CS 325 - DB Reading Packet 10: "Transaction management, part 2"

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 2

As we mentioned at the end of the previous reading packet, three classic categories of algorithms for concurrency control are:

* locks
* timestamping
* optimistic methods

Now, we will discuss some examples of these algorithms.

Locks

In locking (sometimes called **resource locking**), the idea is that we prevent some of the anomalies and inconsistencies mentioned, due to lost updates, dirty reads, etc., by preventing multiple transactions from having copies of the **same** data when that data may be about to be changed. You selectively **restrict concurrency** when information may be updated -- transactions must obtain a **lock** for data before they are permitted to use it (read it **or** write it), and the transaction then **releases** that lock when it is done, so another transaction can obtain a lock on that information if it needs to.

(Note that, in this discussion, we are assuming **implicit locks**, locks placed automatically by the DBMS, which is the norm for multi-user DBMS's that automatically initiate and enforce locking procedures, managed by a **lock manager**. Explicit **locks** would be locks issued as a result of commands made by the application program or query user to the DBMS, and we will not be discussing those here.)

Also note that **lock granularity** is another factor that can vary within a locking approach. **Lock granularity** is how much (or how little) you can lock at once; that is, when you obtain lock, how much is locked? A single cell within a row? a row? a column? a page? a table? the entire database?! Note the

tradeoff between these different lock granularities:

* **larger** granularity is easier for the DBMS to administer, but creates more (potentially-unnecessary) conflict, and could reduce potential "safe" concurrency;
    * consider locking a row versus locking a whole table -- it is less likely that two transactions will want to access the same row at the same time than that two transactions will want to access different rows of the same table at the same time. If the lock granularity is one row, then different transactions accessing different rows of that table -- that could safely proceed concurrently -- will be able to do so, but if the lock granularity is the whole table, they will not be permitted to, even though it would be "safe"; one will have to wait.

* **smaller** granularity is more difficult to administer (more details for the DBMS to keep track of and check, and more overhead for the locks), but conflict is less frequent, and more potential "safe" concurrency is possible.

There are a wide variety of locking algorithms and techniques; here we just mention a couple of the most classic approaches, **binary** locks and **shared/exclusive** locks.

Binary locks

Binary locks have only 2 states, 1 and 0 (locked and unlocked). If something is locked by one transaction, then **no** other transaction can use it, period, until that lock is released; if something is unlocked, then any transaction can lock it for its use. ("Something", here, being whatever lock granularity the DBMS is using.) Every database operation requires that the affected object be locked, and as a rule a transaction must unlock the object after its termination. So, every transaction requires a lock and unlock operation for **every** item to be accessed. (But, since we are assuming implicit locking, remember, this will be automatically managed and scheduled by the DBMS.)

Notice that this is relatively simple for the DBMS -- when a binary lock is requested, the DBMS determines if that something is already locked. If so, the requesting transaction has to wait; if not, the requesting transaction obtains the lock. That is:
Transaction T wants a binary lock on item I. Does it get it?

* if item I is currently locked, transaction T has to wait;
* if item I is currently unlocked, transaction T obtains the binary lock.

It can require as little as one bit to store the lock's state (since its only two states are 1 or 0), so the overhead is fairly low as well (although lock granularity will affect how much overhead is required). However, binary locking is also relatively restrictive, limiting potential "safe" concurrency: for example, concurrent reads are safe, but they will not be allowed under binary locking.

Shared/Exclusive locks (Read/Write locks)

Shared/exclusive locks (also called read/write locks) have 3 states: shared (or read) locked, exclusive (or write) locked, and unlocked.

An exclusive lock (write lock) locks the item from any other concurrent access; the transaction with an exclusive lock can both read and write the item, and no other transaction may do so while that item is exclusive-locked. (A transaction must obtain an exclusive lock on something before updating it!)

A shared lock (read lock) locks the item from being changed but not from being read; multiple transactions are permitted to obtain a shared lock on the same item at the same time. Note the increase in potential concurrency -- you permit concurrent reads when it is "safe" to do so (when the object does not have an **exclusive** lock).

Notice that the algorithm, here, is a little more involved than for binary locking; consider:

* transaction T wants a shared/read lock on item I; does it get it?
    * if item I is currently unlocked: YES, transaction T gets the shared lock;
    * if item I is currently shared-locked: YES, transaction T gets the shared lock;
    * if item I is currently exclusive-locked: NO, transaction T does not get the shared lock, and has to wait until the exclusive lock is released.

