Lecture 27 — Recovery

Jeff Zarnett

Recovery

It's not a question of **if** things will go wrong, but a question of **when**. Bad things will happen and it matters how we deal with them. There are stories of tech companies who have a process specifically designed to cause random things to fail and crash so that the system is always designed to deal with failure at any time. That's extreme, but we need to be prepared. There will be failures: power failures, hardware failures, user failures...

In [SKS11] there are a few different types of failure that are worthy of our consideration:

- **Transaction Failure: Logical Error.** A transaction cannot continue because of an internal condition: invalid value, overflow, resource limit exceeded...

- **Transaction Failure: System Error.** Something has gone wrong and the system is in an undesirable state and the transaction cannot continue but can be restarted. A typical example here is deadlock.

- **System Crash.** A hardware malfunction, bug in the database or OS, that causes the loss of volatile storage contents and all transactions are halted; nonvolatile storage is not affected.

- **Disk Failure.** Allegedly-stable storage suffers a failure; copies from backups are needed to restore the data.

As is typical in a lot of programs, an error or bug brings the system down immediately, which is called "fail-stop" behaviour. This is normal behaviour in a C program, for example, but it may be even more important in the database. The fail-stop assumption is important because it prevents (or minimizes) corruption of the data. If we let the system carry on in an inconsistent state, we risk propagating invalid or corrupt data and infecting, so to speak, the rest of the data. Fail-stop prevents this risk, even if it is sometimes frustrating.

If something goes wrong, we want to "recover" from that failure. Recovery is this sense is, as you will recall, putting the database back into a consistent state. For that to work we need either to have some sort of log, or some sort of backup [EN11].

If things are really bad the recovery method is to restore from backups. We copy the data from the backup storage media and then restart the database server with that restored data set. Any data that has changed between the time of the backup and the time of the crash is quite simply lost. So it is important for backups to be current...

Supposing things are not so bad then we can use the log to try to recover; whether particular operations are undone or re-done to restore the consistent database. Looking at the log tells us what transactions were in progress, and based on what information is available, we can choose to carry out the changes or roll them back. For a transaction that was not complete, roll it back. For one that committed but the data changes have not yet completely been copied out to disk, the changes are carried out.

If we do *deferred updates* then the data on disk is not updated until after the commit point of the transaction. If that is the case, then the changes are noted down in the database log, and the log is written out to disk, and then and only then can we proceed. This strategy is, if you recall, visually the same as how things work in the Windows NTFS file system. If a transaction has failed in some way before the commit, no changes in persistent storage have taken place. It may be necessary to redo steps that are stored in the log if the failure takes place after the commit. This is sometimes called the "no undo / redo" algorithm [EN11].

The opposite approach is *immediate updates*; the data is updated before the commit but still the operations are recorded in persistent storage before they take place. If the failure occurs before we get to the commit point, then

we have to roll back and undo some changes of the transaction. It turns out that under this algorithm both undo and redo operations may be necessary so it is called the "undo / redo" algorithm. A variation that requires us to make all database changes happen in persistent storage means that we only have to undo things, leading to the "undo / no redo" algorithm [EN11].

It is also noted in [EN11] that we should remember that recovery can fail too. That means the undo and redo operations need to be *idempotent*: executing the operation multiple times has the same effect as executing it just once. If we need to restart the recovery process, we don't want inconsistent results. An operation like x = 7; is idempotent because that operation can be repeated arbitrarily many times and the value will remain 7. On the contrary, x++ is not idempotent because every execution changes the value of x. This is why, for example, we would want to record the operation in the database log as being something like an assignment statement, or even better, something that reads like "x is changed from 6 to 7".

Writing to Disk.

If we really want to be sure that a write has succeeded, we need to check; as Ronald Reagan said: "trust, but verify". A transfer may succeed completely, have a partial failure (some data in the block is changed but not all), or suffer a total failure (the destination is not changed at all) [SKS11]. If something goes wrong we need to actually detect it, of course, to know that something has gone wrong.

One way to be sure is the have two physical blocks for each of the database logical blocks (and even better if they are not on the same media). The process for a write is really to write the first block, check that it succeeded, write a copy to a second block. After the second write has happened, the write is finally considered completed.

If a failure takes place during the writing of either one of the blocks, they may not match. If they are the same then no error is detected and we don't have to do anything. If there is a mismatch then we can look at the checksum of each disk block (you do have those, right...?): the block with the invalid checksum is the one that is wrong and the other one will be treated as correct; the correct overwrites the incorrect. If both checksums are valid, the second block is copies over the first: if the write did not complete the full process and update both blocks, we consider it failed and roll it back [SKS11].

Caching (Buffering) of Disk Blocks

As we have already examined, there are often disk blocks in memory being cached, and the recovery process's success or failure may depend on how the data in the cache is handled. Because the cache data is in volatile memory, a crash or loss of power causes it to wink out of existence. Normally, management of paging memory out to disk and back into RAM is the purview of the operating system, and probably you learned about that extensively in an examination of the same. A typical program doesn't need to manage this aspect of memory, but a database server does.

