Good afternoon.

So we're going to continue our discussion about atomicity and how to achieve atomicity.

And today the focus is going to be on implementing this idea called recoverability, which we just described and defined the last time.

So if you recall from last time, the idea is that when you have modules that interact with one another, and in this example M1 calls M2, and M2 fails somewhere in the middle of this invocation and it recovers, the goal here is to try to make sure that the invoker of this module, in this case M1, or all subsequent invokers of M1, don't see any partial results that were computed during this execution when M2 failed.

And this was the idea that we called recoverability.

And the definition of recoverability was that an action, which is made up of a composite sequence of steps is recoverable from the point of view of its invoker, if it looks to the invoker and to all subsequent invokers as if this action either completely occurred, or if it didn't completely occur and aborted, it aborted in such a way that all partial effects of that action were undone or backed out.

So in other words, recoverability is this idea that you either do it all, either complete the action, or do none of the action.

But the effects are as if you were able to back out of the action.

And we use this idea to then talk about a particular special case of recoverability to implement a recoverable sector, which is a single sector of a disk where what we were able to do was to ensure that everybody reading, we defined a put procedure and a get procedure.

So, readers would invoke get.

And we ensure that everybody doing a get would never see the partial results of any put.

So, if a failure were to happen in the middle of a put, people doing a get wouldn't see these partial results.

And, the main idea here was to actually maintain what is more generally known as a shadow version, or a shadow copy, or a shadow object of the data, and we maintained two versions of the data that we call D0 and D1. And, we maintain a sector that we call the chooser sector to choose between the two shadows.

And, what we were able to argue was that this chooser always points to the version that you want people to get from to read from, and so when someone does a put, the idea is first to write to the version that's not currently being read from.

So the chooser points to zero.

Then the putter would put data, write data into one.

And if the failure happened in the middle of that write, there's no problem because people who read would still read from zero.

And we reduce this case of proving this algorithm correct to the case when a failure happened in the middle of writing the chooser sector.

And we were able to argue that as long as people, if a failure happened in the middle of writing here, either of these versions is correct because a failure by definition didn't happen in the middle of writing either of these two sectors.

And therefore you could pick either of them and read from it.

And during this process, we came up with this notion which we're going to generalize today called a commit point.

The commit point is the point at which for any action, the results are visible to subsequent actions.

And if a failure happens before the commit point, then the idea is, in general, you would not want people not to see the partial results that might have accumulated before the failure occurred.

And in this particular case, the commit point is when the chooser sector gets written to the current version of the data.

And that call to writing the chooser sector returns.

And if it returns, then you know that people doing a get will get from the version that just got written.

So, in the implementation of recoverable put, the commit point was when this call returned.

So now, the question for today is how we deal with larger actions -- -- because this is a plan that works pretty well for single sector puts and gets.

So, we were able to make individual sector reads and writes recoverable.

But if you think about any serious application or even any toy application, in most cases you end up having more data than what fits into one single sector.

And, you might have things touching data all over the place.

And, our approach to doing this is to actually first define what a programmer must do, what somebody wishing to write a program that is a recoverable action must do.

And then we're going to implement that underneath in a system so the programmer doesn't have to worry about implementing recoverability.

So the idea here is for the programmer of a recoverable action, to start writing that action using a system call, a call that they call begin recoverable action, and then discipline herself or himself to write some software which has a small number of rules as to what can go in here.

And then, explicitly, when they want to commit that recoverable action, make its results visible to subsequent actions, invoke commit.

And then, they are allowed to do a little bit more work, or a lot of work here.

But, there's very strict restrictions on what they can do after a commit.

And then, they can end using end recoverable action.

And this phase here before the commit is called the pre-commit phase.

This is the post-commit phase.

And the idea here is if a failure occurred here or an abort occurred before the commit and this action was made to abort, then the system must restore the state of all of the variables, and all of the data that was touched here to the same state before this action even got invoked.

