CS 325 - DB Reading Packet 9: "Transaction management, part 1"

Sources:

* Ricardo, "Databases Illuminated", Chapter 10, Jones and Bartlett.
* Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Prentice Hall, 1999.
* Rob and Coronel, "Database Systems: Design, Implementation, and Management", 3rd Edition, International Thomson Publishing, 1997.
* Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
* Korth and Silberschatz, "Database System Concepts"
* Sunderraman, "Oracle 9i Programming: A Primer", Addison-Wesley.
* Ullman, "Principles of Database Systems", 2nd Edition, Computer Science Press.

Transaction Management and Concurrency Control - Part 1

Today's goal is to introduce some more advanced database concepts; many of these could easily be the topic of an entire course. The goal here is to introduce a collection of concepts related to transaction management and concurrency control.

These particular topics have at least some relationship to the concept of a **transaction**. In database terms, we'll define **transaction** more specifically than you might have seen before -- it starts with the concept of a transaction being a **logical unit of work (luw)**. A transaction represents real-world events such as a sale of an item, a deposit into an account, a transfer of funds between accounts, a registration for a course, a reservation for a hotel room, etc.

In reality, even though we think of such events as being one "thing" -- that's what we mean by a logical *unit*, a single thing as far as we are concerned -- such a transaction actually consists of a number of steps in terms of actual execution. Consider the transaction of transferring $100 from a banking customer's savings account to their checking account. We think of that as a single event, a unit in logical/meaning terms, but it takes a number of steps within a computer to accomplish this transaction; for example:

* read savings account balance (from the database into a local variable or a register)
* reduce savings account balance by 100 (in that local variable or register)
* write the new savings account balance (from the local variable or register back into the database)
* read the checking account balance (from the database into a local variable or a register)
* increase the checking account balance by 100 (in the local variable or register)
* write the new checking account balance (from the local variable or register back into the database)

Logically, we expect that all of these steps will be done. But, consider: what if the first 4 steps were completed, but the 5th and 6th steps weren't (perhaps because of a power outage, or power surge, or a hard drive failure, etc.)? You would have one unhappy customer, since they have just lost $100! We would like to ensure that a transaction is either completed, all of its steps done, or is not started at all.

So, we'll define a **transaction** as follows:

> A **transaction** is a **logical** unit of work that must be either **entirely completed** or **entirely aborted**; **no** intermediate states are acceptable.

or, if you prefer:

> A **transaction**, also known as an **atomic transaction** or a **logical unit of work (LUW)**, is a **series** of actions (representing a single "logical" unit of work) taken on the database such that either **all** of the actions are done successfully, or **none** of them are and the database remains **unchanged**.

A multistep transaction, like the savings-to-checking transfer above, must **not** be only **partially** completed. So, if we define the 6 steps shown above for a savings-to-checking transfer as a transaction, then we are saying that either all 6 of those steps will be done, or **none** of them will, preventing the confusion/unhappy result we discussed above.

Defining a series of steps as a transaction is an important aspect of **concurrency control**.

Next, we want to revisit a concept we mentioned earlier this semester: consistency. A **consistent database state** is one in which all **data integrity constraints** are satisfied. So, when transactions enter into the picture, a transaction that changes the **contents** of the database must alter the database from one consistent state to another. To ensure this, in general all transactions are controlled and executed by the DBMS to guarantee database integrity.

ASIDE: levels of DBMS support for database integrity

We have talked a bit about the general idea of database integrity -- how the idea of data **integrity** is not so much that the data in a collection, whether of files or within a database, is correct, but that it is logically **consistent** within that collection of data [Kroenke]. In looking at different DBMS's, it can be important to look at their levels of support for data integrity; to that end, the following terms are frequently used in such discussions:

Entity Integrity

If the DBMS requires "legal", unique, non-null primary keys, then a DBMS can be said to support **entity integrity**. In a DBMS that supports entity integrity, once you define an attribute or a set of attributes as the primary key for a table, the DBMS will not permit insertion of a row with a null primary key, or with a primary key whose value is the same as that in a currently existing row. In this way, then, you can be assured that such a table's primary key indeed uniquely identifies/uniquely determines a row. (And if this is the case, you can see that duplicate rows within such a table will also be impossible, since if the primary key of two rows cannot be the same, neither can two entire rows.)

