions . . . . .                           | 10        |
| <b>6</b> | <b>Conclusion and Future work</b>               | <b>11</b> |
| <b>7</b> | <b>References</b>                               | <b>11</b> |

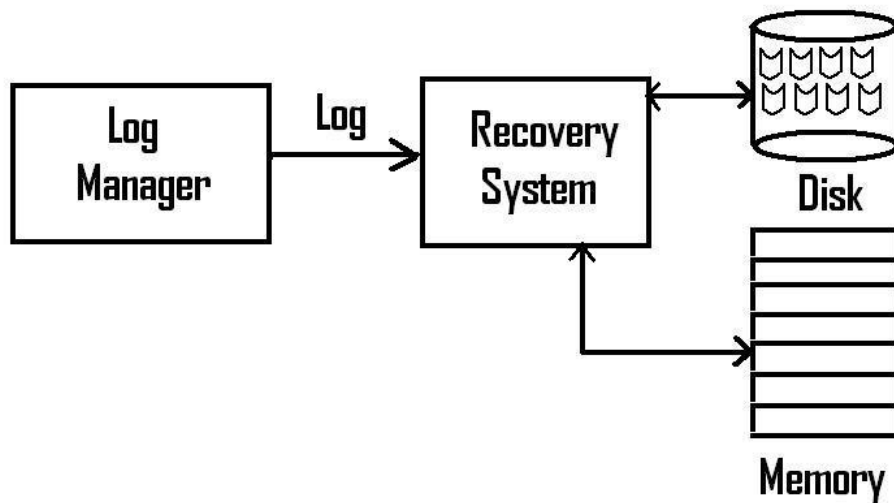
# 1 Introduction

The project aims at building a *Log based recovery system for a database employing immediate database modification*. The project takes into account memory constraints in light-weight devices. It performs log based recovery on the log generated by the log manager. The types of log records supported are :

- Transaction start
- Transaction commit
- Transaction abort
- Update record
- Insert record
- Delete record

## 2 System overview

### 2.1 Architectural diagram



### 2.2 Input

The input to the system is a log file generated by the log manager. The supported log records and their structure is as follows:

- **Transaction start**  
It has the following format  
**s TID**  
where  
s is the start log record type specifier

TID is the transaction ID

**Note : The type specifier is of 1 byte**

- **Transaction commit**

It has the following format

**c TID**

where

*c* is the commit log record type specifier

TID is the transaction ID

- **Transaction abort**

It has the following format

**a TID**

where

*a* is the abort log record type specifier

TID is the transaction ID

- **Update record**

It has the following format

**u TID relFile relFileOff datatype oldVal newVal**

where

*u* is update log record type specifier

*TID* is the transaction ID

*relFile* path to the relation file whose record is to be updated

*relFileOff* offset in bytes within *relFile* of the data item to be updated

*datatype* datatype of the data item to be updated

*oldVal* Value of the data item prior to the update operation

*newVal* Value of the data item after the update operation

The possible values of *datatype* are:

- Float Specified by *f*

For example

u 2 IIT 112 f 89.78 364.551636

- String Specified by *s len*

For example

u 1 IIT 102 s10 PROD KReSIT

Here 's' stands for string and 10 is the length of the string.

- Char Specified by *c*

- Integer Specified by *i*

- **Insert record**

It has the following format

**i TID catFileOff relFile relFileOff recLen datatype1 dataVal1 [datatypeN dataValN]**

where

*i* is insert log record type specifier

*TID* is the transaction ID

*catFileOff* : The number of tuples of each of the relations is stored in a catalog file. *catFileOff* gives the offset in the catalog file of this field for the relation.

*relFile* The path to the relation file into which the record is to be inserted.

*relFileOff* The offset within *relFile* where the record would be inserted. This field is used for undo of the insert operation. For redo, this field is ignored, since the record is appended to the end of the file.

*recLen* Length of the record to be inserted.

*datatypeN* and *dataValN* : The record to be inserted consists of one or more attribute values. *datatypeN* specifies the datatype of the Nth attribute value *dataValN*.

- **Delete record**

It has the following format

**d TID catFileOff relFile relFileOff recLen datatype1 dataVal1 [datatypeN dataValN]**

where

*d* is delete log record type specifier

The rest of the log record fields are the same as that of an insert record.

