**COMP 418 TME3**

**Jason Bell 3078931**

**April 2, 2016**

**Part 1**

1. **Define ACID properties, and briefly explain the methods used to deal with them in a DBMS.**

ACID stands for atomic, consistent, isolation, and durable.

Atomic: transactions should be (or should appear to be) all or nothing: transactions are either carried out or not, including in the case of system failure. This is ensured by logging transactions: upon restart of the DBMS, those that transactions that were interrupted by a failure are undone and reapplied.

Consistency is ensuring that values in the data base are “as expected”. For example, that a bank balance is correct after a deposit, or that an employee has the correct list of dependents in his or her record. These are issues that a DBMS can’t be held accountable for, as it requires contextual knowledge. Therefore, it is the responsibility of the user to ensure consistency.

Isolation is guaranteeing that two or more transactions leave the database in the same state regardless of the order in which they are carried out. This is ensured through locks, so that only one transaction can alter a table at a time. However, this ties into consistency: isolation guarantees that if two transactions add and remove money from a bank balance, the end result will be the same regardless of which is run first. Conversely, if the user executes two transactions that write different names to the same employee record, either name could be seen in the record afterwards. This is a user error.

Durability ties into atomic: it is ensuring that if a transaction is interrupted by a failure, it will be completed when the DBMS restarts. As described for Atomic, this is done by logging the transaction: after it is undone, it is then reapplied.

1. **Why is concurrent execution of transactions important in a DBMS? Define the anomalies that might happen if concurrent execution of conflicting actions is not handled properly.**

Concurrent execution of DBMS transactions is important for performance, which will suffer if all transactions are queued regardless of whether they write to the same table. If two transactions do not write to the same database objects, they may as well be run concurrently.

The anomalies are the same as those seen with threading:

If too many transactions are allowed to run concurrently, “thrashing” occurs: too many compete for memory and CPU time, resulting in a loss in performance. There is a balance where you don’t want too many transactions to run concurrently. How many can run efficiently at once depends on how many CPUs the hardware has.

If there is not a proper locking mechanism to prevent two transactions from writing to the same object at the same time, race conditions can occur. The transactions may overwrite each other, with unpredictable results. Perhaps, by chance, all will be well, or perhaps their data will be interleaved, or perhaps the data will be corrupted.

Finally, deadlocks can happen, just as in threading: one or more transactions stall because they are waiting forever for a lock on a resource to be released. Perhaps two transactions are deadlocked because each holds a lock on a resource that the other needs. Perhaps the last transaction didn’t properly release the lock on the resource. Whatever the cause, the DBMS needs to be able to detect with DAGs when deadlocks may occur, or at the very least detect when a deadlock occurs and reset the transaction.

1. **Explain how timestamps are used for concurrency control and deadlock prevention. Also explain how does the Thomas write rule improve concurrency?**

Timestamping ensures that transactions are executed in the order in which they were created: when a DBMS receives a transaction, it records the time. Database objects also record the timestamps of the last transaction to read from it or write to it.

If a read transaction has a timestamp that is earlier than the write timestamp recorded on a database object, that is being written to by another transaction some time after the read transaction was created (perhaps the read transaction was delayed along the way). The read transaction is restarted with a new timestamp so that it does not conflict with the write. If the read transactions timestamp is newer than the object’s write timestamp, it is safe to read. The object’s read timestamp is updated to the transaction’s read timestamp, if it is newer.

If a write transaction has an older timestamp than the read timestamp of the object, it is restarted with a new timestamp so that it does not conflict with the read. If the transaction’s timestamp is younger than the object’s write timestamp, it is safe to write and the object’s write timestamp is updated.

The Thomas write rule covers the case where the write transaction’s timestamp is older than the object’s write timestamp. This means that a more recent transaction has already written to the object, and would have overwritten the current transaction anyways. Therefore, the current transaction need not complete, since it’s results would not have been observed anyways.

1. **Compare deadlock detection schemes and deadlock prevention schemes, and explain why detection schemes are more commonly used.**

Deadlock prevention only allows algorithms to proceed in a way that ensures that deadlock won’t happen. This could be done through the use of timestamps, conservative 2PL, or by analyzing the transactions before they are carried out. However, these processes add overhead, while deadlocks are quite rare in practice.