OK, it's as if not of this happened.

So this is the not at all part of this definition of recoverability.

Once you reach this point of the commit returns, the only thing you're allowed to do here are things that cause you to complete.

You're not allowed to abort here.

You're not allowed to back out here.

So once you reach the point, it means you're in the do it all part of do it all or none at all.

So you have to complete all the way to the end.

And what this really means is that all of the data that you want to manipulate, and all of the resources that you want to accumulate, and we'll look at locks as a resource that you would like to accumulate in order to enforce isolation, which is a topic for next time, all that has to happen here so that once you reach this point and it ends, then even if a failure occurs when it restarts, you just have to crunch through and finish what was going on here.

And that can just happen.

There's nothing to acquire, no resources to get all of the data variables have already been put in their correct situation in the correct state.

So the interesting part really is what happens between the begin recoverable action and until the commit finishes.

And that's really what we're going to focus on.

Now in addition to commit, there is another call that we have to explicitly think about, and that's abort.

And there's two or three different ways in which abort may be invoked.

The first is a program that might herself or himself have abort in their code.

For example, in that bank transfer application, if you discover that your savings account doesn't have enough funds to cover a transfer, you read it, and then you maybe write something, and then you discover that you don't have the funds to cover the transfer.

You might just abort.

And the semantics of abort are that once abort is called by the programmer, they can be guaranteed that when the next person invokes a recoverable action that involves the same data items, those readers will see the same state as if this action never started.

So what this means is that the system must have a plan of undoing and backing out of any changes that might have occurred before this abort is called.

Another reason an abort might occur is that you're in a, for example, database complication, and you're booking all sorts of things like plane tickets, and air tickets, and hotel reservations, and so on.

And you book a few of them and then you discover you can't get one of the reservations that you want.

You might as a user might abort the whole transaction.

And that causes all the individual things that are in partial state to abort.

Another reason why abort might happen is that, and we'll see this the next time when we talk about locking, anytime you have locks, we already saw that anytime you have locks you have the danger of deadlock.

In one way in which the system implementing these atomic actions, both for isolation in particular, deals with deadlocks is when two or more actions are waiting for each other, waiting on locks that the others hold, you just abort one of them, or abort as many of them as needed for progress to happen.

So the system might unilaterally decide to abort certain actions.

And, what that means is that the systems' abort had better have a plan to undo all partial changes that might have occurred before it returns from abort.

OK, so that's the general model.

So what we're going to do today is to understand what happens when data variables are written inside one of these recoverable actions: how come it's implemented, and how abort is implemented.

And that's the plan.

And, once we do that, we will have implemented recoverability.

So we're going to study two solutions to this problem.

And the first solution uses an idea called version histories.

And version histories really build on an idea that we did see last time when we talked about recoverable sector, which is this rule that we call the golden rule of recoverability, which says never modify the only copy because if you modify the only copy of something and a failure occurs, then you don't really have a way of backing it out because you don't know what the original value was.

Version histories generalize the idea to say, never modify anything.

So the idea is anytime you want to write a variable, you don't actually overwrite anything.

You create another version of the variable and somehow arrange for the set of pointers that, for a variable to point to all of the versions of any given variable.

And to understand that, we need to understand the difference between conventional storage, like a conventional variable that is also called a cell store or a cell storage item, and a variable that allows you to implement versions which we're going to call [a journal?] based storage.

So, cell storage is traditional storage.

So if you have a variable, X, that's cell storage and you set X to some value, V, what ends up happening is that the cell that contains X is you write the value, V, into X.

In other words, you overwrite whatever there is you know, and replace it with V.

And, this overwriting really is what causes the problem if you don't have another copy of this variable somehow maintained, overwriting means that this rule of recoverabilities is being violated.

We're going to use the word install for these writes.

So we'll be installing items into cell stores.

So what that means is assigning a value to a cell store variable.

