Quiz two is this week on Friday from, I hope that's correct, whatever's on the webpage is correct, but I think it's lecture 9 through recitation number 15. So, in particular, the log structure file system paper is not on the exam.

Some of you may have been told otherwise.

What we're going to do today is continue our discussion of atomicity.

Which, as you'll recall, is two properties that have to hold in order for atomicity to hold.

And the first property that we talked about was recoverability.

And the other property, which is what we're going to spend most of our time on today, is isolation.

So, before we get into isolation, I just want to wrap up the discussion of recoverability because we left it a little bit of three-quarters and didn't quite finish it.

So the story here was what we did first was talked about how to achieve a recoverable sector using two copies of the data and the chooser sector to choose between them.

We talked about one way of achieving recoverability using version histories.

And we did complete that discussion.

And then we started talking about a more efficient way of achieving recoverability in situations where you cared a lot about performance using logging.

And the main in logging is this protocol for when you decide to write the log called the write ahead logging (WAL) protocol.

And the idea is very simple.

Always write to the log before you update cell store.

And that's the main discipline that if you always follow then you'll get things right.

But one of the consequences of logging, you get two things from having this log.

The first is when an action aborts.

Independent of failure, when you're running an atomic action and the program calls abort or the system aborts that action, what the log allows you to do is go back through the log, scan the log backwards, look at all of the steps that that action took.

.

And that action didn't quite commit so you have to back those changes out, and the log helps you unroll backward.

So primarily what you need with a logging scheme, in or to do aborts, is the ability to undo.

Which means that what you need in the log, whenever you update a cell store to something else, you need to keep track of the old value.

And this is what we called in the log as an undo step.

But now failures also happen in addition to aborts, and we need a little bit more mechanism in addition to just the ability to undo.

And to understand why, let's take a specific example where you have an application and it is writing data to a database that is on disk.

And what we said was there was a log and the log is stored on disk as well.

And let's assume that the log is actually stored on a different disk.

Now, the write ahead log protocol basically says that whenever you update, so you have an action which has things like reads and writes of cell store.

And whenever there's a write to a cell store, the write ahead log protocol says write to the log before you write to the database or to the cell store.

So two things can happen.

The first thing, the simplest model here is that all writes are synchronous.

What this means is that if you see a write statement in an atomic action, the first thing you have to do is to write to the log.

And then that returns.

And then you write to the database.

And then you run the next action, the next step of the action.

So clearly by the time in this action, if you ever get to the commit point and you're about to exit your commit, it means that all of the data has already been written to the database.

Because, by assumption, we assume that all of the writes are synchronous and you write to the database and only then do you go to the next step.

And, assuming there's only one thread of execution, then in this simple model, you always write to the database, you first write to the log and then you write to this self-store in the database.

But by construction, when you get to the commit point, you know for sure that all of the data has been written to the database.

So if a failure happens now and you didn't get to commit and the system failed before the commit ran any time in the middle of this action, the only thing you really need to do is to roll back all of the changes made by actions that didn't get to commit, which means that in this model that I've described so far, the only thing you need to do is to undo actions that respond to actions that didn't commit.

Any action that committed by construction must have written all of its data and installed that data into cell store.

Because if an action gets to the statement to run commit and then finishes commit, when it got to commit, you know that because of all the writes being synchronous to the cell store, you know that that write got written.

And, by the write protocol's definition, you know the log got written before that.

So you don't actually have to do anything.

Even though the log contains the results of these actions that committed, you don't have to do anything for the committed actions.

So, in fact, this undo log is enough to handle the simple database as well where all of the writes are being done synchronously in this application.

But there are a few things that can happen.

One of the reasons we threw out version histories or discarded the idea to go to this logging oriented model is for higher performance.

And one of the ways we got higher performance is we didn't have to do these link list reversals in order to read and write items.

But it's also going to be tempting, and you've seen this before, to not want to do synchronous writes to cell store on a database in order to get high performance.

You might have asynchronous writes happening to the database and you don't know when those writes complete.

And what could happen, as a result of that, is you'd issue some writes to the database and then you go ahead and you commit the action.

But what could happen is the data that was supposed to be written to this database as store may not actually have gotten written because there's some kind of a cache in memory that actually returned to you from the write, but the write actually never made it to the database.

