12. Transaction Processing

Most of content here is borrowed from following sources-
>> Lecture slides of Jennifer Widom, Stanford (coursera)
>> Book Elmasri/Navathe

Serializability

Serializability is the mechanism for implementing Isolation property of ACID. In this approach, schedules that are executed should be *serializable*.

Serializable schedule is one that is equivalent to some *serial schedule*. So, understanding *serializable* schedule is basically getting answer of two questions- (1) What is serial schedule, and (2) how do we check equivalence?

Serial Schedule

Serial schedule is basically un-interleaved schedule. This means operations from different transactions are not interleaved.

For example, suppose there are two transactions T1 and T2 are executing concurrently; having operations (s1,s2,s3) and (s11, s12) respectively.

Serial execution schedule is either all of T1 is executed before any operation of T2 or all operations from T2 are executed before any operation from T1.

Let us also have a term "Order of Serial Schedule", and express as following

When all operations of T1 are execute any from T2, we express as <T1;T2>. This means operations from these transactions sequenced as: <s1, s2, s3, s11, s12>

When all operations of T2 are execute any from T1, we express as <T2;T1>. This means operations from these transactions sequenced as: < s11, s12, s1, s2, s3>

Serializable Schedule

Serializable schedule is an interleaved schedule that is equivalent to some serial schedule of participating transactions.

Let us take some examples and understand this.

Representation of Schedule

Consider following concurrent transactions T1 and T2.

	T1
	mx1=READ(X)
	mx1 := mx1 – N
	WRITE(X,mx1)
	my1=READ(Y)
	my1 := my1 + N
	WRITE(Y, my1)

T2
mx2=READ(X)
mx2 := mx2 + M
WRITE(X, mx2)

Let us express transactions T1 and T2 as following-

T1: mx1=r1(X); mx1=mx1-N; w1(X, mx1); my1=r1(Y); my1=my1+N; w1(Y,my1);

T2: mx2=r2(X); mx2=mx2+M; w2(X, mx2);

Let the number with r and w, represents transaction identification.

Serial Schedule S1:
mx1=r1(X); mx1=mx1-N; w1(X, mx1); my1=r1(Y); my1=my1+N; w1(Y,my1);
mx2=r2(X); mx2=mx2+M; w2(X,mx2);

All operations of T1 are scheduled before any of T2. That is execute T1 and then execute T2, and let that be expressed as T1;T2

Serial Schedule S2:
mx2=r2(X); mx2=mx2+M; w2(X, mx2); mx1=r1(X); mx1=mx1-N;
w1(X, mx1); my1=r1(Y); my1=my1+N; w1(Y,my1); ==> T2;T1;

Interleaved schedule-1 S3:

T1	mx1=READ(X)
T1	mx1 := mx1 – N
T2	mx2=READ(X)
T2	mx2 := mx2 + M
T1	WRITE(X,mx1)
T1	my1=READ(Y)
T1	my1 := my1 + N
T2	WRITE(X, mx2)
T1	WRITE(Y, my1)

In Compact form:

mx1=r1(X); mx1=mx1-N; mx2=r2(X); mx2=mx2+M; w1(X, mx1); my1=r1(Y);
my1=my1+N; w2(X, mx2); w1(Y,my1);

Serializability requires that this interleaved schedule should either be equivalent to S1 (T1;T2) or S2 (T2;T1).

Similarly, if there are three transactions concurrently executing, interleaved schedule should either be equivalent to T1;T2;T3, or T1;T3;T2, or T3;T2;T1, or T2;T1;T3, so forth.

So how do you compute to which order our schedule S3 is? T1;T2 or T2;T1?

Checking for Serializability

That is checking if an execution schedule is serializable!

Intuitionally, it is done as following: we attempt detecting the non serializability before including an operation in the execution schedule. If detected we do some correction.

Here is a popular technique for detecting the non-serializability:

Conflict Serializability

A common techniques used for checking if a schedule is Serializable "Conflict Serializability".

Intuition behind Conflict Serializability is that orders of all "Conflicting operations" from participating transactions are in same order.

What are conflicting operations?

It can be understood that concurrent access problem would likely to occur only when one of the operation in a transaction is write operations. With this intuition, conflicting operations are defined as following.

