CS580 ONLY Project # 4 Fall 2021

Group Project

|  |  |
| --- | --- |
| Student Names | Antonio Zea |
| Section | CS 580 |

For this project you will be researching transaction processing and concurrency control techniques (Ch. 21-22 in both 6th & 7th edition of textbook). This project consists of both a small research summary of the issues of this topic as well as completing problems related to this topic (below). This is a group project, but be sure all know how to do the problems below as understanding of them will be on the final exam.

SUBMIT:

One word document called P4-LastNamesOfGroupMembers.

In the document include names of students in the group and provide the following:

* A 5 page write up summary of the issues and challenges of transactions and concurrency; and the methods used to handle these challenges. This write up should include both what are the theoretical challenges, but also a description of how such a scenario could occur in a real application.
* The answers to the problems below:

|  |
| --- |
| STUDENT NAME(s): |
| 1. Construct the serializability (or precedence) graph for the schedule specified bellow. Determine if the following schedule is (conflict) serializable. If it is, specify equivalent serial schedule(s).   r2(X); r3(X); w2(X); r1(X); w3(X)    Not conflict serializable |
| 1. Construct the serializability (or precedence) graph for the schedule specified bellow. Determine if the following schedule is (conflict) serializable. If it is, specify equivalent serial schedule(s).   r1(X); r2(X); w2(X); w1(X); r3(X)    Not conflict serializable |
| 1. Construct the serializability (or precedence) graph for the schedule specified bellow. Determine if the following schedule is (conflict) serializable. If it is, specify equivalent serial schedule(s).   r3(X); r2(X); w3(X); r1(X); w1(X), w2(X)    Not conflict serializable |
| 1. Construct the serializability (or precedence) graph for the schedule specified bellow. Determine if the following schedule is (conflict) serializable. If it is, specify equivalent serial schedule(s).   r3(X); r2(X); w3(X); w1(X)    Conflict Serializable  r2(x);r3(x);w3(x);w(1) |
| 1. Consider the three transactions T1, T2, and T3, and the schedule specified bellow. Construct the serializability (precedence) graph. Determine if the schedule is (conflict) serializable. If it is, specify equivalent serial schedule(s).   T1: r1(X); r1(Z); w1(X);  T2: r2(Z); r2(Y); w2(Z); w2(Y);  T3: r3(X); r3(Y); w3(Y);  SCHEDULE: r1(X); r2(Z); r1(Z); r3(X); r3(Y); w1(X); w3(Y); r2(Y); w2(Z); w2(Y);    Conflict Serializable  r3(x);r3(y);w3(y);r1(x);r1(z);w1(x);r2(z);r2(y);w2(z);w2(y);  r3(x);r3(y);w3(y);r1(x);r1(z);r2(z);r2(y);w2(z);w1(x);w2(y); |

Transactions allow a database to handle more than one user or process. Handling transactions naturally leads to the handling of concurrency of those transactions. I will seek to summarize the issues and challenges of transactions and concurrency and the methods used to handle these challenges.

A database system may be classified according to the number of users who can use the system concurrently. A Database Management System(DBMS) is single-user if at most one user at a time can use the system. It is multiuser if many users can use the system concurrently. Single-user DBMS are generally used in small applications that reside on personal computers (i.e., TinyDB in mobile apps). Multiuser databases accomplish this functionality by using interleaved concurrency which means while one operation is working, the other is waiting its turn to work on the database.

A transaction is an executing program that forms a logical unit of database processing. It includes one or more database access operations which can include insertion, deletion, modification or retrieval operations.

One way of specifying the transaction boundaries is by specifying explicit begin-transaction and end-transaction statements in an application program. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called read-only. Otherwise, it is known as a read-write transaction.

Several problems can occur when concurrent transactions execute in an uncontrolled manner. The Lost Update Problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. One example of this problem would occur with bank accounts. Fir example, one transaction(T1) reads the original amount, takes it and adds $100. Before writing that value, another transaction(T2) reads the original amount, takes it and subtracts $50. At this point, T1 writes: original balance + $100. After this, T2 writes the account balance: original balance - $50. The resulting balance in the account will be lower than it is supposed to be by $100 as the $100 deposit is lost.

The Temporary Update (or Dirty Read) problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the update item is accessed(read) by another transaction before it is changed back to its original value. In this problem, imagine the $100 dollar deposit(T1) fails at some point. While T1 is processing, another transaction(T2) reads the account balance including the deposited $100. T2 will have an incorrect balance of the account since T1 fails and will be rolled back to its orginal value. T2 should not have been allowed to read the account balance until T1 completed or was rolled back.

The Incorrect Summary problem occurs when one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items. The aggregate function may calculate some values before they are updated and others after they are updated. If a summary of one’s bank accounts are being retreived and transactions are still completing in the midst of the construction of the summary, the summary will not be accurate.

The Unrepeatable Read problem occurs when a transaction(T) reads the same item twice and the item is changed by another transaction(T’) between the two reads. Hence, T receives different values for the two reads of the same item. For example, a transaction reads the balance to check if it is possible to make a withdrawal. Once it is confirmed that a withdrawal is possible, it reads the balance again to actually perform the withdrawal; however, there is now a different value than the one originally read.

