6.033 Spring 2015, Lecture 18
Hari Balakrishnan
April 13, 2015

Last update: May 17 2015

[This update clarifies Example B, case (f), explaining that the model assumes that no two writes to the same variable are commutative, and also fixes other minor errors pointed out by students.]

Isolation in Transactions: "Before-or-after" atomicity

GOAL: SERIALIZABILITY, i.e., "BEFORE-OR-AFTER". We want "correct semantics" when we have concurrent transactions.

Suppose we have N transactions, T_1, T_2, ..., T_N, where transaction T_i has one or more individual steps, or operations. A transaction processing system may execute these individual steps in any order it wishes (and, if it has multiple processors, some steps may run simultaneously too). Suppose that the system executes the individual steps in some order, causing an outcome or result. We can represent outcome in terms of the state of the system, which itself can be represented in terms of the values of the variables after running all the transactions, and by other actions that might have occurred during the transactions such as reading variables, printing messages to the user, sending email, etc.

Definition: The execution of the individual steps is said to be SERIALIZABLE if the outcome, starting from the initial state, S, is the same as the outcome obtained when running the N transactions one after another in *some* serial order. The specific permutation order of the transactions does not matter; what matters is that the outcome (including the final state, F, and any other steps in the transactions) be exactly the same as *some* serial ordering of the steps of the transaction.

Internally, the system may run the individual steps in any order it wishes, as long as it is serializable. Serializability formalizes the notion of "before-or-after" execution of the steps of a set of transactions.

In this lecture, we will do four things:

1) Using a simple example, we look at what happens when we violate serializability, to show that serializability is a powerful property that provides semantics that are easy to understand by applications. We will also look at a more subtle example where many serial orderings of steps are possible.

2) We define a data structure called a conflict graph, which we will construct for any given order of steps of the transactions. (In the literature, this order of steps is sometimes called the "schedule" and sometimes called the "execution path".) The conflict graph has one node for each transaction and a directed edge between nodes if there is a conflicting operation between the two transactions (we will define what a conflict is, later). We use the conflict graph to show a sufficiency condition for serializability; i.e., if the conflict graph does not have cycles, then the schedule of operations that produced the graph is serializable.

3) We then show a protocol, called two-phase locking (2PL), which is a locking discipline for how locks are acquired and released. 2PL provides serializability; we will establish that by showing that any schedule of steps produced by 2PL must have an acyclic conflict graph.

4) Finally, we discuss some practical issues in implementing serializability in practice in transactional database systems.

1. EXAMPLES

Example A: Let's start with a simple example of two transactions:

```
Transaction T1:                  Transaction T2:
   t0 = read(X)                     t2 = read(X)
   write(X, tmp+100)                write(X, 10*tmp2)
   t1 = read(Y)                     t3 = read(Y)
   write(Y, t1+t0)                  write(Y, 4*t3)
```

Suppose the initial value of X is x0 and the initial value of Y is y0. If T1 ran before T2, then the final value of X would be 10*(x0+100) and the final value of Y would be 4*(x0+y0). If, on the other hand, T2 ran before T1, the final value of X would be 10*x0 + 100 and the final value of Y would be 4*y0 + 10*x0 + 100.

The transaction processing system is free to execute the individual steps of the transactions in any order, as long as the *result* is what you would get if you ran either T1 fully before T2, or vice versa.

Many schedules (order of steps) are not serializabl in this example; for instance, running the first two operations of T2, then all of T1, and finally the last two operations of T2. In this execution, the final value of X is 10*x0+100 and the final value of Y is 4*(10*x0+100+y0). In general, that is not the same as T1 before T2 or T2 before T1.

Similarly, if we ran the first two operations of T1, then all of T2, and then the last two operations of T1, that execution is also not serializable. In this schedule of steps, at the end, X would be equal to 10*(x0+100) and Y would be 4*y0+10*(x0+100), which in general is not what you would get from T1 before T2 or T2 before T1.