And the problem is this gets in the way of the golden rule.

So what were going to do is use these cell storage items that we know how to build that's the memory abstraction to build an expanded version called a journal storage of generalized storage in which nothing is ever overwritten.

The way this works is that if you have X, the very first time you set X to some value, you end up creating a data structure in cell storage that looks like this.

You have a value of V1. And you also keep track of the identifier of the action that created that.

And, that'll turn out to be useful for us to know the identifiers of the actions that created any given variable.

And how you get these identifiers?

When begin_RA is called, it returns an ID, OK, and the system knows that.

And this ID is available to the program as well.

Then the next version, if X gets set by any action to a different value, what you do is you created that as V2. And, you keep track of the identifier that maintains that.

And then you got V3, and so on, all the way.

And the current version, the latest version might be VN that was written by IDN.

Now if the same action repeatedly writes the same variable, you just create new versions.

So it isn't like there's one version per action.

It's just that there's one version every time you write something.

So literally, nothing is overwritten.

And so, that's X.

So, X itself points to the head version, the very latest version that was written.

And, you could imagine that there are these pointers pulling you back like a link list.

But the nice thing about it is this is the journal store.

So, X itself is this whole thing.

And, we'll implement two calls that when you have, this is basically a memory abstraction.

So, you need to read and you need to write.

So, for write, we're going to come up with a call called write journal, which in the notes I think has a slightly different name.

I think they call it write new value.

But write journal makes it clear that it's for journal store.

And, this is easy.

It's some data item, X.

It's some value, V.

And, it's the ID of the action that's doing the write.

And this is very easy to implement.

All you do is you create a new version.

And then you take the current thing that X is pointing to, and make the current version's next pointer point to that.

And then you make X point to the new version.

So, it's just a link list thing, OK?

And, in addition to write journal, we obviously need to implement read journal.

And read journal is going to take a data item that you wish to read, X, and for reasons that will become clearer in a minute, it also takes the ID of the action that wants to do the read, OK?

So if you want to read something, the idea is going to be the following: the idea is going to be that some of these actions are actions; some of these versions are going to have been written by actions that were committed.

OK, and some of these actions were going to have been written by actions that started writing things and then maybe failed or aborted.

So they never committed.

Now, clearly when you do read journal, you don't want to see the results of those actions that were never committed because what you want to see from the definition that we laid out are once you reach the commit point, you want to see the change is visible.

Before that, you don't want anything visible.

So as long as you can keep track of which of these actions committed, and which of these didn't commit, you can implement read journal by starting at the most recent version, and going backwards until you find the first version that corresponds to a value that was written by an action that was committed.

So what you need to do is start from here and look at IDN.

If IDN, you need to maintain another table that tells you whether IDN committed or not.

If it committed, then return that value.

If not, go back one.

And, keep going until you find the most recent version that was written by a committed action.

If you do that, then read journal clearly returns to you what you would want, which is the value that was written by the last committed action.

The only other tweak that you want to do, and the reason why ID is passed as an argument read journal is if the current action has already written, so let's say you are implementing an action and you set the value of X to 17, then when you read the value of X, you would want the value that you set.

I mean, you wouldn't want the previous committed action that's one way of defining read journal.

So as you go from the most recent version to the oldest version, you either look see whether the value that you are reading now is a value that you set, your own action set.

And if it was, just return that.

And then, it'll return to you the last value that this action set.

Otherwise, you keep going until you find the value set by the most recent committed action.

And since we aren't dealing here with concurrent actions at all, right, we've already said last time that, until next Monday, we're only going to be dealing with one action at a time.

There's no concurrent actions.

Clearly, this algorithm will be correct.

You start from the most recent version, keep going until you find the first version that was either done by this action that's doing the read, or the first version that was written by an action that committed.

So, clearly what this means is that you need a table that you have to maintain that stores the status of these different actions.

It needs to store which actions committed, and which actions didn't commit.

