MOUNT ROYAL
UNIVERSITY
1910

MOUNT ROYAL UNIVERSITY

Department of
MATHEMATICS AND COMPUTING

COMP 4522

Database II: Advanced Databases

Contents

EXERCISE

SIX

TRANSACTIONS

The ACID properties of transactions
Locking

WE STARTED WITH THE DEFINITION of a valid relations. When we looked at updating those relations we realized that things were a bit more complex and that to be able to update relations and have confidence that our data will remain correct and consistent, we had to *Normalize* the relations to avoid anomalies.

So we have normalized relations and are ready to start updating, but once again it is not so simple! Once we start updating things we quickly will realize that problems can still occur. (Its never simple and easy.)

To illustrate, here is an example.

The other night my daughter was doing a little on-line shopping for a printer. She found the one she wanted and started the check-out process. But before the final confirmation screen came up there was an error screen that said that the transaction did not complete. Sensing something was off, she then checked her bank account and sure enough there was a debit for the cost of the printer, but no printer was going to ship!

She immediately called the store and asked what was going on. They confirmed that no order was placed and told her she needed to talk to the bank about her money. She was all set to call the bank the next day, but a little later on that evening she checked her bank account again and the charge had been reversed.

So what happened here?

She was trying to perform a *transaction* to purchase a printer. The transaction consisted of a series of steps:

1. Select an item (printer) and click purchase.
2. The inventory level of the printer is checked and updated.
3. Money for the purchase is debited from the credit card.
4. A shipping order is created and sent to the warehouse.
5. The printer ships.
6. The printer arrives.
7. The transaction is complete.

When she was ordering the printer it got to step 3 but then something went wrong and the remaining steps did not take place. For a time things were left in an inconsistent state. Money was debited but no new printer was on the way! Once the system realized this it undid the steps that had been completed so that everything was back to the way it was before the shopping started.

In case you were worried, there is a happy ending to the story. The next day she found that the printer was in stock at the local Best Buy so we drove over and picked it up.

Database Transactions

In the database world a *transaction* is a *logical* unit of work, like purchasing a printer. The unit of work is something that makes an update to the database.

We participate in transactions all day long. Every time you buy or sell something, every time you request a web page, every time you are open a document and save it, every email you send, you are performing a transaction. Often a transaction consists of many steps or operations, as in the example above.

A good example to think about is a student registration system. When you register for a course a number of things need to happen.

1. check the number of seats available. If it is more than 0, then decrement that number;
2. add your student id to course list;
3. add a debit to your account for the tuition.

You can clearly see, that like the ill-fated printer purchase, all the steps need to complete otherwise there will be a seat taken up with no student in it, or worse, you will owe MRU money for a class you are not in.

Another thing to think about is the fact that you will not be the only person registering for courses! Imagine a world where only one student at a time could register classes. You would have to queue up and wait. It would not really be practical. So we need to have multiple people registering at once and be sure to not turn someone away when there actually is a seat or to over enrol the course.

A registration system like this enforces strict consistency — it will not allow two people to register for the same seat. Contrast that to Amazon. Do you know what they do if there is only one copy of Mark Manson's *The Subtle Art of Not Giving a F*ck* in stock and you buy one from here in Calgary at the same time as someone else buys a copy from Australia?

One of you will get an apology and a refund. This is relaxed consistency and it wouldn't work so well for registration or banking, but is fine for Amazon.

Registration Back in the Day When I worked at Kwantlen College in the 1980s students were all assigned a time, in 15 minute blocks, over the course of three or four days. They would show up on campus, hopefully with a registration plan, and an alternate plan. They would fill in the registration form (we still have those for overrides and the like) with the classes they want and get in line. At the head of the line they would hand the paper to a registration person who would enter it into the system. One of my jobs as the IT support for this was to run reports every few minutes that showed the classes with a low or zero seats available. We had a dot matrix printer that printed this in large format and the report would be posted so students in line could see it. Every time the poster went up students would curse and rework their schedule.