If the system ran the first two steps of T1, then the first two steps of T2, then the last two steps of T1, and then the last two steps of T2, the result is the same as running T1 fully and then T2 fully. Hence, it is serializable.

Example B: Let's now look at a more subtle example, which we also looked at in lecture today (but did not do a great job on it!):

```
Transaction T1:                  Transaction T2:
    tmp = read(X.salary)             write(X.salary, 100000)
    write(Y.salary, 1.1*tmp)         write(Y.salary, 110000)
```

We can abstract these in terms of reads and writes, as follows:

```
T1:                               T2:
    read x                            write x
```

```
    write y                          write y
```

Let's write these four steps as r1(x), w1(y), w2(x), w2(y); the numbers 1 and 2 refer to the transaction ID, x and y are the variables, and r and x stand for "read" and "write", respectively.

What are the possible interleavings of steps within these transactions that the system could implement in different executions? (Note that within a transactions, the steps are in the order in which they appear in the transaction.)

(a) r1(x) w2(x) w1(y) w2(y)
(b) r1(x) w1(y) w2(x) w2(y)
(c) r1(x) w2(x) w2(y) w1(y)
(d) w2(x) r1(x) w2(y) w1(y)
(e) w2(x) w2(y) r1(x) w1(y)
(f) w2(x) r1(x) w1(y) w2(y)

Q: In this example, which of these is serializable?

A: If T1 ran before T2, then the final value of x would be 100000 and y would be 110000. If T2 ran before T1, the final value of X would also be 100000 and 110000. Any execution that produces these answers is serializable. So what we need to do is to check, for each execution order above, what x and y are at the end.

In this example, (a), (b), (d), (e) are all serializable.

(c) is *not* serializable in general: the final value of x is 100000 and the final value of y is 1.1*x0 (where x0 is the initial value of x before any step of the transactions runs), so (c) is serializable only if x0 happens to be 100000.

(f) is *not* serializable in general, although it is in this specific case because the values we chose made the two write steps *commutative*. Consider each serial order in turn. First, T1 before T2: here, although the final values of x and y are equivalent to T1 before T2, the read(x) in T1 sees a partial result written by T2. The read seen by T1 is T2's write to variable x, which is not atomic. Now consider whether the order of steps is equivalent to "T2 before T1". In this case, any other value except 110000 for w2(y) could cause the result not to be equivalent to "T2 before T1"; because we have abstracted away the values and focus only on the steps themselves, we consider this schedule to be non-serializable; i.e., we consider any two writes to the same variable to be non-commutative. By non-commutative, we mean that the final result depends on the order of the two writes.

2. CONFLICTS, CONFLICT SERIALIZABILITY, AND CONFLICT GRAPHS

Definition: A conflict occurs between two steps (operations) if the two steps operate on the same variable and if at least one of the two steps is a "write". (We will assume that no two writes of the same variable are commutative; this assumption is reasonable in general.) So two reads of the same variable don't conflict, but a read and a write or a write and a write conflict.

A stricter version of serializability is conflict serializability. We will use this stricter version because:
1) testing whether a schedule has this property is easier, and

2) any conflict-serializable schedule is also serializable.

We will show that the two-phase locking protocol is conflict-serializable (and is therefore also serializable). What we give up in return is that there may be other schedules of steps that are also serializable, and may even perform better than conflict-serializable ones.

Definition: A schedule is conflict-serializable if the order of its conflicts is the same as the order of conflicts in some serial order of transactions.

For example, in Example B above, (a), (b), (d), and (e) are conflict-serializable, but (c) and (f) are not. (a) and (b) are conflict-equivalent to the conflict order of "T1 before T2", while (d) and (e) are conflict-equivalent to the conflict order of "T2 before T1".

How do we systematically determine if any given execution order of steps is conflict-serializable?