And that's going to be done using a data structure called the commit record table.

And this is a very simple table.

It just has ID1, ID2, all the way down to whatever ID's you have.

Every time somebody calls begin RA, you return them an ID, and then you create this table that as soon as they create this action, you set their state to pending, which I'll call P, OK?

And, any time an action commits, you replace this P with a C, which is a commit record.

OK, and once it's replaced with a C for an action, this item is called the commit record for an action.

So now, when you want to do read journal and you're looking to see whether for any given action, things were committed, the corresponding action is committed or not, you look at this.

You see its IDN.

You look for IDN in this table, C, if it's committed or not.

If it's not committed, then you go to the previous version and you do the same thing.

If it's committed, then you return it.

Now, it's not actually clear why you need this pending thing here.

But it'll turn out that you will require the pending thing when you deal with isolation on Monday.

So for now, you don't have to worry about the fact that these pending things are there, OK?

Now, suppose an action starts, and then it aborts.

So I mentioned here that when an action starts and it aborts, the system has to do some kind of undoing of data in order for abort to be correctly implemented.

So, the state of the system's restored to the state before the action even started.

The nice thing about this way of implementing version histories and read journal is you don't have to do anything on an abort.

If the application or the system called abort, nothing has to be done because read journal basically is just going scanning this backward, looking for whether the version was written by itself, that same action or looking for whether the version was written by a committed action.

So as long as you can find for any given ID whether it was committed or not, that's all you need.

OK, but just for completeness, and this will become useful the next time, all we'll do when abort is called on an action, so abort takes the ID of the action as an argument, all we'll do is we'll replace, if ID7 aborts, we'll just replace the pending.

We'll replace that with an abort, OK?

So, this commit record table contains the status of the actions.

And that status could either be committed, pending, or aborted.

When it starts, it's pending.

And then it's pending as long as either it aborts, in which case it aborted, or it's committed.

Now, if it just fails and you don't do anything about it, and there's no abort call, it'll continue to remain in the pending state.

But that's OK because we're never really going to read the value of anything that's the in the pending state that was set by an action that's in the pending state.

So is this clear?

OK, this approach is actually quite reasonable except that it has a few problems.

The first problem it has is, well, it has two related problems.

And that's the first class of problems that it has is that although it looks like we've really nailed this problem of achieving recoverable storage using this journal storage idea, building general recoverable actions so that for any variable that's read inside here or read inside a recoverable action, you use this general storage idea.

It's not quite correct because you have to ask, what happens if the system fails while the system is writing this commit record?

So, the application calls commit.

The system's starting to write this commit record and it fails.

Or you might more generally ask, what happens if I create this new version in write journal, and as I'm creating a new version of a variable, the system crashes.

So some garbage got written here.

Or more likely, some garbage got written not in here but as I was changing this pointer for X to point to the most recent version, some garbage got written.

So, all subsequent reads of X don't quite work.

The answer to this question is that we know how to solve this problem because that question is basically identical.

Both of these are identical.

If we know how to solve the problem of writing a single, recoverable sector, a single, small item of data, then we know how to solve these two problems because both of these are writing recoverably a small amount of data.

In one case, a pointer that takes X to point to the most recent version, in another case it's a single data item that corresponds to the commit record in this commit record table.

And so this shows this idea of bootstrap, that in order to build this atomic action, this recoverable action, we end up [SOUND OFF/THEN ON] and then you bootstrap on something that we know already how to do because there are these cases where you have to make sure that it writes to certain pointers, and some table items are done [commonly?]. And we know how to do that because we just told you how to do recoverable sectors. And you could just take [UNINTELLIGIBLE] objects for these items, and [UNINTELLIGIBLE PHRASE] to get this bootstrap.

So that's the first thing, the first [step problem?].

There's another problem, not so much a correctness problem, but a problem in general using these version histories in order to build recoverable actions.