So this system, for higher performance, if you stick the cache in here, what could happen is that you might reach the commit point and finish commit but not be guaranteed that the cell store data actually got written to the database.

And if you fail now after commit, you have to go through the log and redo the actions for all of those actions that committed because some of those actions may not have made it into the cell storage in the database.

So what this means is that in general, when you put a cash here, any memory cash here or if you have any situation where the writes are asynchronously being done to cell store, you need both an undo log in order to handle aborts and to handle the simple case and you need the ability to redo the results of certain actions from the log.

And both of these are done, so the system fails and then, when you recover, before you allow other actions that might be ready to go to take over, the system has to recover running the recovery process.

And the recovery happens from the log.

And the way that recovery works is, and we went through this the last time, you scan the log backwards from the most recent entry in the log.

And, while scanning the log backwards, you make two lists.

You make a list of winners and a list of losers.

And the winners are all of those actions that either committed or aborted.

And there's one important point in the abort step, which is that when the system calls abort or when the application calls abort and abort returns, what the system does is goes through the log and looks at all the changes made by that action and undoes all of them.

And then it writes the record, called the abort record, onto the log.

So when you recovery and you see an abort record, you already know that those changes made by an aborted action have been undone.

When you compose a list of winners that are committed actions and aborted actions, the only things you really need to redo are the steps corresponding to the committed actions.

You don't have to undo the steps corresponding to the aborted actions because they already got undone.

Because only after they got undone that the abort entry was written to the log.

In addition to these committed and aborted actions that are winners, there are all other actions that were pending or active at the time of the cache, and those are losers.

And so what you have to do to the losers is to undo the actions done by losers.

So the actual recovery step runs after composing these winners and losers and it corresponds to redoing the committed winners and undoing the losers.

And independent of crashes, independent of failures, when you just have actions that might abort, the only thing you really need is undo.

You don't need to redo anything.

If you don't ever have any failures but just actions aborting, the only thing you need to do with the log is to undo the results of uncommitted actions before abort returns.

Now, this procedure is OK.

Failures happen rarely and you're willing to take a substantial amount of time to recover from a failure then the log might be quite long.

If a system fails once a week, your log might be quite long.

And if you're willing to take the time to scan through the log entirely and build up these winners and losers list then this is fine, this approach works just fine.

But people are often interested in optimizing the time it takes to recover from a crash.

And a common optimization that's done is called a checkpoint.

And I believe you've seen this in System R.

And, if you haven't seen it, you'll probably see it in tomorrow's recitation.

And the main idea in the checkpoint is it's an optimization that allows this recovery process not to have to scan the log all the way back to time zero.

That the system periodically, while it's normally operating, takes this checkpoint, which is to write a special record into the log.

And that record basically says at this point in time here are all of the actions that have committed and whose results have already been installed into the cell store.

In other words, they've committed and already been installed so when you recover you don't have to go and scan the log all the way back.

You just have to go back enough so you find all of the actions whose results haven't yet been installed completely which have been committed.

And the checkpoint also contains a list of all the actions that are currently active.

So once you write a checkpoint record to the log, during recovery you don't have to go all the way back in time.

There are a few other optimizations that you can do with checkpoints that are not that interesting to get into here, but the primary optimization that's done to speed up the recovery process is to use this checkpoint record.

And most database systems, the checkpoint record is pretty small.

It's not like you're check pointing the entire state of the database.

The main thing you're doing is it's a pretty small amount of state that you're using to just speed up recovery.

So you shouldn't be thinking that the checkpoint is something where you take the entire database and copy it over.

That's not what goes on.

It's a pretty lightweight, small amount of state rather than the size of all of the data in the system.

So that's the story behind recoverability.

And we're actually going to come back to a piece of the story after we talk about isolation now because it will turn out that the mechanisms for isolation and the mechanisms for recoverability using logs interact in certain ways, so we have to come back to this either later today or on Wednesday.

So now we're going to start talking about isolation.

If you remember, the idea behind isolation is when you have a set of actions that run concurrently, what you would like is an equivalent ordering of the steps of the actions running concurrently such that the results are equivalent to some serial ordering of the actions.

The simple way of describing isolation is do it all before or do it all after.

If you have actions, let's say T1, T2 and so on running at the same time.

