Log Manager Design

1. Log Manager

The log manager will accept the logging request, and do the corresponding writing to the buffer. Also, it should support flushing the current log to disk in order to implement Write Ahead Logging (WAL). Since the log will be written in sequence, and the force-to-disk step should just include the correct amount of logging, we need to add lock to LSN and buffer to make it thread-safe.

2. The Log

The log will have two parts, namely header and payload.

Header contains the length (4 bytes) and type of the log (1 byte), the transaction ID (8 bytes), current LSN (8 bytes), previous LSN (8 bytes) and previous LSN that is for the same transaction (8 bytes).

Payload will be different for different types of logs, which should at least support UPDATE, INSERT, DELETE, COMMIT, ABORT and COMPENSATION. These detail setting will be covered in the introduction of LogRecord class.

LSN will be an unsigned 64-bit integer which could be divided by two parts: the first 32 bits denote the log file name (such as LOG_0000), and the last 32 bits denote the offset inside the file.

3. Classes

We will need three basic classes: LogManager, LogRecord, LogTable

a) LogManager:

This class is the brain of the log manager, and the public method here will be the API to other objects.

Private fields: current LSN, previous LSN, log table and buffer.

Public methods:

sendLogRequest():

This method will be overloaded to support different kind of log type. Basically it will create a LogRecord object, acquire a lock on LSN and buffer, form the header and the payload from LogRecord and write them to the buffer. The current LSN, previous LSN and log table will also be updated. After all the work, it will release both locks on LSN and buffer.

Also, it will call forceToDisk() if the current LSN's first 32 bits might change after updating LSN, since next time it will write on a new file, we need to clean the header before that.

sendForceRequest():

This method will acquire a lock on LSN and buffer to avoid inconsistency, and call forceToDisk() to flush the buffer to the disk. After writing, release the locks.

Private methods:

flushToDisk():

this method will simply flush all the buffer to disk, and clean the buffer after writing.

b) LogRecord

This class is a base class for all the log record, including update, insert, delete, commit, abort and compensation. All the methods here would be private, and befriended with LogManager. The common fields of all is log type and transaction number. Every different kind of log record will have their own fields:

Update: this field will contain the location, the pre-image and the post-image of the updated tuple.

Note that in the update operation the storing location might change, we should also record the possible other location storage block for this operation.

Insert: this field will contain the location of the destination and the image of the tuple.

Delete: this field will contain the location and the image of the tuple

Commit/Abort: no special field needed.

Compensation: as ARIES, we need to record what has been undone, and the last LSN that has not been undone.

Methods:
payload(): Return the payload of this log record.
header(): Return the transaction ID and log type for log manager to build the header.

c) LogTable

This class is a simple hash table, recording the transaction id and their “previous LSN based on transaction”. All the methods here would be private, and befriended with LogManager.

Methods:
getPrevLSN(): get the previous LSN for the given transaction ID.
update(): update the previous LSN of the given transaction ID.
delete(): delete the entry of the given transaction ID (will be called upon commit/abort).

4. Concerns:
a) Payload format depends on recovery manager (the one really works on logs).
b) We might need to change the code in the /storage a little bit, since most of the logging might have to be done in the operation for some reasons.

First, some operation might do something that is unknown before calling the methods. For example, update operation might relocate the tuple to another storage block.

Second, some operation only provides a predicate to search for the satisfying tuples. The transaction will not know which tuple would be updated/deleted beforehand.

Third, some operation might fail for some safety check, then we could avoid writing the log for it.