CS346 EX Component Proposal
Jichan Park

1. Introduction

I'm going to implement **recovery mechanism** for RedBase. Proposed tasks for this project include:

- Modify the parser (parse.y) so that it understands `Begin Transaction`, `Commit Transaction`, `Abort Transaction`, and `Checkpoint` commands
- Implement Write-Ahead Logging (WAL) that logs changes to the database
- Implement undo recovery when `Abort Transaction` is called or the database is found to be in an inconsistent state when opened

To keep the project scope manageable, I will make the following simplifying assumptions:

- All transactions will run in a **serial** order (i.e. no concurrent transactions)
- We assume that NO index exists on tables that are modified and recovered
- Only `UPDATE`, `INSERT`, `DELETE` statements are logged and successfully recovered. (i.e. other SM component operations (except for `LOAD`) will not be recovered)

2. Design Decision

To decrease response time and increase throughput of the buffer pool, I will support no-force, steal policies (like ARIES):

- Do NOT force write to disk at commit
- Allow stealing buffer-pool frames from an uncommitted transaction

To enable recovery under these policies, we do Write-Ahead Logging (WAL):

- Flush all log records before commit
- Force the log record to disk before an update gets flushed to disk

The recovery process then roughly involves:

- Redoing logged actions from beginning to end (Idempotence of REDO is ensured by comparing pageLSN with LSN)
- Undoing logged actions of the last transaction if it is incomplete (Idempotence of UNDO is ensured by using CLR that is never undone)

3. Overall Architecture & Changes to Existing Components

The relationship between LG component and other components are as follows:

- LG_Manager uses PF component to create file and allocate pages for the log. PF component calls LG component to flush the log when a disk write is about to happen.
- LG_Manager uses RM component to undo changes to the record file.
- SM Component calls LG_Manager to create log file upon opening database and destroying it upon closing database.
- Parser, SM Component and QL Component call LG_Manager to start / abort / commit transactions and also make a checkpoint

- **PF_Component**
  - When `PF_Manager::CreateFile("log")` is called, save unixfd for the log file
  - Before a page (not in a log file) is written to disk (in `PF_BufferMgr::WritePage()`), ForcePages() of the log file

- **RM_Component**
  - Add pageLSN field to RM_PageHdr (header of each record page)
  - When a record is updated (`RM_FileHandle::UpdateRec(), InsertRec(), DeleteRec()`), `LG_Manager::InsertLogRec()` is called and pageLSN is updated as LSN of the new log record
  - Add `RM_FileHandle::InsertRecAtRid()` to directly insert a record at certain position (for physical redo)

- **SM_Component**
  - When a database is opened, call `LG_Manager::CreateLog()`
  - When a database is closed, call `LG_Manager::DestroyLog()`
  - `SM_Manager::Load()` should automatically be wrapped in a standalone transaction. Call beginT() at the beginning and commitT() at the end.

- **QL_Component**
  - Individual database modification statements (`QL_Manager::Insert(), Update(), Delete()`) should automatically be wrapped in a standalone transaction.
  - Therefore, call beginT() at the beginning and commitT() at the end of such operations

- **Parser**
  - Modify the parser so that when user types `Begin Transaction`, `Commit Transaction`, `Abort Transaction`, and `Checkpoint`, it calls `LG_Manager::BeginT()`, `LG_Manager::CommitT()`, `LG_Manager::AbortT()` and `LG_Manager::Checkpoint()` respectively

4. Interface Specification

```
enum LogType {
	Insert, Delete, Update, Commit, Abort, End, CLR
}

/* LSN is uniquely defined as PageNum and
offset of the log record in a log file */
struct LSN {
	PageNum pn;
	int offset;
}

/* Page header in each log file page to keep track of space used */
struct LogPageHeader {
	int numLogRecords;
	int availableSpace;
}

/* Log Records for COMMIT, ABORT and END */
struct LogRecord {
	LSN lsn;
	Int XID;
	LogType type;
	LSN prevLSN;
}

/* Full Log Records for UPDATE and CLR */
struct FullLogRecord {
	LSN lsn;
	Int XID;
	LogType type;
	LSN prevLSN;

	int TupleSize;    // Size of the tuple modified/inserted or deleted
	char[MAXNAME] relName;
	RID rid;
	LogType undoType; // Only for CLR; Type of the log record we are undoing
	LSN undonextLSN;  // Only for CLR
}
```

< Note >
Before images and after images of the entire tuple follows a FullLogRecord. Depending on the type of log record, 0 to 2 copies will be saved:
If type == ‘Update’: both old value and new value
If type == ‘Insert’: new value only
If type == ‘Delete’: old value only