I would be very suspicious of a DBMS that does not at least support entity integrity...! Oracle certainly supports this, as you have probably seen during the course of this semester.

Domain Integrity

If the DBMS allows you to specify a particular domain for an attribute, and then ensures that any insertions or updates involving that attribute are within that particular set of possible values, then a DBMS can be said to support **domain integrity**.

Domain integrity is a bit more slippery to support absolutely than entity integrity; what exactly is the domain of an attribute? By definition, it is the set of possible values for that attribute; certainly being able to say that the values of an attribute are a certain data type is at least partial support for domain integrity. Since another aspect of the domain could be considered to be whether NULL is included in that domain or not, the ability to specify whether than attribute can be null is further support for domain integrity. But what if the actual, desired domain is just certain integers? or something which doesn't have a DBMS-supported type, such as telephone numbers? A DBMS that allows more restriction on an attribute domain than just stating a data type is providing stronger support for domain integrity; so, for example, Oracle's **check** clause allows a higher level of domain integrity, allowing one to specify, for example, that the domain should be values within a certain range, or that the domain consists of a set of explicitly-specified values. A DBMS that supports user-defined data types for attributes would allow stronger domain integrity support still.

Referential Integrity

If the DBMS allows you to specify a foreign key relationship, and subsequently that referential constraint is enforced by the DBMS, then that DBMS can be said to support **referential integrity**. In such a DBMS, if, for example, you try to insert a child row containing a value for the foreign key that is not in the corresponding parent table, the DBMS will not permit the insertion of that child row. Likewise, the DBMS will not permit the deletion of a parent row if a child row exists that references that particular parent row in its foreign key attribute(s) (unless "on delete cascade", "on delete set null", or some other such means are set to prevent a breach of referential integrity).

(Some DBMS's permit you to specify "on delete cascade" for a foreign key -- this means that, if a parent row is deleted, all children rows will also be deleted, thus preventing a breach of referential integrity. A DBMS that permits you to specify "on delete set null" for a foreign key will, if a parent row is deleted, set the foreign key field for all affected children rows to NULL -- which will only be appropriate for foreign keys whose values are permitted to be NULL, of course. I believe some other options are possible as well, depending on the DBMS. However, although these may provide more convenience -- less work that the database user needs to do before particular inserts or deletes will be permitted -- notice that referential integrity is still fully supported even with just the basic restrictions.)

You have likely experienced how Oracle indeed supports referential integrity, indeed not permitting the insertion of a child row within a corresponding parent row and indeed not permitting the deletion of a parent row for which there is a corresponding child. Some versions of Oracle permit additional options such as those described above, but not all versions of Oracle do.

Transaction Integrity

Finally, if the DBMS supports atomicity of transactions -- allowing to specify a sequence of steps that will then be enacted as if they were a logical unit of work, either completely done, or not even begun, supported and ensured by the DBMS -- then that DBMS can be said to support **transaction integrity**.

We'll be discussing, later in this handout, one of the ways in which SQL supports transaction integrity.

The five main database transaction properties: ACIDS

In dealing with transactions, there are four (or five, depending on the textbook) main database **transaction properties**, that all transactions need to meet satisfy; they make up the acronym ACID, which we'll extend to ACIDS:

* **atomicity**
* **consistency**
* **isolation**
* **durability**

and, we'll also include:

* **serializability**

Atomicity

We've already mentioned this transaction property, as it is tied into the very definition of a transaction. **Atomicity** requires that all operations/parts/steps of a transaction be completed; if not, the transaction is aborted. (Think of the idea of an "atom", the smallest unit of matter, that cannot be divided (or at least not easily).) Connolly and Begg, p. 553, call this the "all or nothing" property, this idea that the transaction be treated as "an **indivisible** unit of work that is either performed in its entirety or not performed at all. It is the responsibility of the recovery system of the DBMS to ensure atomicity." We want a transaction to be considered a single, **indivisible**, logical unit of work.

Consistency

The transaction property of consistency is satisfied if performing a transaction does not violate database integrity -- "a transaction must transform the database from one consistent state to another consistent state." [Connolly and Begg, pp. 553-554] "It is the responsibility of both the DBMS and the application developers to ensure consistency. The DBMS can ensure consistency by enforcing all the constraints that have been specified on the database schema, such as integrity ... [and referential and domain] constraints" -- but the application program has to do its part, too, working also to enforce the scenario's business rules.