Two operations in a schedule are said to be conflicting, if they satisfy all of following conditions -

- Belong to two different transactions
- They access the same data items
- At-least one of them is write operations

Following are conflicting operations (that is if an data item is accessed by two transaction, and one of them is writing then it is a conflicting operation)-

- r1(X) and w2(X)
- w1(X) and w2(X)

Following are non-conflicting operations

- r1(X) and r2(X)   //both are read
- w1(X) and w2(Y)   //both work on different data item

By using this notion of conflicting operations, we define a serializability as following –

> If order of all conflicting operations in a schedule is same as in some serial schedule, then we can say that schedule in serializable. We call such serializability as "conflict serializability".

Now let us apply this check in schedule S3
S3: mx1=r1(X); mx1=mx1-N; mx2=r2(X); mx2=mx2+M; w1(X, mx1);
my1=r1(Y); my1=my1+N; w2(X, mx2); w1(Y,my1);
Following conflicting operations and there order:

(1) r1(X); w2(X)
(2) r2(X); w1(X)
(3) w1(X); w2(X)

Conflicting operation pair (1) and (3) are in the order of T1; T2 while operation pair (2) in the order of T2;T1. Hence they are not in same order and hence the schedule S3 is not serializable.

Testing Conflict Serializability

- A directed graph is built for checking if a schedule is conflict serializable.
- Let us call that *Precedence Graph* or Serialization Graph.
- It consists of n Nodes T1, T2, … Tn, and a set of edges E1, E2, … Em

- Here edges are such that an edge Ei, say is Tj → Tk, and this denotes that an operation appears in Tj before a conflicting operation in Tk.
- If there is any cycle is formed in the graph then, the schedule is not serializable.
- A cycle can be defined as: if starting and ending node for a sequence of edges is same, then graph forms a cycle.

Now let us create precedence graph for schedules S1 through S4, and confirm S1, S2, and S4 are serializable (actually S1 and S2 are serial) whereas S3 is not serializable.

Before proceeding, let us make our schedule representation further compact.

Note that only read/write operation matter in serializability test. Also, it does not matter what value is getting read and written; thus we express schedule S3 as following, which is otherwise expressed as given below.

S3 (in expanded form): mx1=r1(X); mx1=mx1-N; mx2=r2(X); mx2=mx2+M; w1(X, mx1); my1=r1(Y); my1=my1+N; w2(X, mx2); w1(Y,my1);

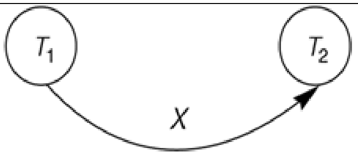
S3 (in compact form): S3: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);

Precedence graph for our S1 through S4

S1: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X);

Conflicting operations (T1;T2)

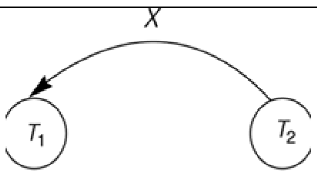
(1) r1(X); w2(X)
(2) w1(X); r2(X)
(3) w1(X); w2(X)



S2: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);

Conflicting operations (T2;T1)

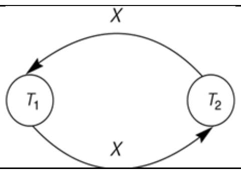
(1) w2(X); r1(X)
(2) r2(X); w1(X);
(3) w2(X); w1(X)



S3: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);

Conflicting operations (not in a consistent order)

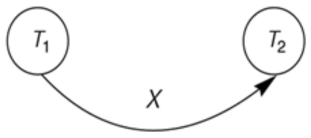
(1) r1(X); w2(X)
(2) r2(X); w1(X)
(3) w1(X); w2(X)



S4: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Conflicting operations (T1;T2)

(1) r1(X); w2(X)
(2) w1(X); r2(X)
(3) w1(X); w2(X)



Algorithm for Testing Conflict Serializability

- For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
- For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create edger Ti → Tj
- For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create edger Ti → Tj
- For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create edger Ti → Tj
- The schedule S is serializable only if, and <u>only if the precedence graph has no cycle</u>.

How Serializability can be ensured

By differing operations from transactions

With this understanding of serializability, let us look at our problem cases 1 through 4, and see how they can be handled, so that problems do not occur!

