HOMEWORK #8

CINS 370 Introduction to Databases
Chapter 20 Transaction Processing
Viraj Sonavane

What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed and give informal examples.

Concurrent execution of database transactions in a multi-user system is where any number of users can use the same database at the same time. Concurrency control is needed in order to avoid inconsistencies in the database.

The Lost Update Problem**:** This 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. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved; then the final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 is lost. For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T_2 reserves 4 seats on X), the final result should be X = 79. However, in the interleaving of operations, it is X = 84 because the update in T_1 that removed the five seats from X was lost.

The Temporary Update (or Dirty Read) Problem: This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value. When T_1 updates item X and then fails before completion, so the system must roll back X to its original value. Before it can do so, however, transaction T_2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T_1. The value of item X that is read by T_2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.

The Incorrect Summary Problem: If 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. For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations occurs, the result of T_3 will be off by an amount N because T_3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.

The Unrepeatable Read Problem: Another problem that may occur is called unrepeatable read, where 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 its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquiry about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Discuss the different types of failures. What is meant by catastrophic failure?

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

A computer failure (system crash)**:** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

A transaction or system error**:** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. Additionally, the user may interrupt the trans- action during its execution.

Local errors or exception conditions detected by the transaction**:** During transaction execution, certain conditions may occur that necessitate cancelation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

Concurrency control enforcement**:** The concurrency control method may abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.

Disk failure**:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

Physical problems and catastrophes**:** This refer to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Catastrophic failure:

Catastrophic failure will occur very rarely. Catastrophic failure includes many forms of physical misfortune to our database and there is an endless list of such problems.

- The hard drive with all data may completely damage
- Fire accident that may cause the loss of physical devices and data loss.
- Power or air-conditioning failures.
- Destruction of physical devices.
- Theft of storage media and physical devices
- Overwriting disks or tapes by mistake.

Discuss the actions taken by the read_item and write_item operations on a database.

The basic database access operations that a transaction can include are as follows:

- **read_item(*X*).** Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X.*
- **write_item(*X*).** Writes the value of program variable *X* into the database item named *X*.

  Executing a read_item(*X*) command includes the following steps:

  - Find the address of the disk block that contains item *X*.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
  - Copy item *X* from the buffer to the program variable named *X*.

  Executing a write_item(*X*) command includes the following steps:

  - Find the address of the disk block that contains item *X*.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item *X* from the program variable named *X* into its correct location in the buffer.
  - Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

What is the system log used for? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important?

To be able to recover from failures that affect transactions, the system maintains a **system log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.

The following are the types of entries called **log records** that are written to the log file and the corresponding action for each log record. In these entries, T refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

- **[Start_transaction, T]:** Indicates that transaction T has started execution.
- **[Write_item, T, X, old_value, new_value]:** Indicates that transaction T has changed the value of database item X from old_value to new_value.
- **[Read_item, T, X]:** Indicates that transaction T has read the value of database item X.
- **[Commit, T]:** Indicates that transaction T has completed successfully and affirms that its effect can be committed (recorded permanently) to the database.
- **[Abort, T]:** Indicates that transaction T has been aborted.

A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully and the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be permanently recorded in the database. The transaction then writes a commit record [commit, T] into the log. If a system failure occurs, we can search back in the log for all transactions T that have written a [start_transaction, T] record into the log but have not written their [commit, T] record yet; these transactions may have to be rolled back to undo their effect on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be redone from the log record

What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered correct?

The DBMS transactions T_1 and T_2 if no interleaving of operations is permitted, there are only two possible outcomes:

- Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence).
- Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

These two schedules called **serial schedules**. In a serial schedule, only one transaction at a time is active the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be independent, is that every serial schedule is considered correct.

Every **serial schedule** transaction is assumed to be correct if executed on its own (according to the consistency preservation property). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result.

A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions. We will define the concept of equivalence of schedules shortly. **Serializable schedule** is correct, because it is equivalent to a serial schedule.

Problem 1: Given the table (below) for transactions T1 and T2, list all the possible schedules; And determine which are conflict serializable (correct) and which are not.

T1: r1(X); w1(X); r1(Y);
T2: r2(X); w2(X);

Number of operations in T1: 3
Number of operations in T2: 2

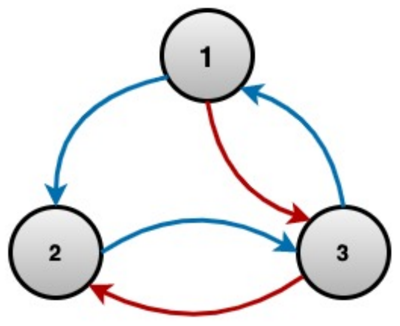
Total number of schedules = (3 + 2)! / (3! * 2!) = 10

Total 10 different possible schedules and their type are as follow:

S1: r1(X); w1(X); r1(Y); r2(X); w2(X); **(conflict serializable)**
S2: r1(X); w1(X); r2(X); r1(Y); w2(X); **(conflict serializable)**
S3: r1(X); w1(X); r2(X); w2(X); r1(Y); **(conflict serializable)**
S4: r1(X); r2(X); w1(X); r1(Y); w2(X); **(Not Serializable)**
S5: r1(X); r2(X); w1(X); w2(X); r1(Y); **(Not Serializable)**
S6: r1(X); r2(X); w2(X); w1(X); r1(Y); **(Not Serializable)**
S7: r2(X); r1(X); w1(X); r1(Y); w2(X); **(Not Serializable)**
S8: r2(X); r1(X); w1(X); w2(X); r1(Y); **(Not Serializable)**
S9: r2(X); r1(X); w2(X); w1(X); r1(Y); **(Not Serializable)**
S10: r2(X); w2(X); r1(X); w1(X); r1(Y); **(conflict serializable)**