Isolation

The transaction property of isolation means that the data used during the execution of a transaction cannot be <u>used</u> by a **second** transaction until the first one is completed (at least, the effect must be as if this was the case). Transactions need to at least conceptually execute independently of one another, even if in reality their steps are interleaved, so that the steps of different transactions are being executed concurrently. "In other words, the **partial** effects of incomplete transactions should **not** be visible to other transactions. It is the responsibility of the **concurrency control** subsystem to ensure isolation."

[Connolly and Begg, p. 554] So, for example, if you are performing a transfer-of-funds transaction, you would not want another transaction to try to query either the checking or the savings account balances while the transfer is taking place, possibly reading an in-progress value that doesn't reflect the actual balance.

What kinds of problems can occur if isolation is not ensured? Both Kroenke (in Chapter 8) and Connolly and Begg (pp. 555-558) include the following amongst the problems that can occur:

* **dirty read/uncommitted dependency**: "when one transaction **reads** a changed record that has **not been committed** by the database"

* **nonrepeatable reads**: "when a transaction rereads data it had previously read and finds **modifications** or **deletions** caused by a **committed** transaction"

* **phantom reads**: "when a transaction rereads data and finds new rows that were inserted by a committed transaction **since** a prior read"

Durability

Durability indicates the permanence of the database's consistent state. In contrast to the property of consistency, which stipulates that a transaction has to transform a database from one consistent state to another consistent state, durability is about the database maintaining its consistent state ***between*** transactions. That is, between transactions, the database's consistent state endures/persistes; its state is not lost even due to system failure, power outage, etc. "The effects of a successfully completed (committed) transaction are permanently recorded in the db and must not be lost because of a subsequent failure. It is the responsibility of the **recovery** subsystem to ensure durability." [Connolly and Begg, p. 554] That is, the DBMS needs to "provide facilities to **recover** the changes of all **committed** actions when necessary".

Serializability

The transaction property of serializability means that the **concurrent** execution of transactions is **equivalent** to the case where the transactions executed **serially** in some arbitrary order. (Get it? **serial**izability?) The idea here is, **if** the results from the **concurrent** execution of the transactions are the same as the results from **any** of the **possible non-concurrent** execution orders for those transactions, then you haven't introduced errors/integrity problems due to the concurrent execution of those transactions. It is typically the DBMS's responsibility to ensure that the database suffers from no anomalies due to concurrent processing of transactions (although sometimes application programs can contribute to this as well).

More formally: when two or more transactions are processed concurrently, the results in the database should be **logically consistent** with the results that would have been achieved had the transactions been processed in **<u>any</u> arbitrary serial fashion**.

A scheme for processing concurrent transactions where serializability is achieved is said to be **serializable**.

Now that we have discussed these ACIDS properties, consider a single-user DBMS. In that setting, many of these properties are not hard to achieve:

* **serializability** is trivial, because a serial execution of transactions is, of course, equivalent to one of the possible serial orders of executing those transactions…! (that is, there isn't concurrent execution of transactions taking place in this setting)

* **isolation** is not a problem, either, because only one transaction is executed at a time -- thus, there is no chance that another transaction can be trying to use the data mid-transaction.

* **consistency** is still an issue; the DBMS still needs to ensure that even serial transactions' changes still each meet any database integrity constraints.

* **atomicity** is still a concern, also; the DBMS still should ensure that either all of a transaction is done, or that it is as if it was never begun. Single user DBMS's may not have sophisticated facilities to ensure this, however.

* **durability** is still a concern, also, but, again, a single user DBMS is unlikely to have sophisticated backup and recovery mechanisms to make sure that a committed transaction's effects remain even in the face of failures between transactions. It tends to be the user of the single-user DBMS who will likely have to ensure this (through careful and disciplined manual backups, keeping backups in different locations, etc.)

A multi-user DBMS that supports concurrent transactions will have more concerns with all of these, but is also more likely to provide more sophisticated support for all of them as well.