And these actions might operate on some data that is in common, they might act on data that's not at all uncommon, what you would like is an equivalent of the state of the system after running these concurrent actions should be equivalent to some serial ordering of the actions.

And it will turn out that what's tricky about isolation is that if you want isolation and you don't care about performance it's very, very easy to do.

So it's very easy to get isolation if you don't care about performance.

It's very easy to run fast if you don't care about correctness.

So, you know, fast and correct is the hard problem.

Fast and not correct is trivial and correct and slow is also trivial.

I mean correct and slow is very easy because you could take all of these concurrent actions and just run them one after the other so you don't actually take advantage of any potential concurrency that might be possible.

So suddenly slow and correct is very easy to do.

And we'll actually start out with slow and correct and then optimize a simple scheme.

There has also been a huge amount of work that's been done on fast and correct schemes.

And, in the end, they all boil down to this one basic idea that we'll talk about toward the end of lecture today.

So let's take an example.

Let's say you have two actions.

I'm going to call them with Ts because very often these are intended to be transactions which are consistency and durability in addition to isolation and recoverability.

So I'm just going to use the word T to represent an action.

Let's say you have an action that does read of some variable x and it does write of a variable y and you have transaction T2 that does write x and it does write y.

So we're going to take a few examples like this in order to understand what it means for actions to run such that they're steps equivalent to some serial order.

So we're going to spend some time really understanding that.

And then, once we understand what we want, it will turn out to be relatively easy to come up with schemes to achieve it.

Let's say that what happens here is that the system gets presented with these concurrent actions.

And let's assume each of these is an atomic step.

So there are four steps that can be interleaving in arbitrary ways, in any number of ways.

Let's say that what happens is you run that first and then you run that second and then you run this third and then you run this fourth.

So what you get is that I'm going to introduce a little bit of a notation here.

I'm going to write these steps as r1(x), w2(x), w1(y) and w2(y). What the r represents is a read, w represents a write, what the subscripts represent is the identifier of the action doing the read or write.

And what's in parenthesis is the variable that's being written.

So this says that action one does a read of x, action two does a write of x, action one does a write of y and action two does a write of y.

Now, if you look at these different actions, there are a few steps that conflict with each other and a few that don't.

If you look at the read of x and the write of y, they're independent of everything else.

If you just have read x here and write y here, they don't conflict with each other because those can happen in any order and the results are exactly the same.

Similarly, the write y and the write x don't conflict with each other.

The only things that really conflict with each other are the read x and the write x in this example because the results depend on which goes first.

Write y conflicts with this write y, and those are basically the two things that conflict with each other in this example.

Generally, if you have two actions conflict with one another, if they both contain a read and a write of the same variable or a write and a write of the same variable.

So there are basically three things that conflict.

If you have some variable, call it z, if you have a read of z and a write of z in one action or the other, if you have write of z and read of z or if you have write of z and write of z and those conflict with one another.

Now a read and a read don't conflict.

Because it doesn't matter what order they run in, they are going to give you the same answer.

If you look at this ordering of r1(x), w2(x), w1(y) and w2(y), the things that conflict are these two and these two.

Now, if you look at the two things that conflict and you draw arrows from which one happens before the other, what you'll find is that for this x conflict one runs before two and for this y conflict one runs before two.

So the arrows, if I were to draw them in time order, would point this way.

And this ordering is basically the same as the same ordering you would get, even though the steps run in different order, it's exactly the same as if you ran T1 completely before T2. Because running T1 completely before T2 says the ordering is r1(x), w1(y), w2(x) and w2(y). But the results are exactly the same, this different interleaving, which does one, two, three, four.

So what this says is that this trace that you get, we're going to use the word trace for this, you present this concurrent actions each of which has one or more steps and the system runs them.

And then the order in which the individual steps run produces a trace.

And then the question is whether that trace is what is called serializable.

A trace is serializable if the trace's results are identical to running the actions in some serial order one after the other.

And so what we're going to be trying to do is, what we're going to do today is to come up with schemes which take these different steps corresponding to the action that produces an order that turns out to be a serializable order.

And the challenge is to do it in a way that allows you to get reasonable performance.

To give you an example of a nonserializable order, if we had the following trace, r1(x) followed by w2(x) followed by w2(y) followed by w1(y). What happens here is that these two guys conflict so you've got an arrow from one to two going this way.