Transactions should possess several properties, often called ACID properties: atomicity, consistency preservation, isolation, durability or permanency. A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all. This will be enforced by a transaction recovery subsystem. If a transaction fails to complete, then the subsystem must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be written to disk.

A transaction should be consistency preserving. This means that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another. This usually is the responsibility of the programmers who write the database programs or of the DBMS module enforcing integrity constraints.

A transaction should appear as though it is being executed in isolation from other transactions even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently. This is enforced by the concurrency control subsystem.

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure. This responsibility lies with the recovery subsystem of the DBMS.

The order in which various transactions execute is known as the schedule or history. Schedules can be characterized in terms of their recoverability. A shorthand notation for describing a schedule uses the symbols b, r, w, e, c, and a for the operations begin\_transaction, read\_item, write\_item, end\_transaction, commit, and abort, respectively. It appends as a subscript, the transaction id (transaction number) to each operation in the schedule.

Two operations in a schedule are in conflict if the belong to different transactions, they access the same item X, and at least one of the operations is a write\_item(X). Schedules can be characterized as recoverable, cascadeless and strict. There is much overlap between all of these destinctions. Strict schedules are also cascadeless and cascadeles schedules are recoverable.

A schedule is serial if for every transaction(T) participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called nonserial. A schedule(S) of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions. A serial schedule is considered correct because every transaction is assumed to be correct if executed on its own. A serializable schedule is considered correct because it is equivalent to a serial schedule, which is considered correct.

Being serializable is distinct from being serial, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness.

A binary lock enforces mutual exclusion on the data item. This system ends up being too restrictive. If every transaction is seeking to read the locked item, then there is insufficient reason to lock the item since read operations from different transactions cannot conflict. Shared/Exclusive locks are favored as they allow any number of transactions to read the share-locked item while an exclusive-locked item is only accessible by one transaction that holds its lock.

The two-phase locking (2PL) protocol is followed if all locking operations (read\_lock, write\_lock) precede the first unlock operation in the transaction. If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Conservative 2PL requires a transaction to lock all the items it accesses before the transaction begins execution. If any of the items needed cannot be locked, the transaction does not lock any item and instead waits until all items are available for locking. In strict 2PL, a transaction does not release any of its exclusive locks until after it commits or aborts. This leads to a strict schedule for recoverability although it is not deadlock-free.

Deadlock occurs when each transaction in a set of two or more transactions is waiting for some item that is locked by some other transaction in the set. One way to prevent deadlock is to use a deadlock prevention protocol such as wait-die or wound-wait. Although both techniques are deadlock-free both may cause some transactions to abort needlessly.

Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. A fair waiting scheme such as FIFO can be used to mitigate starvation. Transactions are enabled to lock an item in the order in which they originally requested the lock.

Wait-die prioritizes younger transactions and allows the older transaction to wait until the resource is available for execution. Wound-wait priotizies older transactions and allows the younger transaction to wait until the resource is available for execution.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur. However, this scheme can cause transactions to abort and restart needlessly.

The cautious waiting algorithm was proposed to try and reduce the number of needless aborts/restarts. Suppose that a transaction (Ti) tries to lock an item (X) but is not able to do so because it is locked by some other transaction Tj. If Tj is not blocked, then Ti is blocked and allowed to wait, otherwise abort Ti.

The use of timeouts is practical because of the low overhead and simplicity. If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

Timestamp is a unique identifier created by the DBMS to identify a transaction by the order in which it was submitted to the system. One method for generating timestamps is to use a counter that is incremented each time its value is assigned to a transaction. Depending on the maximum size of the counter, the system will need to reset this counter to zero when no transactions are executing. Another way to implement timestamps is to use the current date/time value from the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

In Timestamp Ordering protocol, transactions are ordered using timestamps. Each item contains a read\_TS, the youngest transaction to have read this item and a write\_TS, the youngest transaction to have written to this item. With this information we can determine if transaction should be allowed to proceed. A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamps. The algorithm must ensure that, for each item accessed by conflicting operations, the order in which the item is accessed does not violate the timestamp order.

Basic Timestamp Ordering (basic TO) checks that when a transaction tires to issue a read\_item or write\_item operation, the timestamps of read\_TS or write\_TS do not violate the order of transaction execution. If this order is violated then T is aborted and resubmitted to the system as a new transaction with a new timestamp. If the transaction is aborted and rolled back, a cascading rollback may occur. This is one of the problems with basic TO.

In basic TO, if the last successful read of the item is older than the timestamp of the transaction or if the timestamp of the transaction is older than the last successful write of the item, then the transaction is aborted, rolled back and rejected. This means that a younger transaction has already read or written the value of the item and this current transaction will violate timestamp ordering.

Lock based protocols for concurrency control and transaction serialization and its use in verifying concurrency control protocols are major takeaways from this research. Understanding of these concepts is key to understanding how large multiuser databases function.