Any ideas on what that might be?

Like, why would we want to use this?

Is this a space?

Well, you kind of can't really get around that.

I mean, it's true that there are these older versions that you keep forever.

But, there are organizations you can bring to bear that's [UNINTELLIGIBLE] beneath these old version that you can't really care about anymore because really the [UNINTELLIGIBLE] requires, at least for [UNINTELLIGIBLE PHRASE] about this when we talk about isolation tomorrow.

But really, the [UNINTELLIGIBLE] only requires for a single action case the last committed version.

So, you could garbage collect this stuff if you want.

Yeah, it's really slow.

So, for applications where you care about performance, a reasonable performance, [UNINTELLIGIBLE PHRASE] this is really slow.

And naturally, it's not to say that this is a bad idea, an idea that shouldn't be used at all.

In fact, it's a perfectly good idea for many cases where you might, for various reasons, want to store restorative records of old data and you don't care about fast read or write performance.

So it's perfectly good for certain applications.

But it's not good for applications that want reasonably high-performance.

And the reason that this thing is small is because if you think about it, it actually optimizes what you might think of as uncommon case because what it ensures is that when you fail and you recover, you have to do no work.

So crash recovery is really fast in this approach because there's nothing to be done for crash recovery.

But reads and writes are slow because a read involves [traversing?] the list.

A write involves [UNINTELLIGIBLE PHRASE].

And so, it almost optimizes the opposite of what you would want.

If you want to write performance, you want to form the principle of optimizing the common case.

And in order to optimize the common case, what it means, what you want to do here is to make the reads and writes really fast, and maybe pay the penalty of a little bit of extra turning in doing [UNINTELLIGIBLE PHRASE].

It's working now?

[LAUGHTER] Hello?

All right, thanks.

OK, so what you want to do is optimize, whoa, it's loud.

The integral of the volume over time is correct.

OK, so the solution to this problem where we want to optimize the common case of reads and writes, but we are OK taking a bunch of time to do crash recovery is an idea called logging.

So the way to think of a log is it's like a version history except you don't have a version for each variable.

You think of it as an Interleaf version data structure that interleaves all the version histories for all of the data that was ever written during an action, during all of the actions that ran.

So what this means is that you can write the log sequentially.

And you've seen this in yesterday's paper where they use logs for a different application for high performance in a file system for a system where writes normally would incur a lot of seeks.

But you can use the same idea.

In this case, we're going to use a log for crash recovery.

But the fundamental property of a log data structure is that it needs be written only sequentially.

And we know that disks do that pretty fast.

It's only when you have to seek that and read small chunks of data with seeks that you end up being really slow.

So we're going to use cell storage to satisfy our reads and writes.

So all of those are going to go to cell stores.

[You don't read?] means you just read a variable.

You don't traverse any link lists and writes.

You don't create any new versions.

You just write into cell store.

But then the log is going to be stored on a nonvolatile medium such as a disk.

And it's written sequentially.

So once we have those two, our plan is going to be as follows.

And this plan is the same plan that's adopted.

Although there is dozens of ways of doing log based crash recover, they all essentially follow the same basic plan.

You read and write normally to cell storage.

And you also write a copy of what you're reading and writing.

You write an encoding of what you're writing, any updates that you make into the log.

OK, and we'll talk in more detail about what you're exactly right into the log and when you write into the log, OK?

So that allows us to follow this golden rule of recoverability.

It'll turn out that the log is a copy of the data.

So you always have two copies of the data: one in cell storage, one on the log.

So what happens when you fail?

Well, when you fail, unlike in the version history case where you could fail and restart, and you don't have to do anything, here when you fail, the system runs a recovery procedure.

And that recovery procedure recovers from the log that we have conveniently arranged to write in the non-volatile storage.

So, it remains even after a crash, and it remains after a crash recovers.

And there are two things to do while recovering from the log.