Once your course were entered you headed over to the cashiers and paid in full for your courses.

So much for the good ole days.

For a longer example and to illustrate the key issues that we need to be concerned about we will work through an extended example of a speciality store that sells little balls of sweet deep fried dough. If we were doing this class in person we would get to have treat and eat some fried dough, but alas you will just have to imagine it this semester.

THE TIMBITS® STORE

Imagine a table at front of class, that is the store. There is a supply of Timbits® and a cash float. We will have one or more store-keepers and several customers.

Timbits® cost $1 each.

Every purchase (transaction) must follow this protocol:

1. Transaction begins
2. Customer: Do you have X Timbits® for sale?
3. If No, customer leaves, dejectedly. End

4. If Yes, store-keeper hands over the Timbits®.
5. Customer hands over the money
6. Store-keeper hands over any change owed the customer
7. Customer: Thank you!
8. Clerk: Have a great day, and enjoy your fried dough!
9. Transaction ends.

All or Nothing, Atomicity

Imagine if a customer comes to the counter and asks for some Timbits®. They are in stock, so the store-keeper passes them to the customer, who then just leaves without paying.

After losing a bunch of money, the storekeeper decides to change the protocol to avoid that, and has customer hand over money first. This usually works except when just before they hand over the Timbits®they get a text that they must answer, get distracted and never produce the Timbits®.

In both of these cases not all of the steps were completed and we end up with someone getting the short end of the stick.

How can we avoid this situation? Likely have someone monitor and ensure that all steps complete. This is what is done with high value transactions like real estate purchases. A lawyer supervises the whole thing. We have all seen movies where someone is handing over money to free a hostage or buy drugs or something and the steps take to avoid getting stiffed — “put the bag with the money on the ground and walk away” — because they can’t get a lawyer to help.

If not all steps complete, what should happen? Everything should be reversed until we are back to the state that existed before the transaction started. Remember the printer purchase? The charge was reversed.

Consistency

A business needs strict accounting of all goods, cash etc. For our store, it is important that at any given time there must be enough cash taken in to account for all of the goods that have disappeared.

At the start of the day we have N Timbits®. Each one sells for \$1. At any point during the day we will have sold S Timbits® and have M dollars in cash, proceeds from the sales. Therefore,

$N - S + M = N$

Must be true, all day long. Is it?

What could have to happen for it not to be true? Over or under-paying, or giving someone an extra Timbit® could lead to this situation.

During a sale it may not be true. If the money is in the till but the Timbits®have not yet been handed over the equation will not be true, but it should always be true after or before a transaction.

How can we ensure that this is true? We need to check that before we allow a transaction to end, that none of the consistency rules are violated. If the database would be left in an inconsistent state, we must reverse all the steps and return the system to the state that existed before the transaction started.

Isolation

As you know, Timbits® are very popular and having only one clerk may mean that the lineups will stretch all the way out to Mainstreet. What is the solution? We can add another clerk.

Two customers (A and B) come up to two clerks (X and Y). Each ask if there is a Timbit® for sale. "Yes" both X and Y reply, "you are in luck, there is one left".

Clerk X hands the last of the Timbits® to customer A. Clerk Y has to inform his customer that he in fact, lied and there are no more Timbits®. This is very bad for repeat business. It is worse if we collect money first. "Oops, I have to return your money!" We cannot sell the same Timbit® twice!

Other things can go wrong.

- A clerk could give the change to the wrong person.
- First person: do you have a Timbit®? Yes. Hand it over. Second person asks for a Timbit®, "No, none left". First person has no money so the Timbit® goes back in the bin. Second person does not get a Timbit® when there was in fact one available.

How can we ensure that these problems do not occur? We could simply have one transaction complete before any others start, but this pushes us back to low throughput single clerk scenario.

Some form of locking is needed. One clerk could put a hold on a Timbit® and either sell it or take the hold off and make it available to the other clerk if the sale does not complete.