### 3 Implementation details

#### 3.1 Data Structures

The Data Structures used in log based recovery are the Redo and Undo Lists. These are represented as dynamic arrays. These arrays have been initialized to a MAXSIZE. If required the arrays can be resized each time by MAXSIZE units.

#### 3.2 Recovery Algorithm

Following steps are carried out in the log based recovery process:

1. **Find the first log record to be processed**

The log file is scanned in the backward direction, till the occurrence of the checkpoint record. A list of transactions at the checkpoint is formed. If the transaction list is empty, the offset of the record immediately following the checkpoint is returned. Else the log file is scanned backwards till the start transaction record of each of the transactions in the checkpoint list is found. This offset is returned.

2. **Create the Redo Undo Lists.**

The log file is scanned in the forward direction. When a transaction *start* occurs, insert the transaction id in the undo list. When a transaction *commit* occurs, if the transaction id is present in the undo list - Remove it from the undo list and add this id to the redo list. Ignore any other record.

3. **Undo Transactions**

Scan the log file in the backward direction. For every update, insert or delete record, check if the transaction occurs in the undo list. If so, undo the operation of that transaction.

#### 4. Redo Transactions

Scan the log file in the forward direction. For every update, insert or delete record, check if the transaction occurs in the redo list. If so, redo the operation of that transaction.

##### 3.2.1 Undo/Redo of Update operation

As previously mentioned, an update log record is as follows:

**u TID relFile relFileOff datatype oldVal newVal**

- Redo

In order to redo an update operation, following steps are taken:

1. Open the **relFile** and fetch to the offset **relFileOff**
2. Depending on the **datatype**, write the value **newVal** to the **relFile**

- Undo

In order to undo an update operation, following steps are taken:

1. Open the **relFile** and fetch to the offset **relFileOff**
2. Depending on the **datatype**, write the value **oldVal** to the **relFile**

##### 3.2.2 Undo/Redo of Insert operation

As previously mentioned, an insert log record is as follows:

**i TID catFileOff relFile relFileOff recLen datatype1 dataVal1 [datatypeN dataValN]**

- Redo

In order to Redo an update operation, the following steps are taken:

1. Open the catalog file. Fetch to the **catFileOff**. Read the value of number of tuples( $n$ ) in the relation.
2. Open the **relFile**. Compute the offset where the record is to be inserted  $n * recLen$ . Seek to the computed offset.
3. Write each **dataVal** to the **relFile**
4. Update the value of number of tuples in the relation from  $n$  to  $n + 1$  in the catalog file.

- Undo

The undo of an insert is equivalent to a delete. In order to delete a record, it is replaced by the last record in the file. The count of tuples of the relation in the catalog file is decremented to indicate that a record has been deleted. The following steps are taken:

1. Open the catalog file. Fetch to the **catFileOff**. Read the value of number of tuples( $n$ ) in the relation.
2. Open the **relFile**. Compute the offset *replOff* of the last record in the file -  $(n - 1) * recLen$ . Seek to the computed offset.
3. Read the record at the *replOff* and write it to the **relFileOff**
4. Update the value of number of tuples in the relation from  $n$  to  $n - 1$  in the catalog file.

### 3.2.3 Undo/Redo of Delete operation

The undo of a delete operation is the same as the redo of an insert operation and the redo of a delete operation is the same as the undo of an insert operation.

## 4 Directory structure

- **log-recovery**

This is the main directory containing the files required for generating log-recovery.exe. It contains the following files:

1. **Makefile**

Used for compilation of the code.

2. **common.h**

It contains all global declarations and definitions.

3. **main.c**

The main file that contains the function main() that invokes the log recovery process

4. **createUndoRedo.c**

This file contains the functions required for creating the redo and undo lists. The functions are:

- (a) **prepareRedoUndoList()**

This is the main function for preparing the redo and undo lists.

- (b) **addToRedoList()**

The function adds the specified transaction ID to the Redo list.

- (c) **addToUndoList()**

The function adds the specified transaction ID to the Undo list.

- (d) **removeFromUndoList()**

The function removes the specified transaction ID from the Undo list.

- (e) **printUndoList()**

Prints the undo list to the console

- (f) **printRedoList()**

Prints the redo list to the console

5. **redo.c** This file contains functions related to the redo of update, insert and delete operations. The functions are:

- (a) **toRedo()**

Given a transaction ID, this function is used to determine whether the operation is to be redone or not (depending on whether the transaction ID appears in the redo list)

- (b) **redo()**

The main function used for performing redo operations that scans the file in the forward direction and performs a redo of the operations corresponding to the transactions in the redo list.

