ECE356: Database Systems Winter 2018

Lecture 22 — Transactions

Jeff Zarnett *2022-10-14*

Transactions

Transactions are fundamental to database operations. A transaction is a grouping of operations that belong together and should be treated as an indivisible unit. You will recall from a great deal of discussion on the subject of concurrency that bad things can happen when an intermediate state of a multiple-step operation becomes inadvertently visible. Most of the examples we covered in earlier courses looked at things like `x++;` being a read, addition, and a write and how concurrent accesses to `x` could result in the wrong value being read or written. The solution we used then was mutual exclusion: forcing other accesses to `x` to wait while some operation was in progress. The transaction is the database way of dealing with the need to prevent intermediate state from becoming visible (although mutual exclusion can be used for a few purposes as we'll end up seeing later on).

By this stage you have almost certainly done something involving transactions for code management. Version control systems such as git and subversion use transactions. If you are making a large commit in subversion and the wireless network drops out, the commit is cancelled and from the view of anyone else using that repository, your half of a commit never happened. That's a big improvement over the cvs (concurrent versioning system approach) where the half finished commit appears in the repository, breaking the build. If your commit is successfully completed then the repository is updated and all is well. This seemingly-simple use case really captures all the most important attributes of transactions, as we will see. Understanding transactions as a concept is easy... implementing their behaviour is harder...

You may also recall from an operating systems class that modern file systems like NTFS or HFS+ are journaled. This is also a form of transaction management. A write to disk is treated as a transaction in that the updates to the file system metadata takes place inside a transaction meaning the metadata can never be left in an inconsistent state. For various reasons the actual writes of user data may not get that same treatment.

To execute a transaction we need first to create one. A transaction has some sort of begin transaction statement at the beginning, then the operations to take place in the transaction, and finally an end transaction statement. Execution looks something like writing down the transaction into a log, doing the operations in the transaction, and when the last one is complete, if all went well, marking the transaction as successful.

A transaction has a state and the actual implementation may vary from system to system but we will consider the following:

- **Not Started**: Transaction has been created but it has not started running yet (when we talk about concurrency control we'll see why one might be in this state for a while...)
- **Running**: Transaction has started running.
- **Ready to Commit**: Also sometimes called partially committed; all operations are finished but it is not yet committed and permanently recorded.
- **Committed**: Transaction has finished and the effects have been recorded in the database permanently.
- **Failed**: A problem has been encountered in the execution of the transaction and it needs to be rolled back.
- **Aborted**: Transaction has finished unsuccessfully and any changes made so far have been undone.

How might a transaction be unsuccessful? We have already examined one situation in the context of subversion, that is an operation in progress gets interrupted by a network disconnection on the client side. But there are

many more possibilities; the server could crash, or, more interestingly an error might be encountered in execution of the query. A division by zero or foreign key constraint violation will move the transaction to the aborted state immediately and halt all further execution of that transaction, rather like encountering an exception in a language like Java. These exceptions, however, tend to be handled by undoing any partly-done changes.

Transactions should have the *ACID* properties [EN11]:

- **Atomicity**: A transaction is atomic (indivisible) – it either is completed in its entirety or it is as if it never happened at all.
- **Consistency preservation**: If the transaction is completed successfully, it moves the database from one consistent state to another consistent state (i.e., does not break any key constraints etc!)
- **Isolation**: A transaction should appear as if it is the only transaction executing right now even though it's possible many transactions are executing concurrently.
- **Durability**: A transaction's changes should be persistent in the database and not be lost.

The good news is that we have already seen a number of the important concepts for achieving the ACID properties in previous courses. We have already learned a bit about how to ensure atomicity of larger operations as well as a little about durability.

Obligatory, Mandatory, Non-Optional Bank Analogy

It was inevitable that a bank analogy would be applied to transactions. There's just kind of no way around it. All the textbooks do it. And why not, it works well enough for our purposes. The example from [SKS11] revolves around transferring $50 from account A to account B. This is a common enough operation: your friend buys tickets to that Metallica Concert[1] you want to go to, and you send an electronic bank transfer to said friend in the amount of your half. Here's a situation where you need both parts to happen for everyone to be happy. Your friend will not be happy if the money is deducted from your account but never arrives. The bank will not be happy if the money appears in your friend's account but was never deducted from yours. We need a transaction.

If we wanted to write out the transaction in some sort of pseudocode, ignoring for the moment that we need to turn this into database statements, the steps are:

```
BEGIN TRANSACTION
read A.balance
A.balance = A.balance - 50
write A.balance
read B.balance
B.balance = B.balance + 50
write B.balance
END TRANSACTION
```

Those who are familiar with banks will note that the bank, of course, takes the money out of your account first. Those of you familiar with my lectures will note that it sure looks like I hate banks.

Let's think about the ACID principles as they apply here.