Deadlock detection detects deadlocks when they happen, typically by either detecting when transactions are cycling (such as two transactions that are waiting for a lock held by the other) or detecting timeouts. In either case, one of the deadlocked transactions is aborted and restarted, hopefully allowing other transactions to proceed. This is more commonly used because of the rarity of deadlocks: the cost of occasionally restarting transactions is less overall than the cost of preventing deadlocks.

1. **Briefly explain the WAL protocol.**

The write-ahead protocol is the requirement that before modifications caused by a transaction are written to disk, a log of those changes are also written to stable storage in case of failure. This ensures that if the DBMS does experience a failure and must restart, there is a log of transactions that may not have completed, and which can be used to recreate and reapply those transactions.

**Part 2**

1. **One morning the chief information officer (CIO) called an emergency meeting of the system administration team. As a database administrator, you are part of this team. The meeting was about a data inconsistency problem in a newly launched web application. The database in question deals with items and their prices. Your colleague in charge of the problematic database noticed that the phantom problem is happening while the application is following the strict 2PL protocol. He explained that while a first transaction (T1) has locked all existing items of type 1, a second transaction (T2) was able to add new items of type 1 at the same time, which caused the average price calculated by T1 to be incorrect when displayed to the user.**

**The CIO asked the rest of you to check whether the same problem is happening in the databases you are in charge of.**

**Can this Phantom problem occur in any of your databases where the set of database objects is fixed and only the values of objects can be changed? Explain.**

This question is somewhat vague and requires an assumption: that “only the values of objects” can be changed means that transactions can’t add new entries to tables, only modify existing ones.

If this assumption is true, then this phantom problem will not happen with my databases. As long as a row is locked, other transactions must wait for it. Using the example that the CIO gave: a transaction that calculates the average of all rows must wait until any rows being modified are released. The averaging transaction then locks all the rows, and they can’t be modified until the lock is released. The CIO is seeing the phantom problem with the faulty database because it is able to add new entries: the averaging transaction locks existing rows, but then while it is running another transaction adds a new, unlocked, row that the averaging transaction is unaware of. The result is the calculated and actual averages being different. As my databases do not add rows, this won’t happen.

If my database objects can have new entries added, then the phantom problem can potentially occur. If so, one way of preventing this problem is adding a requirement that the averaging transaction lock the entire table, such that other transactions are unable to add new rows until it completes.

1. **As a database administrator of a large manufacturing organization, you are in charge of their website database that handles suppliers, parts, catalog, customers, and payments of delivered items. The following relations are part of the large database:**

**Supplier(SID, Sname, Saddress, Stelephone)  
Customer(CID, Cname, Caddress, Ctelephone)  
Part(PID, Pname, Pmodel, Pcolor)  
Catalog(SID, PID, Price)**

**(The Catalog relation lists the prices charged for parts by Suppliers).**

**Because the application is web-based and the database is accessed by many users, you want to increase the concurrency usage without compromising the data quality. For each of the following transactions, state the minimum SQL isolation level you would use in order to avoid any conflict problems in them. Please explain your choice.**

1. **A transaction that adds a new part to a supplier’s catalog.**

Assuming that the part already exists in the Part table, and the supplier knows the PID, the transaction can be carried out through a view on the Catalog table that provides access to the (PID, Price) columns. This view should be available to suppliers, but not customers. The supplier does not need to know the SID if it is already handled by this view. That is, each supplier has it’s own view of the Catalog table, each using the SID of that supplier when a new entry is added. When the supplier wants to add a part to the catalog, they provide it’s PID and Price, and the view provides the SID. Because the supplier can’t see or use the SID, there is no danger of using the incorrect SID and potentially affecting other supplier’s catalogs.

1. **A transaction that increases the price that a supplier charges for a part.**

The level of isolation provided by the previous transaction will work well for this one as well. To change the price, the supplier needs only provide the PID of the part and the new price. As with the previous question, using this view means that there is no risk of the supplier affecting other catalogs.

1. **A transaction that determines the total number of items for a given supplier.**

In answering this question, I am assuming that this is a query that returns the part count in the catalog for any supplier. Therefore, we need a new transaction, since we need to be able to select on SID: the SID is necessary to find the catalog for that supplier, otherwise we may as well keep it hidden and use the supplier name. Whether or not suppliers and customers have access to this level of isolation would depend on whether it is desirable for them to get the counts of other supplier’s part catalogs or to learn the SIDs of other suppliers. In any case, the view that provides this level of isolation should be read-only.