Problem 2:

a. r1(X); w3(X); w1(X); r2(X); w3(X);

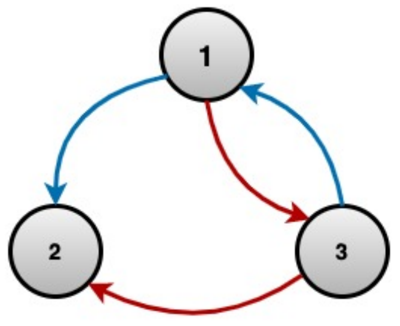


r1(X); w3(X);
r1(X); w3(X);
w3(X); w1(X);
w3(X); r2(X);
w1(X); r2(X);
w1(X); w3(X);
r2(X); w3(X);

Not Serializable

As there are **loops(cycle)** between **1 - 3** and **2 - 3** this schedule is **Not Serializable**.

b. r1(X); r3(X); w3(X); w1(X); r2(X);

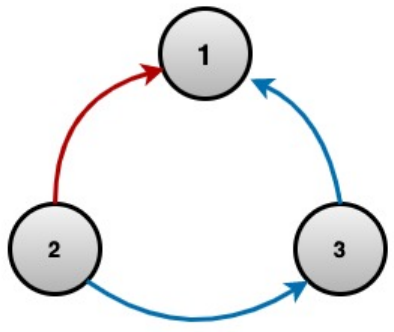


r1(X); w3(X);
r1(X); r2(X);
r3(X); w1(X);
w3(X); w1(X);
w3(X); r2(X);
w1(X); r2(X);

Not Serializable

As there is **loop(cycle)** between **1 - 3** this schedule is **Not Serializable.**

c. r3(X); r2(X); w3(X); r1(X); w1(X);



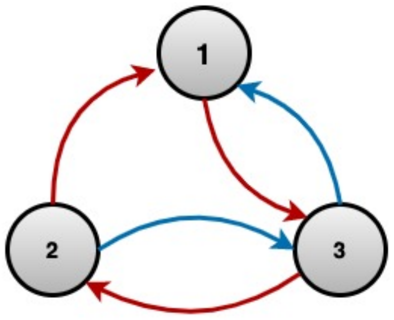
r3 (X); w1 (X);
r2 (X); w3 (X);
r2 (X); w1 (X);
w3 (X); r1 (X);
w3 (X); w1 (X);

Serializable

Equivalent Serial Schedule: T2->T3->T1

As there are **no loops(cycle)** between **1 - 3** or **2 - 3** or **1 - 2** this schedule is **Serializable.**

d. r3(X); w2(X); r1(X); w3(X); w1(X);



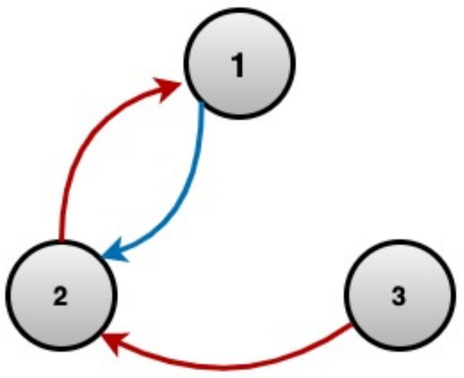
r3(X); w2(X);
r3(X); w1(X);
w2(X); r1(X);
w2(X); w3(X);
w2(X); w1(X);
r1(X); w3(X);
w3(X); w1(X);

Not Serializable

As there are **loops(cycle)** between **1 - 3** or **2 - 3** this schedule is **Not Serializable.**

Problem 3:

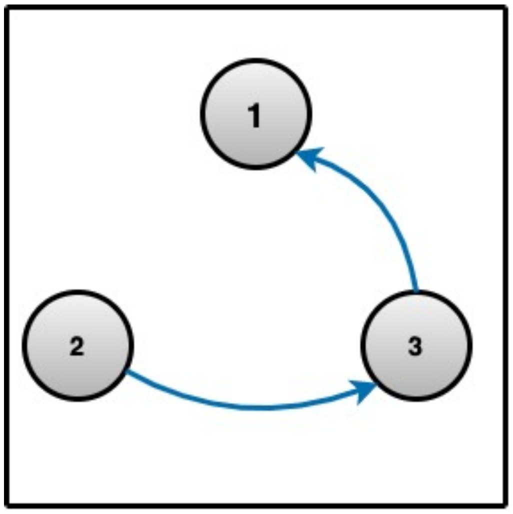
S1: w2 (Z); r1 (Z); r3 (X); r3 (Y); r2 (Z); w1 (Z); w3 (Y); r2 (Y); w2 (Z); r1 (X); w2 (Y);



w2(Z); r1(Z);
w2(Z); w1(Z);
r1(Z); w2(Z);
r3(Y); w2(Y);
r2(Z); w1(Z);
w1(Z); w2(Z);
w3(Y); r2(Y);
w3(Y); w2(Y);

As there is **loop(cycle)** between **1 - 2** this schedule S1 is **Not Serializable**

S2: r1 (X); r2 (Z); r3 (X); w2 (Y); w1 (X); r2 (Y); r3 (Y); w1 (X); w2 (Z); w3 (Y); r2 (Z);



r3(X); w1(X);
w2(Y); r3(Y);
w2(Y); w3(Y);
r2(Y); w3(Y);

As there are **no loops(cycle)** between **1 – 2 or 1 – 3 or 2 - 3** this schedule S2 is **Serializable.**

Equivalent serial Schedule: T2->T3->T1