Transactions Lecture

What is the transaction? You may be familiar with the idea of a transaction in terms of buying something. The store clerk hands you something of value and in exchange you give him or her some money. We call that a transaction. In the database world, it's different. There is no money that is exchanged and there is no transfer of goods. But, one thing that is the same is that it is a set of operations that is self-contained. That is, it's a bunch of work that needs to get done and we can package it up neatly.

Let's go to the first slide, Transaction Handling. In the database world, the transaction is simply a bunch of work that is done together--a group of operations. It is very much an all or nothing proposition. If we designate five different statements as a transaction, then either all five of those statements will be executed or none of them will be executed. You may recall from your Intro to Database class the acronym, ACID. ACID stands for atomicity, consistency, isolation, and durability. This exact definition and what each of these terms means is very important to database engine designers. Fortunately, that's not us. All we need to know, and all I would ask you about on test, is that if the database guarantees ACID compliance, then we are guaranteed that any group of statements we declare to be in a transaction will either be executed together or not executed at all, and data integrity will be maintained regardless of that outcome.

So, you may be wondering why or when I would use transactions; I will talk about that in just a couple of minutes. But what you need to understand right now is that this notion of a transaction is just a way to group a bunch of operations. Now, because of the way transactions are implemented in most databases, they can introduce some problems. After all, we all know there is no such thing as a free lunch. On the slide I list contention issues and stale data problems. We'll see more about this later.

Let's move to the next slide on when to use a transaction. There are two main reasons why I would use a transaction. One is if I have to do some updates on multiple tables and I want to be sure that either all of them happen or none of them happen. Now, if I can manage that by joining the tables together and executing a single SQL statement, then that is certainly the preferred way to go. But oftentimes you will not be able to do that--the tables are related to each other in a logical sense, but we do not have foreign keys to physically connect them. This is an example of when I would want to use a transaction.

There's a classic example of this which I'll describe to you here. Imagine a banking application. Let's also imagine that in this application there are two tables: one table represents your savings account and contains all your deposits and withdrawals; the other table represent your checking account and contains all your deposits and withdrawals from that account. (As an aside, this would be really dumb way to design a banking application, but just go with it the purpose of this explanation.) So now, let's say you want to move $100 from your savings account into your checking account. We would first execute an SQL statement to insert a record into the **checking** account saying withdraw $100, and then we would execute a second SQL statement to insert a record into the **savings** account saying deposit $100. That would have the

effect of transferring $100 from your savings account to your checking account. But now, what if something happened in between those two statements? What if someone tripped over the power cord and pulled the plug? You would have withdrawn $100 from you savings account, but not been able to deposit the money into the checking account. The money would have vanished. However, if we had wrapped those two SQL statements in a transaction, the money would not be lost. We would have been guaranteed that either both statements would be executed or neither one of the statements would be executed.

A reasonable mental model for what's going on when you do a transaction is to think as if the database were making a copy of itself. Imagine that when you say to start a transaction, the database copies itself and then everything you're doing happens on that version. If we get to the end and you say that the transaction is complete, then everything on that version is copied back into the original database. But if something happens in between the statements before you're done, then there was no chance to copy the information back. Your work is lost, but the original database remains intact and the data integrity is preserved.

So that's one time when we would want to use a transaction, **when we are performing multiple SQL statements on the same or different tables and we want to be sure that they get executed as a group.** Another situation is when I need to get some information from the database, do some calculations based on that information, and then update the database with my results. In this situation, I want to be sure that no one has changed the information I'm basing my calculations on while I am doing them. Otherwise, the results I insert back into the database would be inconsistent with the underlying data.

I want to repeat that if these operations can be done with a JOIN, then that is the preferred way to go. There is significant overhead in doing a transaction so we do not want to do it needlessly. However, it is a very powerful tool and one that you will need to use frequently.

Let's go to the next slide regarding how to use a transaction. So, here are the general steps to using a transaction. Depending on the database, the syntax will vary as will the need to explicitly do some these operations. But, generally speaking, this is our process. After connecting and opening the database, we execute a command to start the transaction. We then do all the work we want to do and then either commit or roll back the transaction. Commit means I'm happy and the database management system should go ahead and record all the changes. Rollback means, "Oops, I've changed my mind I don't want to save this information." After we've done that, depending on the database, we may need to explicitly close the transaction.

That's it. I'll get into some of the details, but first let's do a little demonstration. I'm going to tell you exactly what you need to do and I want you to follow along on your computer. To do this, you need to have MySQL and the travel database installed on your machine.