Durability

A customer buys a Timbit®. Sometime later decides he does not want it and drops it back in the box. Timbits® are food. No returns!

A customer buys a Timbit®and the transaction completes and immediately a fire alarm goes off and everyone scrambles out of the building.

When everyone comes back in, the fact that that last sale completed must still be recorded.

All sales are final. If the transaction completes it stays completed.

What Have We Figured Out?

The Atomicity Property. When a transaction requires more than one operation to complete, it is important that all of the operations complete. If they do not, we need to reverse any operations that have taken place and restore the situation as it was prior to the start of the transaction.

The Consistency Property. There are certain properties that must hold for the system to remain consistent. In the Timbit® store this means that the number of Timbits®remaining plus the amount of money collected should remain the same all day long. In a database this could mean that foreign keys fields always point to a valid primary key in another table.

The Isolation Property. Since several transactions can be happening at once, for efficiency sake, we need to ensure that transactions do not interfere with each other. That is, some intermediate result of one transaction will not be visible to another transaction.

The Durability Property. Once a transaction is complete, it must remain complete. In the case of the Timbit® store this means no returns. In a database this means that even if the database fails, once a transaction is complete, it must be permanently recorded.

ACID Together these properties are referred to as the ACID properties. Most RDBMS include tools to allow a developer to ensure that the transactions that are taking place can adhere to the ACID properties.

Typically these include:

- A way to group database operations into a logical transaction;
- A way to signal that a transaction is complete, usually this is called a *COMMIT*.
- A way to signal that a transaction will not complete, usually this is called a *ROLLBACK*, and reverse all operations that have not been done so far.
- A way to provide locking at various levels. Locking of whole tables, locking of particular rows. As well you may need to have an exclusive lock — no one can even look at the data — or a shared lock where people can look at the data, but not update it.
- The RDBMS must provide ways to backup and restore the database — up to and including all committed transactions.

While the RDBMS does provide these tools, it can't make you use them, so in the end it is always up to the application developer to ensure that ACID properties are being adhered to. It is best practise to push as much as possible to the database level where the robust tools can take over.

Non ACID Databases One of the things that we will see when we look at NoSQL databases is that most of them are not ACID compliant. Often this is because the architecture of the NoSQL database does not fit with ACID transactions, but it is also because NoSQL databases are optimized for different properties, namely scalability and availability, rather than consistency.

When working with NoSQL, if you need ACID transactions you will need to build that into the application rather than rely on the DBMS to handle it for you.

Transactions in ORACLE

Transactions are controlled with three statements:

1. `SET TRANSACTION NAME 'Tname';` .
2. `SAVEPOINT savepointname;` .
3. `ROLLBACK;` — performs a rollback, ends the transaction and reverses any changes.
4. `ROLLBACK TO SAVEPOINT savepointname;` — performs a rollback as far as the savepoint.
5. `COMMIT;` — commits the current transaction, ending it and making all changes permanent.

When using the **SQL*PLus** client, by default `AUTOCOMMIT` is on. This means that every statement is surrounded by an implied `SET TRANSACTION NAME 'aa'; stmt; COMMIT;`

This is not always what we want so we can turn autocommit off with `SET AUTO OFF;`

Though most of the time, appropriate locking is done by the DBMS, sometimes you may want to explicitly lock a table:

```
LOCK TABLES foo AS foo1 READ, bar AS bar1 WRITE;
```

and you can unlock the table with

```
UNLOCK TABLES;
```

It is also possible to lock only certain rows with

```
SELECT ... FOR UPDATE;
```

TRANSACTIONS

1. What is a transaction, in the database context?
2. What are the ACID properties of transactions? Name each and explain why they are needed.
3. What is the lost update problem?
4. Why do we need to be able to provide concurrent access to our databases?
5. What is locking?
6. What types of locks are available in MySQL?
7. What is a deadlock? What techniques are used to mitigate the deadlock problem?