And, similarly, these two guys conflict but the arrow in time goes from two to one, which means that if you drew the arrow one to two this way, you would have an arrow going the opposite direction.

So this doesn't correspond to any serial order because, as far as this conflict is concerned, this trace says that action one should run before action two.

And, as far as this conflict is concerned, it says that action two should run before action one.

Which means you're in trouble because there is really no serial ordering here.

This trace does not correspond to either T1 before T2 or T2 before T1 which means this trace, if you have a scheme that runs your actions, the steps of your action that produces the stress it means that that scheme does not provide isolation.

Notice that we don't actually care whether T1 runs before T2 or T2 runs before T1. We're not worried about that.

We're just worried about producing some serial equivalent order.

So that's the definition of the property that we want, serializability.

And what it says is a trace whose conflict arrows, these are these conflict arrows, are equivalent to some serial ordering of the steps of the action.

What we want is a trace conflict that should be in the same order as some serial schedule or some serial order of the actions.

So what we're going to do is in three parts.

The first part is we're going to look at one of these traces.

And given a trace, the first problem is to figure out whether that trace corresponds to some serial order.

And then we're going to derive a property that guarantees that if a sudden property holds we would be assured that a trace is in serial order.

And then the second part is we are going to come up with various schemes for achieving serializability.

And the third part is in order to prove that those schemes are correct, what we are going to do is to prove that this property, that all serial orderings should satisfy holes for the protocol or for the algorithm that we design.

That is the plan for the rest of today.

This property for serializability is going to use data structure or a construction called an action graph.

And it turns out what we are going to do, given one of these traces, is produce a graph out of those traces called the action graph.

And then we are going to look to see whether a certain property holds for that action graph.

Let me show you what this action graph is by example.

The graph itself consists of nodes and edges [UNINTELLIGIBLE].

And the nodes are not these r1s and w2s. What the nodes are, are the actions themselves.

So, if you have four actions running, you have four nodes on the graph.

And then there are edges between these nodes on the graph.

Let's do it by an example.

Let's say you have first action T1 which has read of x and write of y.

And just so we are sure that it is action one, I'm going to draw one underneath as a subscript because they're just reads.

Action two has a write of x and a write of y.

Action three has a read of y and a write of another variable z.

And action four has a read of x.

First of all, given these actions, which of the actions conflict with each other?

Let's first write for T1. Does T2 conflict with T1? Yes it does because the read x, I mean that's the same as that example, so suddenly T2 conflicts with T1. What about T3? Does T3 conflict with T1? What that means is the interleaving of the individual steps matter as far as the final answer is concerned.

Yes, it does because the write of y and the read of y conflict, so you've got T1 and T3 that you have to worry about.

Does T1 conflict with T4? No it doesn't.

The read and the read do not conflict.

Does T2 conflict with T3? Yes, it does because it has got the y.

Does T2 conflict with T4? It does.

And does T3 conflict with T4? It does not.

There's nothing that's shared.

Out of the six possible, or whatever, four, choose two possible conflicts, for conflicts you've got four of them that you've got to worry about.

Now we're going to draw this graph that has T1, T2, T3 and T4, and we're going to call this the action graph.

And what it's going to do is to draw arrows between actions that conflict with one another.

But of course the answer, the arrows depend on the order in which these individual steps get scheduled by the system running these actions.

We need an actual example for that.

Let's say that what happens is you present these concurrent actions.

And what happens is this guy runs first and then two, three, four, five, six and seven.

That's the order in which the system runs the individual steps of this action.

Now we're going to draw these arrows between actions.

If two actions share a conflicting operation and in the first action the conflicting operation occurs before the second action then you're going to draw an arrow from the first action to the second action.

So more generally there's an arrow from PI to PJ.

If I and J have a conflicting operation that individual step is run by the system for I first before J.

If you look here at T1 to T2 there's an arrow between T1 and T2. Because it ran r1(x) before it run w2(x). If you look at T1 and T3, this is a little subtle because if ran read of x before it ran r3(y), but that doesn't matter because read of x there and read of y there don't conflict.

But then it ran w1(y) after it ran read 3y which means that the conflict is equivalent to running that step of T3 before the step of T1. So you actually have an arrow going back from T3 to T1. Now what about T2 and T3? T2 and T3, the same story, w2(x) and r3(y) don't conflict.