For actions that didn't get to finish the commit, for actions that were uncommitted, which is this commit never return, what we have to do is to look carefully to see whether the corresponding cell store had any updates that were made to it.

And it'll turn out that the log is going to help us keep track of what items were updated by any given action.

And what we're going to end up doing is for uncommitted actions, we're going to back out.

In other words, we're going to undo any changes that it made, and the log is going to help us do that.

And conversely, for committed actions, because the semantics we want are that once committed, you would like the changes to be visible to other people.

For committed actions, what you would like to do are to make sure that the changes made by all committed actions are in fact installed in the cell store.

And what this means is that if they turn out to not have been installed, and we're going to use the log to tell us whether they've been installed or not, we will redo those actions.

And, the second thing we need to do is what happens if an abort is called either by the application or by the system.

Well, in this case, what we have to do is to use the log, and to keep track, the log is going to help us keep track of the changes made by this action to the cell store.

The cell store itself doesn't have an ocean of old or new because it's overwritten.

So the log is going to tell us that.

And when abort is called, we just want to back out by undoing the changes of the current action.

And that's the plan.

So the first thing we need to figure out is what this log looks like.

So as we saw from this discussion, the log is going to be required for us to do two things.

We're going to be undoing things from the log, and we're going to be redoing things from the log.

So what that suggests is that any time you update cell store, you change X from 17 to 25. What you'd really like to maintain is what the value was before the change was made so that you can undo if you need to, and what the value is after the change was made so that you can redo if you have to if by chance the actual cell store didn't get written at the right time.

So really the way to think about logging base crash recover is that the log is really the authoritative version of the data.

The cell store itself is you should think of as a cache.

And we've seen this idea before.

The cell store you should think of as a cache.

If a failure happens, you really have to be careful about trusting the cell store.

And, you don't trust what's in the cell store.

You start with a log, and by selectively undoing certain changes that were made and redoing certain changes, you produce a more pristine, correct version of the data, which corresponds to the changes made by all the committed actions being visible, and the changes made by all the uncommitted actions being wiped away to the previous version.

OK, so what does the log look like?

Well, as I've already said, a log is like a version history except it interleaves everything, and it's sequential.

So it's really an append-only data structure.

And there's a few different kinds of records that the log maintains.

In particular, two are going to be interesting to us.

So there are two types of records that we care about.

The first type are update records, which are written to the log whenever a cell store item changes.

So, if X goes from 17-25, what you would write is an update record that looks like this.

You store the ID of the transaction, sorry, ID of the recoverable action that did the update.

And then, you store two items.

One of them is an undo item or an undo action, actually.

And, an undo that might [save/say?], and a redo action.

So what this means here is that let's say that the actual step of this action said X is assigned to some value, new.

In the log, what you would write is keep track of old value, the current value of X, and make that the undo step.

And then, keep track of the change that was made and make that the real step.

So now, after doing this, if the system were to fail, and this action 172 were to never commit then you can systematically start with the log, start with the latest item in the log and go backwards, and undo any changes made by actions that didn't commit.

And conversely, and you might need to do this as well, you might want to look at all the actions that committed, and make sure that all those actions, those individual steps in those actions are redone so that once the crash recovers, you have a correct version of the data.

Now the other thing that you will need, and you'll see why in a moment, is another kind, a record and a log, which we're going to call the outcome record.

And this outcome is the thing that keeps track of whether an action committed or not.

Remember I said you're going to look through the log and figure out which actions committed, and which didn't commit.

You need to store that somewhere.

In particular, what that means is that when an action commits, you had better make sure that there is in it them in the log because the log really is the only correct version of the data.

So you have an outcome record, and this has an ID of the action.

It might be 174. And, there's a status that might stay committed.

And other values for the status might be aborted is a possible value of the status.

Another is pending.

So for various reasons, what we will have is when begin recoverable action returns with an ID, we will create a log entry that says that this action has begun.

So you might have a begin record.