Step 1: Open two different command windows or terminals, depending on your operating system, and start up MySQL in each of them. Each window is going to represent a different user accessing our database.

Step 2: In each window, use the Travel database by executing the "USE travel;" command

Step 3: In each window, execute the following SQL command: "SELECT * FROM equipment WHERE equipId = 568;" and compare the results. You should see the same information.

Step 4: In the **left** window, execute the following SQL command: "UPDATE Equipment SET equipmentName = 'Test' WHERE equipId = 568;"

Step 5: In each of the windows, execute the SELECT command from Step 3 once again and compare the results. You should see that the name changed in both the windows.

Step 6: In the **left** window, execute the following SQL command: "START TRANSACTION;" Then, in the same window, execute this SQL command: "UPDATE Equipment SET equipmentName = 'Continental' WHERE equipId = 568;"

Step 7: In each of the windows, execute the SELECT statement from Step 3 once again and compare the results. You should see a discrepancy. In the **left** window, the one where you were doing the transaction, the name of the equipment should now be "Continental". But in the **right** window, the name is still "Test". This is because we have started a transaction in that first window and are now effectively working on a copy of the database.

Step 8: In the **left** window where you are running the transaction, execute the SQL command: "Commit;"

Step 9: In each of the windows, once again run the SELECT statement from Step 3. You should see that the name of the equipment is now "Continental" in both of the windows. This is because when we committed the transaction, that version of the database was copied over to the original

Let's try another demo.

Step 1: In the **left** window, start a transaction and execute an update statement on the equipment table for equipId 568. You can repeat the same the same UPDATE statement we've been using, just pick yet another name for that equipId.

Step 2: In the **right** window, try to execute the following Update statement: "UPDATE Equipment SET equipmentCapacity = 200;" You will notice that after you hit enter, nothing happens.

Here's why. When you executed the UPDATE in the transaction, it put what we call a "write lock" on the equipment table. It saw that in your copy of the database, you were updating that record and it didn't want anyone else to be pulling the rug out from under you by changing the same record. So, it locked it from allowing anyone to be able to write to the table, They could still read, we saw that before, but they could not update it.

If we let this sit there for a little while, you'll see a message come back saying that the lock wait timeout was exceeded. That means, it was patiently waiting for the transaction to end and the lock to be released, but it gave up. Wait a few moments for this to happen.

Step 3: Let's try again. In the **right** window, once again try to execute the following Update statement: "UPDATE Equipment SET equipmentCapacity = 200;" Again, you will notice that after you hit enter, nothing happens, but now move on to the next step.

Step 4: In the **left** window, commit the transaction by executing the command: "COMMIT;" As soon as you do that, you will see the UPDATE statement in the **right** window get executed. When we committed the transaction in the left window, the lock was released and the statement could execute.

This is a good demonstration that when we do use transactions, we need to keep them short and sweet. If we lock a table for too long, it will cause problems for other users. It is also a good lesson that when you are executing a perfectly good SQL statement, you may still get an exception because some other user, or part of the application, has a lock on the table. Always code defensively.

Okay, that's it for the demo. But, you should play with this and get comfortable with the behavior. Let's move on to the next slide on autocommit. Most database management systems have the concept of autocommit. This means it will automatically commit every single SQL statement as if it is its own transaction. So, when you sit down at the console in MySQL, every individual SQL command that you enter is treated like its own little transaction and is immediately committed. However, there is a system-level configuration parameter that you can set and change this behavior. You can turn autocommit off. That means if you were at the console typing and executing SQL commands, none of them would be committed until you explicitly say commit. What makes this a little challenging is that every database management system is a little different. In MySQL, autocommit is on by default. So, unless you go in and change the value for that configuration parameter, you'll get the behavior you are accustomed to with every SQL statement being committed as you executed. A DBA, when installing MySQL, could set autocommit to be off—you won't know unless you test it. (This rarely happens.) On the other hand, Oracle is the exact opposite. Its default is for autocommit to be on, so none of your SQL statements will be committed until you end the console session. Most of your work as a developer will not be at the console, but rather in your code. So why do I tell you this? Let's look at the next slide on executing transactions.

In your code, there are two ways we can execute transaction. One way is to make-believe we are sitting at the console, what I refer to here as the manual approach. The other way is to create a transaction object in our code and work with our commands that way. Let's go to the next slide and take closer look at the manual approach.

Recall that when we were in our code and executing SQL statements, there were two different methods we could use. One of them was the query method that returned a result set. The other was the execute method that returned an integer indicating success. We can use that execute method for our transaction. The SQL statement that we would pass into the execute method is going to be exactly the same as what you type in the console, "START TRANSACTION". We can then do our other SQL statements and then finish up by using the execute method and passing in the "COMMIT" statement.