Conveniently, operating systems make some low-level system calls available to allow this to happen. We'll call those operations on a block B input(B) for reading a block into memory, and output(B) for writing a block to disk (replacing what was on disk already in that spot).
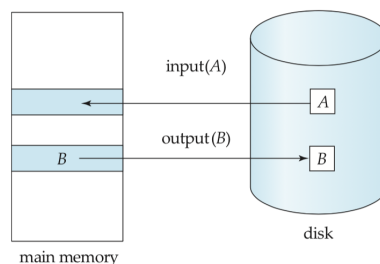
Block storage operations. [SKS11]

In addition to this, the database has a small directory of what items are in the buffers (which resembles a page table in operating systems, actually) [EN11]. If some item is needed for a read or write, the directory is checked to find out if what we need is already in cache; if it is not then it must be retrieved from disk to be used. It might also be necessary to flush some of the data out to disk if there is no more space and we must replace a block.

That sort of "normal" writing out to disk has some restrictions. Previous discussions of caching have covered the various algorithms and keeping track of whether the block was modified, but there exists an additional restriction. This is whether or not the block is *pinned* – that is, forced to remain in memory and not permitted to be written to disk. This is often implemented as just a single bit in the block.

A very common example of why a page might be pinned by the recovery system is because it has been updated by a transaction that has not been committed [EN11]. If we wait for the transaction to commit, the block in question won't be written out to disk in a partially-updated state.

Writing a block to the output via the `output` command itself might not result in immediately writing a particular block to disk, because that block still might be needed for other operations. That can cause a bit of a headache because a crash before the output operation takes places means the data is lost... The recovery system can choose, however, to insist a block be written to disk immediately, and that is called a *force output* if a write command is explicitly issued [SKS11].

Ensuring Atomicity

What we really want to avoid is that a transaction is incompletely recorded in the non-volatile storage, violating the atomicity of the transaction. The example we'll use (from [SKS11]) is the standard banking sort of example: we want to transfer $50 from account A (initial balance, $1000) to account B (initial balance, $2000). If the system crashed during the execution, after the write of the block containing A has taken place but before the write of the block containing B has, things get interesting. Volatile memory is lost in the crash so we don't know the fate of the transaction.

Under these circumstances, looking at the data usually does not tell us enough. If A shows a balance of $950 and B a balance of $2000, we know (from the setup of the example) that this is incorrect. The database system, however can't tell if those values are correct or incorrect without some additional information (just as we can't either). And as we have already discussed, we can solve this by writing the information to the log before performing the actual write.

More formally, the log is a tuple of 4 elements: T_i, the transaction in question; X_j, the data element to be modified; V_1 the old value; and V_2 the new value. Other log records would indicate when a transaction begins, when it commits, or when it aborts.

To prevent the log size from getting out of hand, we will need to eventually delete some old history. We could take the NTFS-like approach and delete transactions from the log once they are well and truly finished, but that, as we will soon see, is not actually what we do.

These give us enough information to perform the undo or redo operations as described above. In the case of the bank transfer example, we can see that the step to change the balance of A from $1000 to $950 has succeeded but the step to change the balance of B from $2000 to $2050 has not. There are two valid paths for proceeding: undo the change to A OR redo the change to B. Both produce consistent results. You might imagine that it is always preferable to redo the operation, but that can really only happen if the transaction has already committed... otherwise we don't know what else the transaction was going to do.

As you might imagine, if there are multiple things to be undone or redone, the order matters to produce the correct result. If you don't believe that order matters, next time you bake a cake, first bake all the ingredients, then mix them. The redo approach then, requires us to scan the log from top to bottom, performing appropriate redo actions where necessary on each log record [SKS11]. This is efficient and sensible, preserving order and reading the log only once, sequentially.

The undo operation works a little differently: it restores data items to their old values, but also writes a log noting

what it has done. The log records created are noted as "redo-only" because they don't contain the old (now-erased) value of the log record [SKS11]. That looks like it's very strange, but we'll see shortly why this is so. And when all undo operations for a given transaction are completed, an abort log record is put into the log to show that the undo is finished. This guarantees that every transaction will have either a commit or abort record in the log, even those that were unfinished at the time of the crash.

With these rules in mind, given a log, how do we determine what transactions need to be undone or redone? Obviously, if a transaction does not have either a commit or abort log message it was unfinished and will be aborted. If a transaction has a commit in the log, it needs to be redone. But if a transaction has an abort in the log it needs to be redone as well. Wait, what? When a transaction is undone, as above, we need to undo the changes with those redo-only log entries to make sure everything is put back where it should be.

Let's look at an example:

$<T_0$ start$>$	$<T_0$ start$>$	$<T_0$ start$>$
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit$>$	$<T_0$ commit$>$
	$<T_1$ start$>$	$<T_1$ start$>$
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit$>$
(a)	(b)	(c)

A transaction log at three different points [SKS11].