It's not that important to worry about for now.

But the status of a committed record and an aborted, and the update type are important to understand.

So once you have this log structure understood, or the log data structure understood, what you have to think about our there are two questions that you end up spending a lot of time thinking about in designing these log-based protocols.

The first one is when to write the log.

And the second one is, you know, I sort of said you just look through the log and undo the guys who didn't commit, and redo the people who committed.

But you have to be very careful about doing that.

And that corresponds to this question of exactly how to recover, how to systematically recover so the state of the system is as I have described before.

So those are the questions we're going to deal with for the next few minutes.

Let's do this with a specific example.

And it will turn out and to answer doesn't really depend on the example.

But the example is good to give you the right intuition.

And this example is actually pretty common example of a disk-bound database.

So a disk bound database is one where you have applications writing to a database, which is where the cell storage is implemented.

And the cell storage is on disk.

So, you might have writes of cell items, X, and they go to a database.

And similarly, in any disk bound database that you want crash recovery for, you need to maintain a log.

And for various reasons having to do primarily with dealing with failures of the disk hardware itself, it's very often useful to an experience to maintain the log on a different disk.

So we'll maintain for this example the log on a different disk.

So whenever write X is done, just looking at the log data structure, you need to write an update record and append that to the log.

So at some point you would need to write this to the log.

You need to log the update -- -- that says that X change from something to something else.

So the question is, when do you write both of these?

So one approach might be that it really doesn't matter.

As long as the log gets the data, you're fine.

But that has a couple of problems.

In particular, suppose you write X without writing the log entry.

And as soon as you write X, before you have a chance to write to the log, you crash, or the system causes this program to abort, or the program itself aborts.

It writes X and then it does some calculation and the it decides to abort.

Now you are in trouble because the log hasn't kept track yet the log hasn't had a chance of keeping track of what the old value was, which means that if you really want to restore this database by undoing this write to X, you have to do a whole lot of work.

And it might be impossible to do it.

If you didn't know, for example, what the current value was, there was absolutely no way for you to restore to the old value.

So what this suggests is that you better not write to the cell store before you write to the log because if you wrote to the cell store log write, and the system crashed right after or failure about it, you won't really have a way in general of reverting to the version of the data item before this write.

And you do need to revert because it just aborted or fails.

So you need to back out of all changes that were made.

So that suggests the first part of our protocol which we are going to call the wall protocol.

Actually, that is the wall, I mean, not the first part.

This suggests this wall protocol.

Wall stands for write-ahead logging.

And the protocol says update the log or append to the log before you write to the cell store.

It's what it says.

Write ahead log says write the log before you write the cell store.

The advantage of writing the log before you write to the cell store is that suppose now you set X to some value and then you crashed.

Then you're guaranteed that if the cell store got written, the log got written, which means that if this action didn't commit, you can go through the log and undo that action because you know that the log entry got written correctly before the cell store got written.

And if the log entry didn't get written, then you know the cell store didn't get written, which means you don't have to undo anything for that particular data item.

So either way you're fine.

There is another part of this protocol that we're going to need to meet the semantics of a recoverable action that we wanted, which is that once you reach commit, you want the changes made by that action to be visible to all the other people, all of the other actions that are subsequent actions.

And what that means is that before you return from the commit, you had better make sure that the commit record for this action is logged to the disk, is logged, because if you didn't do that, and you just returned, then you can't be guaranteed that all of the writes that were done to the cell item were actually put on to the cell store.

There's no guarantee that these writes to the cell store actually got written to the cell store because all you are doing in this protocol is ensuring that the writes to the log are being written before the writes to the data.

Nobody is saying when the writes of the cell store really are happening and finishing, which means if the action commits, and you return committed to the user to the application, then you had better have a way of making sure that if the failure now happened, the system when it recovers knows that this action committed, which means it follows, then, that if you want those semantics that you'd better write the commit record, the fact that this action committed to the log before the commit returns.