If type == ‘CLR’, since undo is done only once, there is no need to store old value:
If type == ‘CLR’ and undoType == ‘Update’ or ‘Delete’: new value only
If type == ‘CLR’ and undoType == ‘Insert’: none

```
LG_Manager Class {
Public:
	LG_Manager(PF_Manager &pfm, RM_Manager &rmm);
	~LG_Manager();

	RC CreateLog();
	RC DestroyLog();
	RC InsertLogRec(FullLogRecord logRec);
	RC FlushLog();
	RC BeginT();
	RC CommitT();
	RC AbortT();
	RC Checkpoint();
	RC Recover();

Private:
	PF_Manager *pfm_;
	RM_Manager *rmm_;
```

```
        PF_FileHandle logFile_;

        Int bInTransaction;     // Whether there is a transaction running currently
        LSN currXactLastLogRec; // LSN Of last log record (of the current transaction)
        LSN nextLogSlot;        // Where the next log record will be inserted
        int nextXID;            // ID for the next new transaction
}
```

5. Functionality Description

`CreateLog()` creates a file named “log”.
If the log file already exists, it means that the last database session terminated abnormally (without properly destroying the log file). In such case, call `Recover()` to initiate recovery process.
After creating a log file, the function opens the file with PF_FileHandle logFile_.
currXactLastLogRec is set to a null value and nextLogSlot is set to zero.

`DestroyLog()` destroys a file named “log”.

`FlushLog()` flushes pages of a log file to disk. This is called by `commitT()` and `PF_BufferMgr::WritePage()`.

`InsertLogRec(FullLogRecord &logRec)`

1. logRec is prepopulated with type, bInsertFlag, relName, rid, datasize and undonextLSN
2. The function populates logRec with lsn, XID, prevLSN and undonextLSN
3. logRec is inserted to nextLogSlot
4. LogPageHeader is updated for the page in which the record was inserted
5. Update currXactLastLogRec and nextLogSlot

`BeginT()` starts a new transaction.

1. Check if bInTransaction == FALSE (to ensure transactions are serial), then set it to TRUE
2. Set currXactLastLogRec to null value
3. Increment nextXID

`AbortT()` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

1. Check if bInTransaction == TRUE
2. WRITE <ABORT> record to log
3. Follow chain of log records backward via `currXactLastLogRec` then prevLSN field (until it sees <END> record of a previous transaction)
4. Write Compensation Log Record (CLR) with an extra field (undonextLSN) that points to the prevLogRec of the record we are currently undoing
5. Restore the old value, set pageLSN to LSN of CLR
6. WRITE <END> record to log for the transaction undone
7. Set bInTransaction = FALSE

When `CommitT()` is called, all log records are flushed to disk:

1. Check if bInTransaction == TRUE
2. Write <COMMIT> record to log
3. Flush all log records in the log file to disk

4. Write <END> record to log for the transaction committed
5. Set bInTransaction = FALSE

`Checkpoint()` function is called when a user types CHECKPOINT command. It flushes all changes to the database and starts the log anew to prevent the log size from growing too long.

1. Check if `LG_Manager::currXactStatus` == RUNNING (Note that transactions are serial and `CHECKPOINT` is called between transactions).
2. Flush log records to disk (by closing log file)
3. Flush all dirty record pages to disk
4. Call `LG_Manager::DestroyLog()` then `LG_Manager::CreateLog()`

`Recover()` returns the database to a consistent state after crash. It first redoes all changes from the beginning of the log, then undoes the changes of the failed transactions.

Redo Phase

We first repeat log history to reconstruct state at the crash:

1. Follow chain of log records from the beginning sequentially
2. For <UPDATE> and <CLR> records, **if pageLSN < LSN**, reapply logged action and set pageLSN to LSN

Undo Phase

Since we assume that transactions are serial, the only transaction that might be incomplete and might need to be undone is the last transaction in log history. Check the last log record,

If type == 'CLR', set toUndoLSN = undonextLSN;
If type == 'ABORT', set toUndoLSN = prevLSN;
If type == 'INSERT' or 'UPDATE', set toUndoLSN = LSN of the log record;
If type = = 'COMMIT' or 'END', no undo is necessary

1. Starting from toUndoLSN, follow chain of log records backward via prevLogRec field
2. For each <UPDATE> and <INSERT> log record encountered, write Compensation Log Record (CLR) with an extra field (undonextLSN) that points to the prevLogRec of the record we are currently undoing
3. Restore the old value, set pageLSN to LSN of CLR
4. Do this until log record of another transaction is encountered (note transactions are serial)
5. WRITE <END> record to log for the transaction undone