* transaction T wants an exclusive/write lock on item I; does it get it?
    * if item I is currently unlocked: YES, transaction T gets the exclusive lock;
    * if item I is currently shared-locked: NO, transaction T does not get the exclusive lock, and has to wait until (all the) shared lock(s) are released;
    * if item I is currently exclusive-locked: NO, transaction T does not get the exclusive lock, and has to wait until the exclusive lock is released.

Note that there is a little more overhead here -- you need at least 2 bits to represent the 3 possible lock states of unlocked, shared-locked, and exclusive-locked, and that's before you handle keeping track of how many transactions are currently have a shared lock on some item -- and the algorithm is a little more involved, but there is also more potential "safe" concurrency, especially if reads are more frequent than updates.

Two-phased locking

Note that locking helps to achieve isolation, but it does not, by itself, ensure serializability; that is, using locks alone does not necessarily result in a serializable transaction schedule. Additional protocols must be added to locking to ensure serializability.

One classic protocol for this is **two-phased locking**; serializability can be guaranteed if a two-phased locking protocol is used:

* with this strategy, transactions may obtain locks as necessary, but once the **first** lock (of any kind) is released, **no other lock** can be obtained.

* and, it is called **two-phased** because, when you enforce the above, it results in a transaction having a **growing phase**, in which locks are obtained, and a **shrinking phase**, in which locks are released.

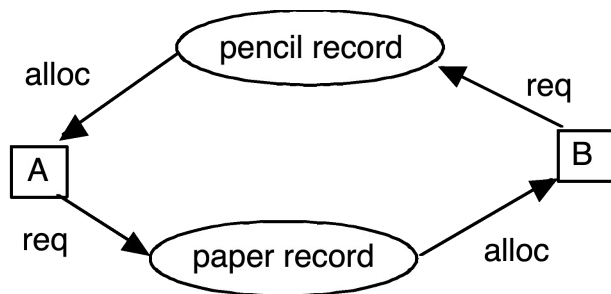
* Note that a more-restrictive (and easier-to-implement) variation of two-phased locking, used by DB2 (and some other DBMS's, also?), simply does not release **any** locks until a **COMMIT** or **ROLLBACK** command is issued;
    * so, no lock is released until the transaction has essentially been completely done, or completely **not** done;

* note that locks can thus be obtained **throughout** the transaction...the shrinking phase simply does not begin until the transaction is complete or aborted.

Deadlocks

While locking helps solve inconsistency problems due to concurrent transactions, it introduces another class of problems...**deadlock** conditions!

Consider the following scenario:



* A wants to get some pencils, and if they can get them, then they want to get paper

* B wants to get some paper, and if they can get them, then they want to get pencils

So...

1. A requests and obtains a lock on paper;
2. B requests and obtains a lock on pencil;
3. A requests a lock on pencil, but cannot get it, and so has to wait;
4. B requests a lock on paper, but cannot get it, and so has to wait.

See the problem? A and B are each waiting for something that the other has locked; they are locked in a so-called **deadly embrace**, they are in a state of **deadlock**.

Strategies for Deadlock Management

There are a number of strategies for deadlock management; those strategies include (but are not limited to):
* **timeouts**
* **detection**
* **prevention**

Timeouts

If a transaction requests a lock and has to wait, it will only wait for a system-defined period of time. [Connolly and Begg, p. 570] If a lock request times out, then the DBMS assumes that the transaction MAY be deadlocked (although it might not be...!) and aborts and automatically restarts it. This strategy is very simple, practical, and Connolly and Begg notes that it "is used by several commercial DBMS's".

Detection

Detection strategies allow deadlock to occur, detect it, and then break it. That is, the DBMS periodically tests the database for deadlocks (using, for example, a wait-for graph --- a dependencies graph. For example, it can build a dependencies graph based on what transactions are waiting for what items, and then look for cycles within that graph: any such cycle is a deadlock.). Once detected, the DBMS breaks the deadlock by selecting one of the deadlocked transactions, aborting it, and restarting it.

Prevention

Prevention strategies prevent deadlock from occurring at all. For example, a transaction requesting a new lock will be aborted if there is a possibility that a deadlock can occur as a result of that lock request -- (and remember that, based on two-phased locking, aborting a transaction causes all of its locks obtained up to that point to be released). The aborted transaction is the rescheduled for execution. Prevention works because it avoids the conditions that lead to deadlock; however, Connolly and Begg claim that these strategies are more difficult and generally avoided.

Timestamping Algorithms

Timestamping algorithms are an alternative to locking approaches for scheduling concurrent transactions; they are another, different class of algorithms for this.