[1] Somehow I doubt their tickets are as cheap as $50 but let's pretend.

Atomicity. If we do not have atomicity in this transaction then it is possible that some halfway completed state is revealed, such as after the deduction has occurred from account A but before the increase in account B. That would be inconsistent, and we do not want that state to be visible (see: isolation). Moreover, if we have a crash at this stage, the operation is only partway complete and that is undesirable. But we already wrote A.balance (and let's just say for now that write goes out to disk).

The way we prevent this problem is the same way that NTFS does it (if you remember that part of the operating systems course): there is a log file. The transaction steps are written in the log file before they are executed. The transaction is treated like a checklist: after each operation is done we check off the box next to the operation and then we may proceed to the next. If there is a crash and we have an inconsistent state, we can use the log to go backwards and undo the partially complete operation. And, as something new for database transactions, we might even be able to pick up where we left off and finish it. Either way, we can't leave it half done: forward or back, we have to be at one consistent state or another.

Consistency. There aren't any sort of foreign key or unique key rules here but there does exist a consistency rule: the sum of account A and account B must remain the same after the transaction is done. If that is not true, then something is wrong and money has gone missing or been created out of thin air. Destroying and creating money are the job of the central banks and they get very upset when you try to do it yourself at home.

There could exist other consistency rules in this system that come in to play. For example, the balance may be required to be non-negative. So if you attempted to transfer $50 to someone from an account that contains only $31.25 then the transaction will not succeed because the consistency check that says balance must be greater than or equal to zero will be violated. Banks do not necessarily follow this rule in real life because they have the ability to overdraft (i.e., temporarily allow your account to go into the negative) but overdraft has a limit as well, so if you are allowed overdraft of $50 then your account may have a balance of -$50.00 but not lower and the same sort of rejection of the update would occur if that constraint is violated.

Isolation. As we know, when the same data is being accessed by multiple threads in a C program, there is the possibility that an error will result. An unlucky interleaving of statements could happen here as well. In a simple model, we can just solve the problem by doing transactions one at a time. This has adverse impacts on performance but guarantees the isolation property. Some other strategies that we have previously learned, such as mutual exclusion, locking will come into play here.

Isolation is a very complex subject that will be the subject of the next lecture. On top of that, we will need to also dig into the topic of concurrency control in the database which is, again, an upcoming topic.

Durability. Finally, once the transaction has reached the end statement and it is ready to commit, that change should be permanently reflected in the data on disk. For a transaction to be truly durable it means that if the system crashes at this point, the transaction's changes will not be lost. In short, either (1) the data has already been written to disk or (2) the information already written to disk is sufficient that we could redo the changes if the system is restarted after a failure [SKS11].

Digression: Stable Storage

For a transaction to be really considered permanent, it might not actually be enough just to write it out to disk. Disk is not volatile storage and data on disk will be preserved if the power goes out or the server process crashes or something, but that does not account for another mode of failure: hard drive death! Hard drives can and do die at the worst possible times. Far, far too many people go around these days acting as if their drives will never die and the single copy of important things (pictures, documents, whatever) will always be accessible. If you do not believe that hard drives die, I will be glad to show you one that died on me in April of 2015. With no warning.

Ultimately I wasn't very upset about the loss of that drive even though it contained installed programs, my music collection, et cetera. Why not? I had backups of everything that could not be obtained again easily (e.g., just download the game again from Steam).

In [SKS11] a further level of storage beyond non-volatile is called *stable storage* which they define as where data can never get lost. Never is somewhat exaggerated: no matter how many forms of storage you have, a nuclear war will probably get them all (or at least all of us), and if nothing else, the heat death of the universe will certainly finish them off. Physics can actually be kind of depressing. Maybe we are digressing. The point of stable storage, though, is to be sure that even the failure of nonvolatile storage is not sufficient for data to get lost, and a transaction isn't really truly durable until it has been written into the stable storage.

In real life where we don't have infinite money and plan for more reasonable scenarios than nuclear armageddon or the rise of Skynet[2] there are a few common techniques for creating (an approximation of) stable storage.

The first is the RAID array (redundant array of independent disks) so the failure of any individual disk does not result in data loss. RAID has several levels and some of them offer no redundancy at all, others offer a lot more. Choose wisely. I will also note, even though it is a bit of a digression, that it's a super bad idea to buy all your RAID disks at the same time from the same manufacturer. They likely came from the same lot and will tend to live about equally long. So if you lose a drive, you think this is recoverable, only to discover that the strain of rebuilding the data from the other drives in the RAID array has killed another of the drives and now your data is lost. Not good.

The next strategy is backups: having another copy of the data on an independent storage medium. That is not to say that the original and the backup cannot be both hard drives, just that they need to be two separate hard drives. If you don't have backups and a drive has died you can send it out to a data recovery service and they are not cheap and cannot guarantee success. Really. Buying a backup drive today is like a tenth of the price of sending your dead drive out for recovery.