Let's go to the next slide. This is another way I can use the manual approach. Here, I am using my connection object to change the autocommit property. The connection object has a method that allows me to set the autocommit property. By setting it to false as I show here on the slide, I am telling it to stack up all the subequent SQL commands until I commit them. I can then do a commit by calling the commit() method on my connection object. Let's go to the next slide to see how we do this with objects instead of mimicking the console operations.

With the object approach, we simply create a transaction object, specify statements to be executed, execute them, and then commit or roll back the work. That's the general approach. Of course, the specifics differ with every language. Let's take a look at a couple of languages. The next slide shows how to do this with C#.

As before we've seen earlier in the course, we start by creating a new command object and specifying the database connection to use. There's nothing new there, but pay careful attention to the next two lines, these are important. First, we create the transaction object using the BeginTransaction() method of our connection object. Next, we need to connect our transaction and our command objects. We do this by setting the transaction property of the command object. Look at this carefully. At the center of all this is the command object. It has two properties that we're setting, one is what database connection I should use and the other is what transaction this is part of.

The next part is quite easy. Using our command object, we set the SQL statement and run it. We do that is many times as we want with as many SQL statements as we want. When we're done with all our SQL, we call the Commit() method of our transaction object. That's C#, let's look at Java on the next slide.

Java does not have a transaction object. So, we have to use the manual method. First, we set autoCommit to false. Then we do all the SQL commands as we would normally do them. And then, when we're done, we invoke the commit() method or the rollback() method. And it's important to be a good citizen and clean up after ourselves, so we set autoCommit back to true.

Thus far, we have committed all of our transactions. I keep mentioning this roll back idea. Let's go to the next slide to see more about that. There are two reason why I might want to do a rollback. The first one is if in a lengthy set of statements I discovered some condition that meant I didn't want to continue, then I would want to undo or roll back everything done so far. If we go back to the bank account example, I might first want to check whether you actually had $100 in your savings account before trying to transfer it. So, I would start a transaction, query the balance, and then decide whether to roll back or continue.

The other situation would be in attempt to cleanup when an exception was caught. Depending on the nature of the exception and where in your code it occurred, this may or may not be necessary, but it is certainly something you should think about. The syntax for the rollback is very simple as you see here on the slide.

There is another concept that is not on the slides, but you may encounter on the job. It is the notion of a SAVEPOINT. What if you had a particularly complex transaction consisting of several somewhat independent steps. You might have a situation where even if you couldn't continue with, for example, step 5, steps 1 -3 were still good. In that case, you would set a SAVEPOINT after step 3 and if you executed a ROLLBACK at step 5, the work of the first 3 steps would still be saved. MySQL does not support this, Oracle does.

As I said earlier, there is significant overhead in doing a transaction so let's look at the next slide and go over a few things. The first one is pretty easy to think about: don't close the database in the middle of the transaction, nothing good can come from that. And your reaction may be, well duh, of course not. But this is when we get into that issue that I mentioned on the prior slide of catching exceptions. Perhaps an exception was thrown in the middle of a transaction and in the catch block you're trying to be a good citizen and close the database. Well, what if you forgot to first end the transaction? I can't tell you exactly what would happen because this behavior is undefined. That means, even the folks who designed the DBMS aren't sure what would happen. Don't do it.

Some other things that will cause exceptions are trying to do a second commit when you have already done the first one or a commit after a rollback. Number four is important also, you want to have a nice, clean, straightforward code. Not too different is number five which points out that you should be explicitly committing a transaction and not letting it be committed by default when you change the autoCommit property. And finally, a general catchall statement, write your code with purpose. The next slide has one more caution and it is important. Make sure that everything the transaction depends on, happens within the transaction. In a heavily used multi-user database there are lots of updates, in fractions of a second. Doing a select before you start a transaction leaves open the possibility that someone will change that data in between your SELECT and your BEGIN TRANSACTION.

The next, and last, slide shows you a situation and asks you where you should begin and end your transaction. Where do you think it should go? Give it some thought and then post your

thoughts in the discussion area for transactions. Something like this would be a good test question.

So that's it for this lecture. This approach is new to all of us so I am open to any feedback you have on how to make the situation better. I wrote this pretty much as I would say it. If that works for you, good. If you'd prefer a more formal, textbook like feel, let me know. Feel free to post something in the general area of our discussions or shoot me an email. Thanks.