A CONFLICT GRAPH is a data structure that we can use to test for conflict serializability. In a conflict graph, each node is a transaction. Edges are directed: there's an edge from node (transaction) T_i to node (transaction) T_j if, and only if, there is a conflicting step between T_i and T_j, and in the trace it occurs first in T_i.

Going back to the 6 interleavings for T1 and T2 in Example B above, we can see that a, b are T1 --> T2, whereas d, f are T2 --> T1. The other two have arrows in both directions.

To build some intuition, let's take a more elaborate example:

Example C: Four transactions.

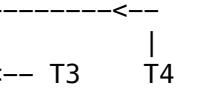
```
T1:          T2:         T3:        T4:
    r1(x)        w2(x)       r3(y)      r4(y)
    w1(y)        w2(y)       w3(z)
```

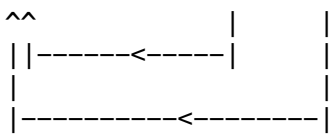
Suppose the trace of execution is
r1(x) w2(x) r3(y) r4(y) w1(y) w2(y) w3(z)
(First row left-to-right, then second row left-to-right.)

The conflicts are as follows:
r1(x) and w2(x)
r3(y) and w1(y)
r3(y) and w2(y)
r4(y) and w1(y)
r4(y) and w2(y)
w1(y) and w2(y)

Note that r3(y) and r4(y) do not conflict because both are reads. Note also that w3(z) is irrelevant because no other step uses z.

The conflict graph for the given trace of execution is:





Q: Is there a conflict-equivalent serial order of transactions here?
A: Yes. T3, T4, T1, T2. There's another too: T4, T3, T1, T2.

CONDITION FOR CONFLICT-SERIALIZABILITY: ACYCLIC CONFLICT GRAPH

Proposition 1: No cycles in conflict graph ==> conflict-serializable.
The converse, conflict-serializable ==> acyclic conflict graph, is
also true.

For example,

```
T1:                              T2:
    t1 = read(x)                     t1 = read(x)
    write(x, t1+10)                  write(x, t1+20)
    t2 = read(y)                     t2 = read(y)
    write(y, t2+20)                  write(y, t2+40)
```

We could run the first two steps of T1, then all of T2, and then the
next two steps fof T1. If x=X and y=Y before any step runs, the final
answer in this sequence of steps is x=X+30 and y=Y+60. The same values
of x and y hold if we were to run all the steps of T1 before T2, or
vice versa. But the transaction graph here has cycles because as far
as x is concerned, T1 runs before T2, and as far as y is concerned, T2
runs before T1.

As mentioned above, any conflict-serializable schedule is also
serializable, but any serializable schedule is not necessarily
conflict-serializable. (In Example B above, execution order (e) is
serializable but not conflict-serializable.)

We will show two propositions:

Proposition 1) Conflict graph is acyclic ==> conflict-serializable.

Then, later, we will show that, for a particular locking algorithm,
called TWO-PHASE LOCKING (2PL),

Proposition 2) Execution order produced by 2PL produces an acyclic
conflict graph.

Therefore, 2PL is conflict-serializable. And 2PL is also
serializable. The reason is that
2PL ==> acyclic conflict graph ==> conflict-serializable ==> serializable.

Proof of Proposition 1: Suppose the conflict graph is a directed
acyclic graph (DAG). Then, we can topologically sort its nodes. A
"topological sort" of a DAG is an ordering of nodes so that all edges
are between an earlier node to a later one in the sort order. Such a
topological sort is a serial ordering of the transactions.

Why is a topological sort possible? The reason is that every DAG has
at least one node that has no edges coming into it. (For if not, by
following a chain of arrows, you can get a cycle!) So start with that
node (breaking ties arbitrarily). Remove that node. We still have a
DAG in the induced graph on the remaining nodes, so we can apply this
idea of finding the node with no incoming edges recursively to

produce the topological sort.