Let us discuss in a bit more detail what we mean by concurrent transactions. Concurrent processing of transactions happens when the steps of two or more transactions are **interleaved** with one another, which is very common in a multi-user environment. Consider nrs-labs -- many of us can be logged in at once, working concurrently, but we each don't get the CPU all to ourselves. As you have discussed/will discuss in CS 374 (if not before), instead everyone logged into a multi-user, timesharing operation system **shares** the CPU, alternately getting slices of time to use the CPU. It is a similar idea for concurrent execution of transactions -- usually a DBMS doesn't process all the steps of one transaction, and then all the steps of the next transactions -- instead, it interleaves the steps of the transactions taking place concurrently. To maintain isolation and serializability in such a setting is among the responsibilities of the DBMS that permits concurrent transactions -- while still assuring atomicity, consistency, and durability. The DBMS will need to provide means for **concurrency control**.

SQL's support for transaction atomicity: COMMIT and ROLLBACK

Before we discuss some of different means of providing concurrency control, however, let's mention one way of implementing atomicity, and a way of helping to achieve durability. We'll start by discussing SQL's support for transaction atomicity: its **COMMIT** and **ROLLBACK** statements.

According to the American National Standards Institute (ANSI) standards for SQL database transactions, it is required that, when a **transaction sequence** is **initiated** by a user or an application program, it must **continue** through all succeeding SQL statements until one of the following four events occurs:

* a **COMMIT** statement is reached: the transaction is considered to be successfully completed, and so it is proper that all changes are permanently recorded in the database;

* a **ROLLBACK** statement is reached: the transaction is considered to have been aborted, and so any actions taken thus far must be "undone", so that the effect is that the transaction was **never even begun**; the rollback rolls the database back to its previous consistent state (to its state as a result of the latest COMMIT);

* the end of a program is successfully reached; this is considered to be equivalent to a **COMMIT**, and so all changes are permanently recorded in the database;

* the program is abnormally terminated; this needs to be treated as equivalent to a **ROLLBACK**, with changes aborted and the database rolled back to its previous consistent state (to its state as a result of the latest COMMIT)

The ANSI SQL standard only requires that COMMIT and ROLLBACK be provided to support transactions, but different implementations of SQL may provide additional support for defining transactions as well. One can consider a transaction to have implicitly begun when the first SQL statement is encountered, but some SQL implementations may include additional statements, such as BEGIN TRANSACTION, and some even permit parameter assignments for a transaction as part of a BEGIN transaction statement.

Remember how we discussed transaction integrity, earlier in this handout? A DBMS that includes an implementation of SQL that meets the ANSI standard, including COMMIT and ROLLBACK statements, is thus providing support for transaction integrity (and Oracle SQL does indeed include COMMIT and ROLLBACK).

An example of one way to support durability: transaction logs

Now let's briefly discuss one of the several means that a DBMS might employ to help ensure **durability**: use of a **transaction log**. A **transaction log** keeps track of the actions of all transactions that update the database, and can then be used for **database recovery** if some failure occurs. One approach to this can be as follows:

* periodically make a copy of the database (called a **database save**);

* keep a transaction log of the **changes** made by transactions against the database since the save;
    * this log contains a record of **data changes** in **chronological order**

    * changes are written to the log **before** they are made to the database

        * (can you see why this would be important? if the system crashes between the time the

change is logged and the database is actually changed, then at worst there is now a record of an unapplied change.

If one changed the database before writing the change to the log, and the system crashed between the change and the log entry regarding that change, then that could result in a database change not recorded in the log -- depending on the recovery approach being used, this could result in a data anomaly.)

* when failure occurs, you then either:
    * **rollforward**: the database is restored to the previous saved state, and only appropriate changes logged since the save are reapplied;
        * which are the changes that should be reapplied? only those actions that are part of **committed** transactions. That is, you only roll forward **committed** transactions.

        * **note** that this is **not** the same as reprocessing the applications! Just the database changes made as a result of committed transactions are redone. The application program(s) that originally made the transactions are **not** involved in this. We just redo the database changes **resulting from** those transactions that had been committed at the time of the failure.

        * Note that the actions of transactions that ended up being rolled back before the time of the failure are not redone; note that the actions of transactions that had not yet been committed at the time of the failure are not redone. Thus, when the rollforward is complete, the database is in a consistent state reflecting the effects of all committed transactions at the time of the failure.

    * **rollback:** in this approach, you have the database state at the time the failure occurred. You then **undo** any changes made by transactions that had not been committed at the time of the failure, so that none of them violate atomicity by being only partially done. Note that this will not be possible unless you still have that database state at the time of the failure...