But r3(y) before w2(y), that's the conflict, so you have an arrow going this way.

So we've got three of them, you need a fourth one, and that is between T2 and T4. Between T2 and T4, w2(x) runs before r4(x) which means you have an arrow going this way.

If you actually look at this picture, and we'll come up with a method to systematically argue this point, but if you look at that schedule, as shown here, where the system runs the individual steps in that order, this is actually equivalent to T3 running and then T1 running and then T2 running and then T4 running.

The order of interleaving these different steps in the way shown in that picture, one, two, three, four, five, six, seven is actually equivalent to the same result, it's the same result that you get if you run T3 completely and then T1 and then T2 and then T4. In fact, if you think about it, it's also equivalent to the same ordering that you get if you run T3 and then you run T1 and then you run T4 and then you run T2. That just says that for the exact same scheduling of individual steps in the concurrent actions you might find multiple equivalent serial orders that all give you the same answer.

Is that clear?

The arrow from two to four is correct.

Write effects runs before read effects.

So I think this is correct.

Is there a problem?

All right.

So, in this example, it isn't multiple serial orders.

But in general it appears that there are multiple serial orders possible.

What does this action graph got to do with anything?

It turns out, and we'll prove this, that if the action graph-- --for any ordering of steps within an action, if the action graph does not have a cycle-- --then the corresponding trace from which the action graph was derived is serializable.

What that means is that what you have to do is, given a certain ordering of steps, you construct this action graph, you look to see if it has any cycles.

And, if it doesn't have any cycles, then you know that the order is equivalent to some serial order of the steps of the individual actions.

And it actually turns out the result is a bit more powerful.

The converse is also true that if you have a serializable trace then the corresponding action graph has no cycles.

Now, to turn out, the more interesting result for us is going to be the following direction where if the graph is acyclic then the trace is serializable.

Because what we're going to end up doing is inventing one or two protocols for achieving isolation, for achieving serializability.

And we're going to prove that those protocols are correct by proving that the corresponding action graph, all of the possible action graphs produced by those protocols all have no cycles.

So this direction is the direction that's actually more important for us, but the opposite is also true and not that hard to prove.

So what is the intuition behind why, if you have an action graph that's serializable, sorry, that doesn't have cycles the trace is serializable?

Well, notice one thing about this which is to draw a little bit of intuition.

Suppose, in fact, what happened here was we didn't execute the actions, the steps in that order, but what we did was to run this at step five and that at step six.

What would happen with the resulting action graph is that you would actually have an arch from T2 to T1 going the other way as well.

Because what this says is between T1 and T2 there is one conflicting operation that goes this way where action one runs before two.

And these two guys conflict where the step in two runs before the step in one and those two steps conflict with one another.

And this is actually the cycle here that causes the whole scheme to be not serializable anymore.

That's a little bit of intuition as to why this acyclic property is important.

But to really prove this notice that if you have a directed acyclic graph you could do something called a topological sort on the graph.

How many people know what a topological sort is?

OK.

For those who don't, the idea is very simple.

In any directed acyclic graph there is going to be at least one node that has no arrows coming into it.

All of the arrows going out are only going out of the node.

And you can actually prove that by arguing the contradiction.

If it turns out that every node has an arch coming in and an arch going out then by traversing that chain of pointers you'll end up with a cycle.

So it's pretty easy to see that in any directed acyclic graph you're going to have some node that has no arrows coming into it and only arrows going out of it.

So find that action, in this picture it is T3, and take that action and put it in first.

That's the first action that you run.

Because it has no arrows coming into it and only arrows going out, what that means is that no other action in a serial order runs before it.

Or at least there's no reason for any action to run before it because there are no arrows from any other action coming into this action.

So you put that in first.

Now, remove that node from the entire graph.

The resulting graph is acyclic, right?

You cannot manufacture a cycle by removing a node so you recursively apply the same idea, find some other node which has no other things coming into it and only things going out of it.

And if there are ties just pick one at random.

And, therefore, construct an order.

And all such orders that you construct are topological sort orders.

This topological sort order, by construction, is a serial order of the actions.

And this topological sort, if you now draw the arrows in the topological sort, they're going to be the same arrows as in the original directed graph, it's exactly the same graph, and now you have an equivalent serial order.