As long as the view is read-only, we can safely provide a view on (SID, PID) of the Catalog table. This will allow us to count the number of unique PIDs for a given SID while preventing the alteration of either. Again, this is probably a view that is best inaccessible to suppliers and customers if they are not to know SIDs, or could safely be available since they can’t alter the SID. This is likely a managerial decision to be made by the business providing this database.

1. **A transaction that shows, for each part, the supplier who has the part at the lowest price.**

This is another level of isolation that needs to be read-only, since no writes are required. A read-only view on all columns of the Catalog table will accomplish it, though may be non-optimal. This transaction can be carried out by obtaining the list of unique PIDs found in the catalog table, then returning the SID with the lowest price for each unique PID. However, the cost of getting the list of unique PIDs may add unnecessary overhead when there is already a list of unique PIDs found in the Parts table.

Therefore, we could also provide read-only access to the (PID) column of the Parts table. This will require a join of the two tables on PID, which may be better or worse performance depending on the sizes of the tables.

1. **A new application is being developed and will be using a database that includes a relation about items: Item (item\_id:integer, item\_name:string, color:string, price:real).**

**Both the purchasing department in charge of obtaining raw material and the manufacturing department in charge of manufacturing the items can change the price of manufactured items according to changes that may happen in the raw material cost or production cost. The two departments use different transactions to update the price of items. The new application uses the following sequences of actions, listed in the order they are submitted to the DBMS:**

**Sequence S1: T1:R(X), T2:W(X), T2:W(Y), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit**

**Sequence S2: T1:R(X), T2:W(Y), T2:W(X), T3:W(Y), T1:W(Y), T1:Commit, T2:Commit, T3:Commit**

**For each of the following concurrency control mechanisms, describe how they will handle each of the sequences (S1 & S2).**

1. **Strict 2PL with timestamps used for deadlock prevention.**
2. **Conservative (and Strict, i.e., with locks held until end-of-transaction) 2PL.**

**Note: Assume that the DBMS processes actions in the order shown. If a transaction is blocked, assume that all its actions are queued until it is resumed; the DBMS continues with the next action (according to the listed sequence) of an unblocked transaction. Write any other assumption you are using for your answer.**

I’ll answer b first since it’s the simpler case. Conservative 2PL obtains all needed locks at the start of the transaction and doesn’t release them until the end. If a transaction only reads, it requests a shared lock that permits other reads but not writes. If a transaction writes, it requests an exclusive lock that prevents other transactions from reading or writing.

For example, for sequence 1, even though T1 only needs a shared lock on X, T2 must wait for T1 to complete since it needs an exclusive lock to write. Since there are not instances where two or more transactions only read, and all locks must be obtained before starting, the result is that the transactions happen serially:

Sequence 1 with Strict 2PL:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| Shared(X) |  |  |
| Exclusive(Y) |  |  |
| Read(X) |  |  |
| Write(Y) |  |  |
| Commit |  |  |
|  | Exclusive(X) |  |
|  | Exclusive(Y) |  |
|  | Write(X) |  |
|  | Write(Y) |  |
|  | Commit |  |
|  |  | X(Y) |
|  |  | W(Y) |
|  |  | Commit |

Sequence 2 with Strict 2PL serializes the same way except for the order of the writes performed by T2:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| Shared(X) |  |  |
| Exclusive(Y) |  |  |
| Read(X) |  |  |
| Write(Y) |  |  |
| Commit |  |  |
|  | Exclusive(X) |  |
|  | Exclusive(Y) |  |
|  | Write(Y) |  |
|  | Write(X) |  |
|  | Commit |  |
|  |  | X(Y) |
|  |  | W(Y) |
|  |  | Commit |

As we can see, deadlocking can’t happen as no transactions are ever stuck waiting for a lock while already holding a lock.

For a: timestamps allow the following rules:

If the transaction timestamp is greater (newer) than the object’s write timestamp, it is safe to read, otherwise block with a new timestamp.

If the transaction timestamp is greater (newer) than the object’s read timestamp, it is safe to write, otherwise block with a new timestamp.

If the transaction timestamp is less than (older) than the object’s write timestamp, the transaction can be ignored since it would have been overwritten anyways (Thomas’ rule).