And really the only reason you need that is that we've established; we've decided that we wanted the semantics [the?] different action commits, you want the results to be visible to everybody else.

And later on, we'll see that this is related to this notion of durability.

So write commit record before -- returning for commit.

So two main ideas: write ahead logging means make sure that you write the log, append to the log before you write to the cell store.

And in order to make sure that committed actions, the results of committed actions are visible even after failure to subsequent actions, log the commit record before you return from the commit.

So now we are actually in good shape to specify this recovery procedure that I've alluded to before because the log is going to contain these update records and these outcome records.

And that's going to allow us to decide what to do upon crash recovery.

And actually the only other piece we need is to decide what happens on an abort.

And that's actually pretty straightforward.

If the system calls abort, or if the user application calls abort on an action, what abort has to do is to look through the log.

Remember that all of the rights have been written.

Any time a write happens, you don't actually care about when the write actually happens at the cell store.

What you care about is that the write happens to the log before the write happens to the cell store.

So, if an abort were called, all you have to do is to ensure that before abort returns, all of the actions done by, all of the steps taken by this action around done, and the corresponding cell values are on done.

And that's all you have to do when you implement abort.

So one thing that I haven't really specified very clearly is when the actual writes happen to the disk or to any cell store.

And it turns out that it really doesn't matter.

If there's no failure, as long as you ensure, you could have caches in the middle.

You could have anything else.

So, as long as you ensure that if there's no concurrency, we'll deal with that next time.

But as long as you ensure that when you have actions that come one after the other that are recoverable that the values that are read are only the values that were written by previously committed actions, then it really doesn't matter when those were actually written to disk.

[NOISE OBSCURES] main thing that matters is make sure the log keeps track exactly of all the things to undo for uncommitted actions.

And for things that got committed, to make sure that the log keeps track of the commit record before the commit returns.

So given the story, the way the recovery procedure works as the following.

The first step is the system fails, and that it recovers.

You scan the log backwards.

And as you are scanning the log backwards, you keep track of two kinds of actions.

You keep track of actions that were either committed or were aborted, OK?

And what that means is that for actions that were committed or aborted, the cell store for those actions is in a certain state or needs to be in a certain state.

For committed actions, it needs to be in a state that's the result of finishing the committed action.

And for the aborted actions, what it means is that when the abort returned and there was an aborted action, abort already undid the state of the cell store by definition by the definition of the abort procedure.

So what that means is for log records that contain a type outcome and the status abort that you don't have to do anything because the changes are already on done before that abort record was written.

So what you do in scanning the log backwards is you build up two kinds of actions.

You build up winners, which are actions that were committed or aborted.

And you build up a list of losers that were none of these.

In other words, they were pending actions that kind of just during a failure they were pending, so they didn't commit.

And they were never aborted.

And so the plan now is to make sure that the cell store is correctly restored to the state that was before the crash where all of the committed actions' results are visible, and none of the uncommitted actions, you know, all of those are blown away.

All you have to do is to [UNINTELLIGIBLE] were committed.

You don't have to do anything for the aborted winners because they were already undone.

So you have to redo committed winners, and you have to undo any changes made by losers, right, because these losers by definition were things that didn't commit or didn't abort.

And the reason you only redo the committed winners rather than all winners is it makes no sense to redo aborted winners.

And you don't need to undo them because they were already undone when the abort record was written to the log.

So this is the basic idea for dealing with one of these databases.

But there's five or six optimizations that end up making this kind of system go faster.

You'll see some of these optimizations buried inside the system R paper, which is the discussion for tomorrow.

But what I'll do on Monday, I'll spend five minutes talking about the most important optimizations, and I think the whole story will become clear.

So the plan for the subsequent lectures on this topic are: on Monday we'll deal with isolation, and on Wednesday we'll continue to talk about isolation, and then talk about a different issue of consistency.