Note that the transaction log itself needs to be implemented quite robustly, on non-volatile memory -- it may itself be in the form of a database, and sometimes a DBMS supports mirroring the log on several different disks or media to reduce the risk of system failure that might otherwise corrupt the log.

A FEW words on database recovery management

Recovery means restoring a database from a given (usually inconsistent) state to a previously consistent state.

In recovery, we want to make sure, as we have mentioned previously, that transactions are treated as **atomic** --- like a single, logical, indivisible piece of work. So, each transaction either has all of its actions applied and completed, or **none** of them, so that it is as if it has never even been started. As we mentioned in the transaction log discussion, then, transactions that were committed must be redone as part of recovery, and those not committed must be rolled back or never started (depending on the recovery approach being used).

Backup and recovery capabilities are very important components of multi-user DBMS's! Some allow for **automatic** database backups, that the database administrator (DBA) can schedule, to automatically backup the database to permanent secondary storage devices like disks or tapes; such backups may be mirrored to different locations, for additional protection.

Different levels of backup are possible:

* a full backup or "dump" of the database;

* a differential backup of the database, in which only the modifications from the last full backup are copied;
    * a differential backup is faster, since you only copy the changes, not the entire database;
    * but, of course, it may slow down recovery, since, in the case of failure, you have to obtain the previous full backup and apply the modifications to that.

More on Concurrency Control

Recall that **concurrency control** is the management of concurrent transaction execution (interleaving of the steps of different, concurrent transactions). Based on our earlier discussion, you should be able to see that multiple users doing concurrent transactions has the potential for trouble -- with users processing the database data concurrently, one user's work may interfere with another's (both should not be able to purchase the same airline seat for the same flight, for example). And, recovery after failure is more complicated; simply reprocessing the transactions could actually result in different results from the original results, if one is not careful. (This is one reason that transaction logs log database changes in chronological order.)

We've already mentioned some of the problems that we want to avoid in concurrent transaction execution, such as dirty reads, nonrepeatable reads, and phantom reads. Another is **lost updates** (sometimes called the **concurrent update** problem). Consider the following scenario: the row in a relation Item for item_number 100 shows that, currently, 10 of item_number 100 are in stock (Its quantity attribute has the value 10.) Assume that, for an order transaction, you need to:

(a) read the ordered item's current quantity attribute into local memory or a register;
(b) change that copy of the quantity to its new value (as a result of this order transaction);
(c) write the new value of quantity to be the ordered item's quantity attribute's updated value;

Consider the following sequence of steps of two interleaved order transactions A and B, each trying to process orders of item_number 100:

1. A seeks to order 5 of item_number 100, and B seeks to order 3 of item_number 100.
2. A (a) - reads item_number 100's quantity, and reads a quantity of 10
3. B (a) - reads item_number 100's quantity, and reads a quantity of 10
4. A (b) - change its copy of quantity from 10 to 5 (want to order 5 of this item)
5. A (c) - write the new value of quantity, 5, to be item_number 100's new quantity
6. B (b) - change its copy of quantity from 10 to 7 (want to order 3 of this item)
7. B (c) - write the new value of quantity, 7, to be item_number 100's new quantity

See the problem? We have **lost** A's **update** (thus, the name **lost update**). It occurred because B read data that A was about to update, so one of the updates ended up being essentially ignored. Where the database should have reflected sales of 8 items, so that the resulting quantity should have ended up as 2, it instead ended up as 5.

A multi-user DBMS tends to have a built-in **scheduler** that establishes the order in which the steps within concurrent transactions are executed, **interleaving** those steps to ensure **serializability**. It has classic tradeoffs to balance; it wants to maximize "safe" concurrency for better transaction performance, but still ensure serializability and isolation, and ensure these in such a way that the overhead for ensuring these does not itself overly impact transaction performance.

Algorithms for concurrency control are a large topic in their own right! We shall mention here just three of the classic categories of algorithms for concurrency control, and within them briefly describe just a simple example or two of the many examples within each category. Each is worthy of further study in its own right.

Three classic categories of algorithms for concurrency control are:

* locks
* timestamping
* optimistic methods

In the next reading packet, we will discuss some examples of these algorithms.