For best results, the data you consider important should be backed up and there should also be off-site backups. There are stories of companies where they were good about taking backups but the backups lived in the same server room as the original servers and the original and backup copies were both lost in a server room fire.

For the strictest possible definition, a transaction is really only durable when it is not only written to stable storage, but the backup copies of the database have been updated as well to reflect the change. It doesn't get any more permanent than that.

Atomicity and Durability

We can also take some time to take a closer look at atomicity. We know the key idea and how it should work, but have only gone over briefly how it actually takes place. We will examine two approaches.

The log file is the most important key to the usual approach. The more detail is written into the log file, the more options we have for dealing with a failure of the transaction if it occurs. The trade-off here is of course the size of the log and the amount of time spent maintaining that log. A sample log entry from [SKS11], represented here as a table structure:

Transaction ID	Relation	ID	Attribute	Old Value	New Value
385	account	8675309	balance	5000.00	4950.00

The table headers probably don't appear in the log and it may be binary data rather than especially human-readable. Nevertheless, the idea is hopefully clear. By showing the old and new data we can easily check if the data was updated before the crash and whether we need to set it back to the old value if the transaction did not succeed.

Another way that we can get atomicity is the ZFS (A file system developed by Sun Microsystems) approach. In this case when a transaction is to be executed, a copy of the data is made. The copy is modified, and if the transaction runs to completion, then the copy replaces the original data.

[2] If a very large-muscled guy with an Austrian accent ever says to you "Come with me if you want to live.", you go.

That is a neat approach but comes with some drawbacks, such as the fact that in ZFS if the file system gets full you can never delete anything to make more space; or in other words you do need a significant amount of extra space, and accordingly, a lot more reads and writes to accomplish the same thing. The upside is that if the transaction is to be aborted, we just throw away the copy and that is very efficient; nothing needs to be undone. Optimizing for the failure case, however, maybe seems overly pessimistic. We would expect that most database operations succeed rather than fail the log approach seems a bit better if we think that will be the case.

Getting to the state "committed" actually means that the transaction is not only completed but written out to disk in such a way that if the system crashes now that transaction will not get lost. That is an important rule to hold to because without it we could check off an item as done when it is not actually done.

If the transaction succeeds and is committed then it can be removed from the log. This keeps the log file size reasonable. In many cases the log contains only the in-progress transactions (i.e., the committed transactions are removed right away) so a quick glance at the log file will tell the database server on boot up whether any transactions were in progress at the time of crash.

Once a transaction is committed, we cannot cancel or roll back its effects. In the words of Lady MacBeth: "What's done is done and cannot be undone." You can return the system to an equivalent state to the before by executing a *compensating transaction* [SKS11]. If the bank makes a mistake and withdraws $50 from your account, and this needs to be corrected, it can't be corrected by annulling the initial transfer. Instead, the compensating transaction would be the opposite: deposit $50 into your account to put you back into the state you were in before. Sometimes, however, there is no going back! Compensating transactions are not always possible when information is tossed away such as a deletion or truncate statement and are generally left to the user to fix. You took backups, right? You can restore data out of those...

Suppose that a transaction does not succeed. There are now two ways forward from the aborted state. The first is to re-try the transaction. Sometimes the transaction failed due to temporary circumstances, network disruption being one such example. As we will see, transactions sometimes interfere with each other a bit and that can cause one of them to need to start over again. The transaction, having been aborted, is assigned a new number and it tries again. It is worth noting that there is likely a need for a limit on how many times a transaction may be re-started this way, otherwise an unsuccessful transaction may be doomed to constantly retry and fail for all eternity, wasting database resources.

The alternative is to kill the transaction, which is really just the same as giving up. Giving up is mandatory if the operation in question failed due to some unrecoverable error such as trying to violate a constraint or any other reason the transaction can never succeed. An error of some sort will be returned the requesting program/client to indicate the failure. Once that happens it is out of the database server's hands and purview; the user may try again or the application program may have its own error handling routines to deal with this.

Writing to console and network. Another concern raised in [SKS11] has to do with external writes, i.e. making results available to users and other devices or individuals outside the database system. Once data has gone out over the network or has been printed to the console there is no way to get it back. The fully correct rule to handle this would be to allow no such output until the transaction is committed; we could write data to a temporary location until such time that is true.

Sometimes the state of an external write of some sort is fairly indeterminate: what happens if we aren't sure whether the data was sent to the printer? Do we try printing again? Probably not, the user will either print again manually if necessary or just give up and use some other printer if it doesn't work.

Sometimes, sadly, we actually want to show the status of a transaction to a user. Users love progress bars. For good reason. An operation that will take a long time may seem concerning to a user, who fears that it is stuck. For this reason and some others we might need to allow long running transactions to be interactive. That comes with its own headaches...

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.