RDBMS Transactions

Transactions

- We have mentioned the importance valid relations
- When we looked at updating those relations, we realised 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.
- 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 see that problems can still occur: It’s never simple and easy…

Buying a printer on-line

- We found one we liked, select it and checked-out
- Before the final confirmation screen came up there was an error screen: transaction did not complete
- We checked our bank account, and the money was taken
- Called the store, they said that transaction –in fact- did not go through. We planned to call the bank the next day!
- That same evening, another bank transaction appeared, reversing the original transaction
- What has happened?

Logic Unit - Transaction

The logic transaction had several steps (operations, perhaps?):

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.

So, what happened?

- When we ordered 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 reversed the steps that had been completed so that everything was back to the way it was before the shopping started
- Actually, this situation meant a happy ending! The system could have not recovered!

Database Transactions

- In the database world a transaction is a logical unit of work, like pur-chasing 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.

Database Transactions

When you are doing a bank transaction, or buying a plane ticket, or registering for at class at MRU:

- Are you the only one performing a transaction, or several of them are happening simultaneously?
- Buying goods on-line: Relaxed consistency
  - Is there is only a copy of an item and two people are trying to get it, one of them will receive a successful message while the other will get an apology and a refund
- Buying a plane ticket or doing a bank operation:
  - The system won't allow two people to grasp the same –only one left-seat
  - What about two bank transactions on the same account going simultaneously in?

Welcome to the wonderful world of ACID!

How can we address the aforementioned conditions/situations?

ATOMICITY: a logic transaction is atomic in nature!

- If not all steps are successful, what should happen?
  - **Everything should be reversed** until we are back to the state that existed before the transaction

CONSISTENCY: database status must be consistent before and after each transaction

- If you sell Timbits in a store (@$1 each) and at the start of the day you have **N** Timbits, and at a point in time you have **M** dollars in cash, and have sold **S** Timbits, then: N – S + M = N
  - If this does not hold, then we could be Over or Under paying, or giving someone the wrong amount of Timbits
- 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 do we make it happen then:
  - 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.

Welcome to the wonderful world of ACID!

ISOLATION: what if we hire two more salespeople for our store? Now we can sell 3 Timbits simultaneously!

- There are only 2 Timbits left and three customers ask for one. Clerks X, Y and Z, all sell a Timbit and charge the customer. Now, clerk Z must inform the customer that there are no more Timbits and give the money back!
  - We can solve this by **having each transaction complete BEFORE the next start**, but then we go back to low throughput…
  - We need some sort of **LOCKING**! 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: after buying a Timbit, a customer comes back and drop it in the box, but 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.

ATOMICITY & CONSISTENCY

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.

ISOLATION & DURABILITY

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: Implementation

All these properties together are known as ACID properties. Most RDBMS include this tools to allow developers to ensure that the transactions that are taking place can adhere to the ACID properties. 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**.
- The RDBMS must provide ways to backup and restore the database -up to and including all committed transactions.

ACID: Implementation

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

Locks

We said that we need to provide locking at various **levels** and using different **types of locks**.

- Levels
  - Database
  - Tablespace
  - Table
  - Record/Row/Relation
  - What about Columns/Attributes? No! We should not!
- Types
  - Exclusive lock: no one can even look at the data
  - Shared lock: where people can look at the data, but not update it.

No 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.

An ORACLE Transaction Example

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.**

Other RDBMS may use different instructions, but mostly they will have equivalent operators.

Log System Structure (Journals & Journaling)

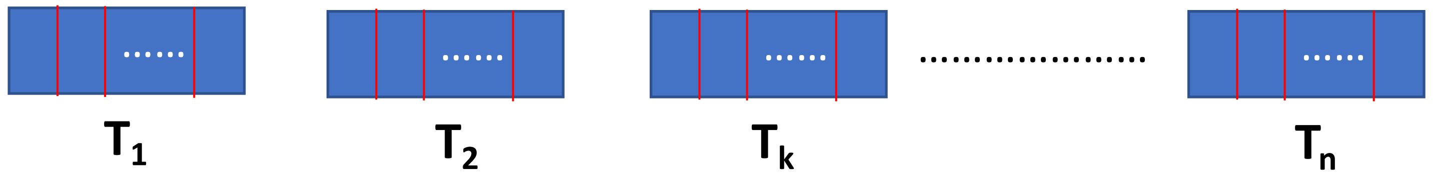
What do we need to know in order to be able to reverse an UPDATE transaction?

- Where it is happening
  - Table Space -> Table -> Relation/Row
- What is being affected
  - Attribute name/location
  - Value **BEFORE** update
  - Value **AFTER** update
- When the transaction happened
  - Time Stamp
- Unique Transaction ID (it could derivate from Time Stamp)
- Who made it
  - User ID of the person running the transaction

Journal/Log (possible) Structure

T_k						
TransID	Table	Attribute	BEFORE Image	AFTER Image	Time Stamp	UserID Owner Transaction

A Journal/Log is a collection of Transactions, with Structure T_k



Applications

- You will use a Log/Journal to revert an incomplete (logical) transaction
  - It could be done with AUTO-ROLLBACK (must advanced RDBMS do it)
  - It could be done by using latest “reliable” Backup, and restore the consistency of the Database by applying FORWARD the data stored in the the Log/Journal
  - It is assumed that the latest Reliable Backup* was taken at a point of 100% consistency of the RDBMS

*: a backup is a full copy of your Database.

In your Assignment No. 1

- It could be done with AUTO-ROLLBACK (must advanced RDBMS do it)
  - This corresponds to the Bonus version of the assignment

- It could be done by using latest “reliable” Backup, and restore the consistency of the Database by applying FORWARD the data stored in the the Log/Journal
  - You are PARTIALLY doing this in your Assignment No. 1
    - ➢ When an ERROR is detected, you will:
      - ✓ STOP all operations
      - ✓ Show the (current) Attributes Values in your Database (**simulating** the contents of a Backup)
      - ✓ Show that your Journal/Log contains all the bit and pieces required to **revert** the database to a consistent state