If the system crashes at point (a) in the above diagram, transaction T_0 is started but has neither committed nor aborted. We need to therefore, undo the operations it has done. So the following log entries will be created: a log entry setting B back to 2000, an entry setting A back to 1000, and a record noting that T_0 is aborted.

If the crash takes place at point (b), then transaction T_0 is complete but T_1 is not. Right away we can conclude that we need to abort T_1 and roll it back as above. So C is set back to 700. As for T_0, we will need to redo the transaction and we will see A at 950 and B at 2050 in the end.

Should the crash take place at point (c) in the diagram, then both T_0 and T_1 are committed and need to be redone. The final values: A is 950, B is 2050, C is 600.

Remember that if a transaction aborts, the transaction is "redone" in the sense that redo-only statements are carried out to cancel out anything that was done before it. So the transaction might write a value to X only to have it reversed by the subsequent operation before the abort.

Those who do not forget history are doomed to repeat it. [1] As just mentioned, the log structure we have means we write a lot into the log but never take anything out. And in theory to recover from a crash we have to scan the entire log and redo or undo anything needing to be redone or undone. This can be, however, a very long list. Repeating the redo operations many times does no harm, but it is a huge waste of time.

What we'd like is to remove from the log any transactions that are well and truly done and we're sure a redo operation is totally redundant. We can remove them one at a time as the very last step of actually recording a transaction as done, but that can require a lot of searching the log to find which one is the right one to remove. Instead, we can try to do this en masse through *checkpoints*.

A checkpoint is a particular place in the log where we "save our progress", so to speak, so that if there is a crash we need only resume from the last checkpoint rather than starting at the beginning of the log. You are very likely already familiar with checkpoints, because they have been something you encountered in another context. Suppose everyone's favourite Italian plumber plunges to an untimely death: you can resume from some partial

---

[1]With apologies to George Santayana.

progress, such as the beginning of the level, or even some halfway point within the level, rather than having to start the quest to save the princess from the very beginning.

The steps to perform a checkpoint are [SKS11]:

1. Write all logs currently in volatile storage to stable storage.

2. Write all all modified buffer block to stable storage.

3. Place a checkpoint record in the log.

Transactions are not allowed to make changes to buffers or logs during the time a checkpoint is being created (although we might be able to relax this rule).

The checkpoint record notes that it is a checkpoint and has a list L of all transactions that are active at the time of the checkpoint. The checkpoint means that any transaction before that point in the log is completed and for sure put out to stable storage, so those transactions do not need to be repeated. Excellent.

If a crash occurs, we just look for the last checkpoint in the log (the one closest to the end of the file) and then the only redo and undo operations that need to be performed are those on the transactions in the list L and any that started after the checkpoint. From there the routine works as we have previously described: for each transaction, redo or undo depending on the state of the transaction.

The checkpoint also allows us to get rid of some records once they become sufficiently old: anything that is before the first transaction in L of a checkpoint is "ancient history" and can be removed from the log.

The Recovery Algorithm

The hall is rented and the orchestra engaged: now it's time to see if we can dance[2] Now it is time to actually execute the algorithm!

Recovery takes place in two phases, first the redo phase and then the undo phase (really). We start at the last checkpoint in the log and scan our way forward, then reverse direction and go back. The algorithm is described briefly in [SKS11]:

The *redo phase* requires replaying updates of all transactions, scanning forward in the log from the last checkpoint. Even the ones that were rolled back (as we have seen) and including those transactions that were incomplete. We still do those operations, but we will note for later the incomplete transactions.

More formally:

1. The list of transactions to be rolled back is set to the list L of the last checkpoint record in the log.

2. Each log operation is then carried out, both the "regular" and redo-only operations.

3. If we encounter the start of a transaction marker for a new transaction, add that transaction to the list of transactions to be rolled back.

4. If we reach an abort or commit statement for a transaction, remove that transaction from the list of transactions to be rolled back.

Following this phase of the process, we have identified all transactions that were incomplete (neither committed nor aborted) at the time of the crash.

Then we have the *undo phase*: roll back all the transactions in the list we have just generated in the redo phase. We scan the log backwards from the end moving up.

---

[2]Watch the episode of Star Trek: The Next Generation called "Q Who".

1. Whenever we find a record in the undo list, we perform the usual undo actions as if it was rollback of a failed transaction.

2. When we reach the start log entry of a transaction we know we have undone all actions of that transaction, so we can put an abort record at the end of the log, and remove this transaction from the list.

3. The undo phase terminates when the undo-list is empty (i.e., all incomplete transactions are properly undone).

Following the undo phase, the database is in a consistent state again and new transactions can be permitted to begin... in other words, the system may begin normal operations. The database is once again in a consistent state and all transactions have the atomic nature they promise: they have either completed successfully or it is as if they haven't happened at all. Some operations were cancelled and potentially permanently lost, but the database is in a consistent state and that is the goal of the recovery process.

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.