3. TWO-PHASE LOCKING (2PL) PROTOCOL

We now have the background to invent a locking protocol to achieve serializability. Many protocols are possible; 2PL is one such. We will show that 2PL is correct, i.e., that it produces a serializable order.

2PL protocol:

Each shared variable has a lock. Before any operation on the variable, the transaction must acquire the corresponding lock. The 2PL protocol is simple: NO ACQUIRE OF ANY LOCK AFTER A RELEASE.

The acquisition and release of locks happens in two phases: an acquire phase and a release phase.

```
      acq l1
      acq l2
      release l1
      release l2
is 2PL

      acq l1
      acq l2
      release l2
      release l1
is 2PL

      acq l1
      acq l2
      release l1
      acq l3
      release l2
is NOT 2PL
```

Proposition 2: 2PL produces an acyclic conflict graph.

Proof: By contradiction. Suppose the conflict graph produced by an execution of 2PL has a cycle, which without loss of generality, is T1 --> T2 --> ... --> Tk --> T1. We shall show that a locking protocol that produces such a schedule must violate 2PL, i.e., that at least one transaction must acquire some lock after releasing a lock.

Let the shared variable between T_i and T_{i+1} be represented by xi. Looking at the different acquires and releases, with time flowing downwards, they must have occurred as follows:

```
 T1       T2       T3 ......               Tk    T1
 acq(l1)


          acq(l1)
|         acq(l2)
|
|
|                  acq(l3)
|                            ...
|                             ...
time
|                                 acq(lk)
```

```
|
v                                       acq(lk)

 acq(lk)
```

Now, note that for T2 to have done acq(l1), T1 must have done release(l1) *before* that the acquire(l1) in T2.

Hence, T1 looks like this:

```
       acq(l1)
       ...
       release(l1)
       ...
       ...
       acq(lk)
```

which violates the 2PL rule of no acquire after any release.

Therefore 2PL must produce an acyclic conflict graph.

From Proposition 1, it follows that 2PL is conflict-serializable. Because any conflict-serializable schedule of steps is also serializable, 2PL produces a serializable schedule.

4. PRACTICAL ISSUES

0. When can we release a lock? Because we need to hold the lock if the transaction ABORTs, no release for a variable we have written can actually occur before the COMMIT point. Locks for variables we are only reading may be released before COMMIT, but of course no acquires can occur after any release.

1. Types of locks. As described above, we lock even when two transactions simply want to read an item, which reduces concurrency.

Solution: read and write locks. A read lock is non-exclusive. A write lock is exclusive, and may be acquired only if no *other* transaction has a read lock.

2. Deadlocks: can deadlocks arise in 2PL?

Yes, T1 could hold a lock and T2 another, each waiting for the other. One way to handle it is using timers to ABORT slow transactions that appear to be making no progress.

Another approach is to detect cycles in the "waits-for" graph and terminate a transaction from the cycle.

3. Do we need to lock while recovering from a write-ahead log?

Two cases to consider: (1) we want to support new transactions while recovering, and (2) we finish recovery, and are then "open for business" for new transactions.

Most systems do some variant of (2) because checkpoints can make recovery fast enough. With (2), we don't need any special locking, because the log items would have been written before the crash by transactions following a suitable locking protocol, so there's no need to worry about it again, as long as we undo and redo in the order shown in the log.

4. Different semantics from serializability: in practice, serializability is often at odds with high concurrency and high performance, so systems and engineers have tweaked the semantics a lot. For example, some systems support "READ COMMITTED" transactions, in which a read is always guaranteed to obtain the last COMMITTED value, but only write-write steps are considered conflicts.

A different semantic is "snapshot isolation", which many databases provide by default. It's hard to understand why this is actually correct, but people do it anyway. The semantics here are that ONLY write-write steps are conflicting, not read-write. It does increase concurrency greatly, but makes the semantics harder to reason about.