Strict but non-conservative 2PL still requires that locks be held until the end of the transaction, but obtaining the lock can be delayed until it is needed: we don’t need to obtain all locks before starting.

Sequence 1 before applying timestamps or Thomas’ rule (note that delaying obtaining locks allows parallelization: T3 can obtain a lock on Y before T1 does, though now T1 may have to wait for T3 to release the lock on Y):

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| Shared(X) |  | -Exclusive(Y) |
| Read(X) |  | -Write(Y) |
| Exclusive(Y) |  | -Commit |
| Write(Y) |  |  |
| Commit |  |  |
|  | Exclusive(X) |  |
|  | Write(X) |  |
|  | -Exclusive(Y) |  |
|  | -Write(Y) |  |
|  | Commit |  |

No deadlock happens in the above because T1 is processed first, and T2 can’t start until T2 commits and releases it’s shared lock on X. When applying Thomas’ rule, and timestamps we can also drop any writes that are overwritten by writes submitted afterwards in the sequence and that don’t affect reads: the writes denoted with a negative sign in the above table. Note that this drops T3 entirely:

|  |  |
| --- | --- |
| T1 | T2 |
| Shared(X) |  |
| Read(X) |  |
| Exclusive(Y) |  |
| Write(Y) |  |
| Commit |  |
|  | Exclusive(X) |
|  | Write(X) |
|  | Commit |

Sequence 2 demonstrates deadlocking (shown by the gray area) that may occur if timestamps (shown as the transaction’s position in the sequence) and Thomas’ rule aren’t considered:

|  |  |  |
| --- | --- | --- |
| T1 | T2 | T3 |
| Shared(X) | Exclusive(Y) |  |
| Read(X) (1) | Write(Y) (2) |  |
| Exclusive(Y) | Exclusive(X) |  |
| Write(Y) (5) | Write(X) (3) |  |
| Commit | Commit |  |
|  |  | Exclusive(Y) |
|  |  | Write(Y) (4) |
|  |  | Commit |

Applying timestamps and Thomas’ rule resolves this and results in the same sequences as was the case with sequence 1:

|  |  |
| --- | --- |
| T1 | T2 |
| Shared(X) |  |
| Read(X) |  |
| Exclusive(Y) |  |
| Write(Y) |  |
| Commit |  |
|  | Exclusive(X) |
|  | Write(X) |
|  | Commit |

5. **Using the following execution shown below, explain what is done in each of the ARIES recovery algorithm phases:**

|  |  |  |
| --- | --- | --- |
| **LSN** |  | **LOG** |
| 00 |  | begin\_checkpoint |
| 10 | end\_checkpoint |
| 20 | update: T1 writes P1 |
| 30 | update: T2 writes P2 |
| 40 | update: T3 writes P3 |
| 50 | T2 commit |
| 60 | update: T3 writes P2 |
| 70 | T2 end |
| 80 | update: T1 writes P5 |
| 90 | T3 abort |
|  | CRASH, RESTART |
|  |  |  |

**In addition to the execution shown here, the system crashes during recovery after writing two log records to stable storage and again after writing another two log records.**

ARIES will first identify dirty pages and interrupted transactions in the analysis phase. We can see that T2 commits before the crash, so it’s results have been written to disk. T1 is interrupted, and T3 aborted, so both need to be handled. All five pages were updated after the checkpoint, so these pages are dirty. We start from the most recent checkpoint, which holds the transaction and dirty page logs at that time.

During the recovery phase, the transactions in the dirty pages are reapplied in the order of their LSN. This includes T2, even though it committed: it must be reapplied to ensure consistent state: the user believes that it committed, so expects it to still be committed after a crash. So after the checkpoint, all actions before T3 aborting at LSN 90 are reapplied (ie, LSN 20 through 80 are reapplied).

During the undo phase, transactions that were aborted or active at the time of the crash need to be undone: T1 and T3. These are undone by working backwards through the log and reversing each logged update for those transactions. So, updates are undone for LNS 80, 60, 40, and 20, in that order.

If a crash occurs during recovery, it is fortunate that thanks to the WAL protocol there is a record of the recovery updates that were underway during the crash. Thus, although the recovery did not complete, there is a record of what stage it was at when the database crashed again, and we can start the recovery process over as before, except with additional actions to undo.