In timestamping algorithms, a **time stamp** is assignment to each transaction when it is started. This time stamp is not based on a 24-hour clock, however -- it has some very particular requirements:

* it must be **global**
* it must be **unique** for each transaction
* it must be **monotonic** (it must have the property of **monotonicity --** the time stamp values must **always increase**)

All database operations within the same transaction will be considered to have the transaction's time stamp. The DBMS then ensures that **conflicting** operations are performed in **time stamp order**, thereby ensuring **serializability** of the transactions. (Do you see why this assures serializability? Because the concurrent transactions' effects thus must be the same as a serial order of those transactions, the serial order of their happening to be executed in time stamp order!) What if a transaction's conflicting operation would have to violate time stamp order to be done? Then that transaction will be aborted, rescheduled, and assigned a new (and larger) time stamp when it is started again.

Going into a bit more detail about this approach: with each data item **Q**, you associate **two** timestamp values:

* **W-ts(Q)** - the **largest** time stamp of any transaction that executed **write(Q)** <u>successfully</u>
* **R-ts(Q)** - the **largest** time stamp of any transaction that executed **read(Q)** <u>successfully</u>

Assume that **Ti** is a transaction, and that **TS(Ti)** is the time stamp of transaction Ti. Then the timestamp-ordering protocol ensures that any **conflicting** reads and writes are executed in **timestamp** order (thus ensuring serializability):

* If Ti issues **read(Q)**:
  * if TS(Ti) < W-ts(Q), then Ti needs to read a value of Q that was "already" overwritten by a "later" transaction (according to timestamp ordering);
    * the read(Q) will be **rejected**, and Ti will be aborted, rolled back, and restarted with a new (larger) time stamp;

  * else (if TS(Ti) >= W-ts(Q)), then it is "safe" to execute the read(Q);
    * the read(Q) will be executed, and
    * R-ts(Q) will be set to the **maximum** of R-ts(Q) and TS(Ti)
      * (make sure that this is clear! You never want to set R-ts(Q) to be **smaller** than it was before; it always needs to have the largest time-stamp of a "successful" read! If the latest successful read(Q) has a smaller time stamp than the current R-ts(Q), then R-ts(QP should be UNCHANGED.)

      * (it is important to note the time stamp of the "latest" read, so that another write does not get done "before" this read, in terms of time stamp ordering of conflicting operations)

* If Ti issues **write(Q)**:
  * if TS(Ti) < R-ts(Q), then a "later" transaction has "already" read and used Q -- overwriting it now would be bad (and would violate time stamp ordering), because there is no way now for that "later" transaction to see the new value.
    * the write(Q) will be **rejected**, and Ti will be aborted, rolled back, and restarted with a new (larger) time stamp;

  * else if TS(Ti) < W-ts(Q), then Ti is trying to write an "obsolete" value of Q (already overwritten, according to time stamp ordering)
    * the write(Q) will be **rejected**, and Ti will be aborted, rolled back, and restarted with a new (larger) time stamp;

  * else it IS safe to execute write(Q), so:
    * the write(Q) is executed, and
    * W-ts(Q) is updated to be TS(Ti)

Note that there is more overhead for this approach, the overhead for these timestamps! Each item modified needs this R-ts and W-ts stored for it. But while livelock/starvation could occur, deadlock is not an issue here, and it is an interesting alternative to locking algorithms for concurrency control.

Optimistic methods

This category of algorithms for concurrency control is based on the **assumption** that the majority of database operations **do not conflict**. These algorithms do not require locking or time-stamping -- instead, a transaction is executed without restrictions until it is committed. That is, the transaction moves through three phases:the **read** phase, the **validation** phase,and the **write** phase:

* **read** phase: the transaction reads the database, executes the needed computations, and makes the

updates to a **private** copy of the database values;
  * **all** updates are recorded in a temporary update file, accessible only by that one transaction (and **not** any others running simultaneously)

* **validation** phase: the transaction is validated to assure that the changes made will not affect the database's integrity or consistency;
  * if the validation test succeeds? The transaction goes to the write phase.
  * if it fails? The transaction is **restarted**, and its changes **discarded** (they were made in a temporary update file, remember, and not to the actual database)

* **write** phase: the changes are permanently applied to the database.

This approach is acceptable for mostly-read or mostly-query database systems that require very few update transactions.

We could take this discussion of concurrency control options much further, but this is where we will stop. Hopefully it has given you an idea of some of the different means of providing concurrency control and some of the issues involved.