- (c) **updateRedo()**

This function is used to redo a update operation.

- (d) **insertRedo()**

This function is used to redo a insert operation.

(e) **deleteRedo()**

This function is used to redo a delete operation.

6. **undo.c** This file contains functions related to the undo of update, insert and delete operations. The functions are:

(a) **toUndo()**

Given a transaction ID, this function is used to determine whether the operation is to be undone or not (depending on whether the transaction ID appears in the undo list)

(b) **undo()**

The main function used for performing undo operations that scans the file in the backward direction and performs a undo of the operations corresponding to the transactions in the undo list.

(c) **updateUndo()**

This function is used to undo a update operation.

(d) **insertUndo()**

This function is used to undo a insert operation.

(e) **deleteUndo()**

This function is used to undo a delete operation.

7. **common-functions.c**

This file contains functions that are common to the redo and undo operations. The functions are:

– **readPrevLine()**

Function used to read the file in the backward direction.

– **getFirstRecToProcess()**

Function that returns the offset in the log file of the first record to process.

• **testcase**

This directory contains the relation files and log files used for testing.

The relation files are:

1. *IIT*
2. *Student*

The log files are:

1. *log-iit.log* This log file contains the update log records for updates on the IIT relation.
2. *log-std.log* This log file contains the update log records for updates on the Student relation.
3. *redoInsDel.log* This log file contains the insert/delete log records to be redone.
4. *undoInsDel.log* This log file contains the insert/delete log records to be undone.

The *student-log-description* file contains a description of which records of the log-std.log file would be redone and which would be undone. It contains the expected output.

The *iit-log-description* file contains a description of which records of the log-iit.log file would be redone and which would be undone. It contains the expected output.

The *simp-relations* file is the catalog file.

## 5 Miscellaneous

### 5.1 Integration with the existing code

The log based recovery system is compatible with the existing simpdb. Given an input log file, and the catalog and relation files used in simpdb, the recovery system successfully performs undo and redo operations of DML statements.

In order to completely integrate the system with simpdb, the transaction manager is required to generate a log file (in the correct format). This can be achieved by making appropriate changes in the *update-rel.c*, *rel-insert.c* and the *delete-tuple.c* files.

### 5.2 Performance

The log based recovery system employs the following strategies for performance improvement:

1. The log file is kept as compact as possible. The field specifying the the log record type occurs in each record of the log file. Hence this field is kept only 1 byte long. A compact log file size allows the log file to be efficiently read (for processing) in limited memory databases (like simpdb).
2. The usual approach for deleting a record from a file is to copy all but the deleted record to a new file, followed by renaming of the file. However in our system, a record is deleted by replacing it by the last record in the file, and then decrementing the count of tuples of the relation in the catalog file.

### 5.3 Compilation and execution of the code

The directory contains a Makefile for compilation of the code. Perform the following steps:

1. Execute *make clean* to get rid of all the intermediate files from the directory.
2. Execute *make* to generate **logrecovery.exe** in the current directory.
3. Execute *./logrecovery.exe recover.log* where *recover.log* is the name of the log file.
4. In order to run the test cases, please refer to the *readme.txt* file.

### 5.4 Tools used

The gnu C compiler has been used.

### 5.5 Assumptions

1. Immediate database modification technique is used.
2. Write-ahead logging is used.
3. If concurrent transactions are to be supported, then they must observe strict or rigorous 2-phase locking protocol.
4. In case of index structures the index-based concurrency protocol is supported. This means that the logs must be generated using the tags as specified above for logical logs.

5. Check-pointing is supported and recommended. However it is assumed that check-pointing is non-fuzzy.
6. The recovery system makes 2 passes over the log. In the first pass, it creates the undo and redo lists and in the second pass, the actual undo operations and redo operations are executed.

## 6 Conclusion and Future work

Since the system is compatible with the existing simpdb code, it can be integrated with simpdb. Undo/redo of DDL statements can be taken up as a part of the future work of the project.

## 7 References

1. Database System Concepts by Silberschatz, Korth and Sudarshan
2. Fundamentals of Database Systems by Elmasri and Navathe
3. picoDBMS paper
4. Efficient data management on Light weight computing devices by Rajkumar Sen and Krithi Ramamritham
5. Recovery system for Main memory database by Anurag Gupta and Han-Yin Chen