Case-1 and Case-2; having LOST UPDATE problem and go as following

T1 read tuple t
T2 reads tuple t
T1 write t after necessary modification
T2 writes t after necessary modification

Solution: <u>if we defer T2's read, and allow T1 write before T2 can read the tuple</u>.

Case-4:

- If we ensure conflicting operations in one of other T1; T2 or T2; T1; then there would not be any problem.
- That is - T2 either reads all tuples before T1's update (equivalent to T2;T1); or reads all tuples after T1's update (equivalent to T1;T2)
- That is Query in T2 will either include either all salary old or all new!

<u>Note: Serializability does not guarantee order of equivalence, it just guarantees, serializability.</u>

<u>ISOLATION property of ACID is to ensure execution of SERIALIZABLE schedule.</u>

<u>Why Serializability is important?</u>

- We cannot afford to run serial schedule.
- We need to interleave operations from simultaneous transactions.
- We interleave, but interleave such that they are equivalent to some serial schedule.
- We call such schedule as serializable.

How serializability is ensured?

Serializability is basically ensured by ordering the operations such that DBMS executes schedules in serializable manner.

DBMS module responsible for this is called "Concurrency Control" module. It implements certain concurrency control techniques. Even though it is heavily influenced by appropriate concurrency control techniques, essence of its typical working is as following -

- DBMS ready accepting operation requests (say read and write) from various transactions/clients, and put them in schedule. [Assume that once an operation has been put in a schedule, it is assumed as executed unless it is aborted (rolled back)].
- Before putting an operation in schedule DBMS ensures that resultant schedule will be serializable. There are various techniques for this (Locking, Time Stamp Ordering, etc.)
- If DBMS detects that if an operation under consideration leads to non-serializability, DBMS does one of following (and that depends on what algorithm it is based on, and so) –
    - Ask concerned transaction to wait; transaction goes in wait mode. DBMS definitely will notify the waiting transactions when it can accept the request
    - Ask requesting transaction to abort, because it may not possible that request to be executed as per serialization requirements.

Recoverability

Recoverability – systems ability to recover from crashes.

Crashes may leave database –

- Committed transaction not physically saved on disks, or
- Updates of uncommitted transactions (or partially done transactions) are physically saved on disks

Both are undesirable; therefore when system restarts, it should ensure-

- Updates of committed transaction reflected on database on disks
- Updates of uncommitted is not reflected on disks

To simplify recovery process there are certain basic conditions ensured while generating schedules.

Recovery Principle

- If we follow a discipline that a transaction T cannot commit until all transactions from which T reads (reads value written by other concurrent transactions), are committed.

- How does this help -

    – Suppose T2 reads from writes of T1, and T1 aborts; T2 also has to abort;

    – More serious if T2 is committed;

    – More serious if T3 reads from writes of T2; if T1 aborts; T2 has to abort, and hence T3 as well; leads to a "**cascaded roll-backs**"

- With assumption of “recovery principle”, we limit cascaded aborts only up-to uncommitted transactions.
- This principle also ensures us that we will never have to UNDO a committed transaction.
- “Recovery principle” principle also simplifies recovery process.
- The “recovery system” of DBMS should ensure that the database gets effects of all committed transactions and none of the uncommitted transactions.

Recoverable Schedule:

- A schedule is recoverable, in which no transaction T commits, till all transactions from which T reads are committed or aborted (i.e. transactions that have written data that T reads, are committed). [Note if any of transaction from which T reads aborts, T should also abort]

Note if transactions never abort; recovery is rather easy. Since all transactions eventually commit - DBMS can simply executes database operations as they arrive.

Primary responsibility of recoverability is to ensure “Durability” and “Atomicity” properties of ACID. DBMS maintains “logs of all updates” on database; and Durability and Atomicity is ensured by “redoing” certain operations (typically of committed transaction) and from logs. To ensure atomicity again systems uses logs and undoes typically undoes operations of uncommitted transactions

We will see recovery process later!

Concurrency Control Techniques

Objectives Concurrency Control is
to ensure “serializable” and “recoverable” schedules

Concurrency control and Recovery system go hand in hand; here, we plan to have abstract understanding of following techniques and algorithms

Concurrency Control Techniques (Algorithms)

- 2PL protocol
- Timestamp ordering based Protocols
- Multi-version Concurrency Control
- Snapshot Isolation

Recovery (Algorithms)

- Write Ahead Logging Protocol
- ARIES recovery algorithm