That's the reason why the cyclic property is actually important as far as serializability is concerned.

Now we can look at schemes that actually guaranty serializability.

And the schemes we're going to discuss all are in this system where you have cell storage and where you have logs for recovery.

The logs are not going to matter, for the most part.

You can also do isolation in version histories, and one of the sections of the notes deals with that at length.

I personally think it's not that important, but that doesn't mean it's not on the quiz.

I think you get a little bit more intuition reading that in addition to this discussion.

And a bulk of this discussion is actually not in the notes.

It's just another way of looking at the problem.

The mechanism we're going to build on is, in fact, described in the notes as well.

It's a mechanism you've seen before, and it is called locks.

As you recall, if you have variable x or any chunk of data you can protect it using two calls, acquire and release.

So you could do things like acquire lock of x and release lock of x.

And the lock protocol is that only one person can acquire a lock at a time.

And all of the people wanting all other actions wishing to acquire the same lock will wait until the lock is released and then they fight to acquire it.

And ultimately at the lowest level the lock has to be implemented with some low level atomic instruction.

For example, something like a test and set lock instruction.

It's the same story as before.

Now, there are a few things to worry about with locks.

The first one is the granularity of the lock.

You could be very, very conservative and decide that the granularity of the lock is your entire system.

So all of the data on the system gets protected with one lock, which means that you're running this very slow scheme because what you're guaranteeing is, in fact, isolation but you're running essentially one action at a time and you have no concurrency at all.

The other extreme, you could be very, very aggressive and every individual cell stored item is protected with the lock.

And now you're trying to max out the degree of concurrency, which means that although things could be fast you have to be a lot more careful if you want things to be correct.

And correct here means that you have an order that is equivalent to some serial order.

Why does the locking protocol actually matter?

Well, let's go back to these two action examples of r1(x) w2(x) and w1(y) and w2(y). And let's just throw in an acquire lock of x here and an acquire lock of y here and a release lock of x here and in between here you have the read and here you do the release of the lock of y and similarly you do an acquire lx here and you do an acquire ly here.

And then you release the locks at the end here.

So acquire lock x, read x, release lock x, acquire lock y, write y, release lock y, and acquire right, acquire right, release, release.

And, on the face of it, this kind of thing is actually reasonable for the old style synchronization, some of the old style synchronization things that we wanted because we didn't actual care about atomicity of full actions.

If you look at what happens with this kind of locking, as shown here, you might be a little bit in trouble because it could be that these three steps happen first and then this whole set of steps up to here happen second, actually, up to the end.

And then this chunk happens third.

Now you're in trouble because read x happens before write x and the write y happens before this write y.

And now you have not achieved isolation because, if you draw the conflict graph, you'll get an arrow from T1 to T2 and an arrow coming back from T2 to T1 and you have a cycle, which means that just throwing in the right acquires and releases isn't going to be sufficient to achieve isolation.

So we're going to need a much better set of skills, a better scheme than just this throw in the right acquires and releases.

The first scheme we're going to see is a simple scheme.

It's called simple locking.

The idea in simple locking is that every action knows beforehand all of the data items that it wants to read or write, and it acquires all of the locks before it does anything.

Before doing any reads or writes it acquires all of the locks.

The idea would be here you would acquire the lock of x and acquire the lock of y, and then you run the steps.

Similarly, the other action does the same thing.

And by construction, if you have actions that conflict with one another and one of the actions reaches the point where all of the locks it needs have been acquired then by construction no other conflicting action could be in the same state.

Because if you have another action that conflicts then it means that there's at least one data item common to them which means that only one of them could have reached this point because they're both trying to acquire.

This protocol where every action acquires all of the locks before running any step of the action will guaranty isolation.

And the isolation order, the equivalent serial order that it guarantees is the same as the order in which the different actions reach this point where they have acquired all of the locks.

And that point is also called the lock point.

The lock point of an action is defined as the point where all of the locks that it needs or it wants to acquire have been acquired.

And, because in this discipline where no action does any operation until all of the locks it needs have been acquired, you're guaranteed that this serial order of every action following this protocol, independent of knowledge of any other actions, everybody just blindly follows this protocol, the serial order is the same as if the actions had run in the order of acquiring the lock points, whatever that order might be.

