ECE356: Database Systems Winter 2018

Lecture 28 — More Recovery & ARIES

Jeff Zarnett

Fuzzy Checkpointing

Thus far when we have talked about creating a checkpoint assumes that we stop all execution, briefly, to take the snapshot. This sort of thing is not unheard of in programs, such as the "stop the world" garbage collector behaviour in Java. It might be, however, undesirable or even unacceptable: for a sufficiently large buffer it might take an usually long time. Therefore we would like to modify the checkpoint system to avoid having to halt all execution.

The term for a checkpoint that isn't such a clean division between here and there is called a *Fuzzy Checkpoint*. A fuzzy checkpoint allows updates to start once the checkpoint has been created but before the modified buffer blocks are written to disk [EN11]. There's still a pause in execution while the checkpoint is created, but that should be the fast part.

This does potentially present a problem: the checkpoint data is written to disk before the blocks are written and we could have a crash before all the blocks are written to disk. What do we do then? Keep track of the last completed checkpoint, and don't update the last completed checkpoint marker until all blocks have been output.

Note that even with fuzzy checkpoints, a block cannot be updated while it is being output to disk (although other blocks can be modified) [SKS11].

Recovering from Loss of Nonvolatile Storage

The recovery routines considered so far are about booting the server up again after a crash or power failure or other temporary outage where nonvolatile storage is unaffected. But bad things will happen to even nonvolatile storage. So we need to take copies of the data to account for the fact that even supposedly stable storage is lost.

But how do we back up the database data? It is called a *dump*. It is basically taking a copy of all the data. If we are doing it live, we require that no transactions are running and the steps are [SKS11]:

1. Write all log records currently in main memory to stable storage
2. Write all buffer blocks to disk
3. Copy the contents of the database to stable storage
4. Note completion of the dump in the log

The form the dump takes could be raw data, but there are tools in most database packages that exports the data as a SQL dump: it writes out the data description language (create table statements, for example) and insert table statements for the data. Such a dump can be used to copy the database from one system to another or migrate it from one database server to another.

The approach described above does have the obvious downside of needing to stop execution for a very long time. That is, however, not realistic. A tool that dumps the database like `mysqldump` takes a complete database dump, but it does so as a transaction: it gets a snapshot of the data at the time the dump is requested; further changes can still go on in the meantime but this transaction will be left running for as long as it takes (and it can be HOURS!).

Another way that backups tend to get taken is at the level outside of the database: the operating system. The data is ultimately stored somewhere on a volume (or volumes) in the database and the operating system can do things like shadow copies and image backups of the drives.

ARIES

In spite of sounding like the name of a NASA program to Mars or some such, the ARIES algorithm is a database recovery routine used by IBM. It relies on three main pillars: write-ahead logging, repeating history during redo, and logging changes when undoing something. The description of how ARIES works is all from [EN11].

The algorithm and the approach we have examined for recovery works and it produces the correct results, but it comes with some drawbacks and performance decreasing elements. ARIES is hopefully going to address it. ARIES is broken up into three phases: analysis, redo, and undo.

Analysis. In the analysis phase, the goal is to figure out what pages in the buffer were dirty. We don't have the actual pages, of course, because the crash caused the loss of that data. However, e can determine the information based on the log data from the checkpoints: the dirty page table, transaction table, et cetera. During this phase a determination will also be made as to where to begin the redo phase.

Redo. In the redo phase, updates from the log are done in the database. The normal recovery routine expects that we only redo committed transactions, but in ARIES we will try hard to skip unnecessary work: and only redo operations that are necessary.

Undo. The last phase is the undo phase, and the log is scanned backwards and transactions that were active at the time of the crash are undone in reverse order.

Every log has a LSN – *Log Sequence Number*; this is an incrementing counter that indicates the log record's address on disk; each number corresponds to a specific change of some transaction. Each data page stores the LSN of the most recent log record that changed that page. Log records are generated for updating a page, committing a transaction, aborting a transaction, undoing an update, and ending a transaction. When an update is undone, a compensation log record is added; when a transaction ends, successfully or unsuccessfully, an end log record is written.

The log structure has numerous fields, including the previous LSN for that transaction, the transaction ID, and what type of transaction is being written. The LSN can link (in reverse order) the log records for the same transaction. Other fields in each log record might include where to find the item (page, offset, etc), as well as the before and after values.

To perform a recovery, we need the log as well as the *transaction table* and *dirty page table*, both of which are maintained by the transaction manager. The transaction table lists the various transactions in progress. Similarly, the dirty page table contains the pages that are dirty (shocking, I know). These are lost when a crash occurs but are rebuilt when the system comes back online, obviously by looking at the log data. Thos tell us about what information is in progress.

Remote Backups

Somewhat related to the idea of recovering our database if we have a crash is the idea of an online backup: a second copy of the database standing by to take over if the primary one should go down (whether through a crash or by losing network connection). The second copy is certainly on different hardware, and should more likely be located at a different physical location.

By moving it to a different physical location, of course, that adds some latency to the connection: it takes nontrivial time for data to be transferred from the primary to the backup location. And that means that a strategy for keeping these in sync is necessary, but there are other considerations:

Detecting failure. As you might imagine, the backup system needs to detect that the primary has failed so it knows when to take over. Ideally there are multiple independent communication lines between the primary and the backup so that the failure of any single one is not mis-detected as a failure of the primary system [SKS11].

Taking over. Suppose a real failure is detected and the primary has crashed. In the meantime, the backup does the job of the primary. If the first system comes back online, it could be the new backup or it could retake the role of primary. Either way, it needs to get caught up on the things that have happened in the meantime. Once both systems are back online, the redo logs can be sent to the system that is behind to get it caught up [SKS11].

Making extra sure. In this sort of configuration, is a transaction really committed until it has reached the backup site? It turns out we get a choice of how much we are willing to tolerate [SKS11]:

- **One-Safe**: If the transaction has been written to stable storage at the primary location. If there is a crash at the primary before that transaction has made it to the secondary location, then it may appear to be lost. When the primary site comes back, it may happen that in the meantime there has been an incompatible change made only on the secondary. Imagine you book seat 4A on the primary system and it crashes; someone else then books 4A on the secondary system. Who gets the seat? This is something that may require humans to sort out.
- **Two-Very-Safe**: A transaction is only committed once it is in both the primary and backup stable storage... which means that no transactions can be executed if one site is down.
- **Two-Safe**: A transaction is only committed once it is in both the primary and backup stable storage... if both are up; otherwise it is the same as one-safe: the primary suffices.

Either two-safe or two-very-safe necessarily comes with a performance penalty: when the transaction is committed the changes must be propagated to the other server (if up) to consider the transaction fully done.

References

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.