This is called simple locking, and it does provide isolation.

But the problem with simple locking where you acquire all of the locks before running anything is two-fold. The first problem is that this dictates that all of the actions know the different items that they want to read or write beforehand.

Which means that if you're deep inside some action and there are all sorts of conditional statements you kind of have to know beforehand all of the data items you might be reading and writing.

And that could be pretty tricky.

In practice, if you want to adopt simple locking, you might have to be very conservative and sort of try to lock everything or lock a large amount of data which reduces performance.

Second, it actually doesn't give you enough opportunity to get high performance, even when you do know all of the data items beforehand.

For example, one thing you could do is acquire the lock of x and then read x.

And while you're off trying to read x then you might acquire a lock of y and read y.

And so, depending on how you have structured your system scheme where you do some work, maybe some computation as well in between the lock acquisition steps might give you higher performance.

So the question is can we be a little bit more aggressive than the simple locking scheme in order to get isolation as well as a little bit higher performance?

And the answer is that there is such a scheme and it's called two-phase locking.

And although a lot of work has happened for maybe a couple of decades on very high performance locking schemes, it turns out in the end they all, or at least for a large class of schemes that use this kind of locking they all boil down to some variable of two-phase locking.

And the idea is very simple.

The two-phase locking idea says there should be no release before all the acquires are done.

So do not release any lock until all of the locks that you need have been applied.

That's what happens here.

This schemes violates two-phase locking because you actually release lock x before you acquire lock y, and that violated two-phase locking.

And this idea that you don't release before all the acquires, there is a little bit of a subtlety that happens because you want recoverability which is that the action could at any stage abort.

And, in order to abort an action, you need to go back and undo that variable, the value to a previous value.

Which means that in order to abort you need to hold onto the lock.

You better make sure that an action in order to abort has the locks for all of the data items whose values it wishes to change to the original value.

Which means that in practice, at least for all the items that you're writing, the locks that you hold should not be released until the commit point, until the place where the action calls commit.

So no release before all acquires is basically equivalent to-- There should be no acquire statement after any release statement.

The moment you see a release statement and then an acquire after that of anything, then you know that this violates two-phase locking.

It turns out that two-phase locking is correct that it provides isolation.

And to see why let's look at a picture.

We're going to go back to this action graph idea and look at a picture of what happens with two-phase locking.

What we're going to prove is that if you use two-phase locking and you construct an action graph of what you get from running a sequence of steps that action graph has no cycles.

And we know that if you have no cycles in the action graph you're guaranteed that it is equivalent to some serial order.

We will argue this by contradiction.

Let's say you have T1 and T2 all the way through some action Tk and you have a cycle going back from Tk to T1 in the action graph.

Now, if there is an arrow from T1 to T2, it means that there is some data item x1 in common between T1 and T2. And you know that T2 ran after T1 which means that in T1 there was a release done of l1 after which in the system there was an acquire done of l1. Likewise, between T2 and T3, there is some data item x2 such that a release was done of l2 by T2. And after that an acquire was done of l2 by T3. And the release has to have been done after the acquire of l1 because we're following two-phase locking.

If you continue that down up to here out for Tk, Tk did an acquire somewhere later in time of lk minus one.

And then it did a release of some data item, a lock of lk where lk is actually some data item that is shared between Tk and T1, so there is actually some data item xk whose lock is lk.

And you know that Tk did a release of lk before T1 did an acquire of lk.

But the T1's acquire of lk must have happened after the release for lk so it must have happened at some point here. T1 must have done an acquire of lk at some point at the bottom here.

Just going out in time, by two-phase locking you get release of l1, then the other guy doesn't acquire of l1, release

of l2, acquire of l2 all the way out, then release of lk, and then after that in time an acquire of lk.

So that must have happened later on in time, but now this picture here violates two-phase locking because T1, for this cycle to hold, has to have done an acquire of lk after release of l1. But that violates two-phase locking because you're not allowed to acquire anything after you've released something.

So two-phase locking, therefore, cannot have a cycle in the action graph.

And, from the previous story, it means that it's equivalent to some serial order that corresponds to the topological sort of the directed acyclic graph.

I'm going to stop here.

We will continue with this stuff and then talk about other aspects of transactions next time.

And if there are any questions either send me an email or ask me the next time.