**MITOCW | ocw-6.033-lec17**

0:00:00 Good afternoon. So we're going to continue our 0:00:03.086 discussion about atomicity and how to achieve atomicity. 0:00:06.859 And today the focus is going to be on implementing this idea 0:00:10.907 called recoverability, which we just described and 0:00:14.268 defined the last time. So if you recall from last 0:00:17.56 time, the idea is that when you have modules that interact with 0:00:21.814 one another, and in this example M1 calls M2, and M2 fails 0:00:25.724 somewhere in the middle of this invocation and it recovers, 0:00:29.702 the goal here is to try to make sure that the invoker of this 0:00:33.818 module, in this case M1, or all subsequent invokers of 0:00:37.454 M1, don't see any partial results that were computed 0:00:40.952 during this execution when M2 failed. 0:00:45 And this was the idea that we called recoverability. 0:00:48.25 And the definition of recoverability was that an 0:00:51.246 action, which is made up of a composite sequence of steps is 0:00:55.006 recoverable from the point of view of its invoker, 0:00:58.129 if it looks to the invoker and to all subsequent invokers as if 0:01:02.081 this action either completely occurred, or if it didn't 0:01:05.523 completely occur and aborted, it aborted in such a way that 0:01:09.219 all partial effects of that action were undone or backed 0:01:12.725 out. So in other words, 0:01:15.211 recoverability is this idea that you either do it all, 0:01:18.78 either complete the action, or do none of the action. 0:01:22.281 But the effects are as if you were able to back out of the 0:01:26.118 action. And we use this idea to then 0:01:28.475 talk about a particular special case of recoverability to 0:01:32.38 implement a recoverable sector, which is a single sector of a 0:01:36.419 disk where what we were able to do was to ensure that everybody 0:01:40.594 reading, we defined a put procedure and a get procedure. 0:01:44.297 So, readers would invoke get. 0:01:48 And we ensure that everybody doing a get would never see the 0:01:51.557 partial results of any put. So, if a failure were to happen 0:01:55.055 in the middle of a put, people doing a get wouldn't see 0:01:58.311 these partial results. And, the main idea here was to 0:02:02.127 actually maintain what is more generally known as a shadow 0:02:06.169 version, or a shadow copy, or a shadow object of the data, 0:02:10.21 and we maintained two versions of the data that we call D0 and 0:02:14.536 D1. And, we maintain a sector that 0:02:16.876 we call the chooser sector to choose between the two shadows. 0:02:21.13 And, what we were able to argue was that this chooser always 0:02:25.314 points to the version that you want people to get from to read 0:02:29.639 from, and so when someone does a put, the idea is first to write 0:02:34.107 to the version that's not currently being read from. 0:02:39 So the chooser points to zero. Then the putter would put data, 0:02:42.21 write data into one. And if the failure happened in 0:02:44.842 the middle of that write, there's no problem because 0:02:47.526 people who read would still read from zero. 0:02:49.736 And we reduce this case of proving this algorithm correct 0:02:52.684 to the case when a failure happened in the middle of 0:02:55.368 writing the chooser sector. And we were able to argue that 0:02:58.368 as long as people, if a failure happened in the 0:03:00.789 middle of writing here, either of these versions is 0:03:03.421 correct because a failure by definition didn't happen in the 0:03:06.526 middle of writing either of these two sectors. 0:03:10 And therefore you could pick either of them and read from it. 0:03:14.331 And during this process, we came up with this notion 0:03:18.012 which we're going to generalize today called a commit point. 0:03:22.271 The commit point is the point at which for any action, 0:03:26.097 the results are visible to subsequent actions. 0:03:29.346 And if a failure happens before the commit point, 0:03:32.811 then the idea is, in general, you would not want 0:03:36.203 people not to see the partial results that might have 0:03:39.957 accumulated before the failure occurred. 0:03:44 And in this particular case, the commit point is when the 0:03:47.612 chooser sector gets written to the current version of the data. 0:03:51.612 And that call to writing the chooser sector returns. 0:03:54.903 And if it returns, then you know that people doing 0:03:58.064 a get will get from the version that just got written. 0:04:01.483 So, in the implementation of recoverable put, 0:04:04.322 the commit point was when this call returned. 0:04:08 So now, the question for today is how we deal with larger 0:04:13.764 actions -- 0:04:15 0:04:19 -- because this is a plan that works pretty well for single 0:04:23.142 sector puts and gets. So, we were able to make 0:04:26.357 individual sector reads and writes recoverable. 0:04:30 But if you think about any serious application or even any 0:04:33.814 toy application, in most cases you end up having 0:04:37.16 more data than what fits into one single sector. 0:04:40.305 And, you might have things touching data all over the 0:04:43.785 place. And, our approach to doing this 0:04:46.261 is to actually first define what a programmer must do, 0:04:49.808 what somebody wishing to write a program that is a recoverable 0:04:53.891 action must do. And then we're going to 0:04:56.434 implement that underneath in a system so the programmer doesn't 0:05:00.583 have to worry about implementing recoverability. 0:05:05 So the idea here is for the programmer of a recoverable 0:05:08.995 action, to start writing that action using a system call, 0:05:13.138 a call that they call begin recoverable action, 0:05:16.541 and then discipline herself or himself to write some software 0:05:20.98 which has a small number of rules as to what can go in here. 0:05:25.346 And then, explicitly, when they want to commit that 0:05:29.045 recoverable action, make its results visible to 0:05:32.448 subsequent actions, invoke commit. 0:05:36 And then, they are allowed to do a little bit more work, 0:05:39.635 or a lot of work here. But, there's very strict 0:05:42.676 restrictions on what they can do after a commit. 0:05:45.783 And then, they can end using end recoverable action. 0:05:49.154 And this phase here before the commit is called the pre-commit 0:05:53.186 phase. This is the post-commit phase. 0:05:55.566 And the idea here is if a failure occurred here or an 0:05:59.003 abort occurred before the commit and this action was made to 0:06:02.903 abort, then the system must restore the state of all of the 0:06:06.737 variables, and all of the data that was touched here to the 0:06:10.571 same state before this action even got invoked. 0:06:15 OK, it's as if not of this happened. 0:06:16.721 So this is the not at all part of this definition of 0:06:19.229 recoverability. Once you reach this point of 0:06:21.344 the commit returns, the only thing you're allowed 0:06:23.704 to do here are things that cause you to complete. 0:06:26.065 You're not allowed to abort here. 0:06:27.639 You're not allowed to back out here. 0:06:30 So once you reach the point, it means you're in the do it 0:06:33.069 all part of do it all or none at all. 0:06:35.043 So you have to complete all the way to the end. 0:06:37.564 And what this really means is that all of the data that you 0:06:40.744 want to manipulate, and all of

the resources that 0:06:43.375 you want to accumulate, and we'll look at locks as a 0:06:46.171 resource that you would like to accumulate in order to enforce 0:06:49.514 isolation, which is a topic for next time, all that has to 0:06:52.639 happen here so that once you reach this point and it ends, 0:06:55.764 then even if a failure occurs when it restarts, 0:06:58.285 you just have to crunch through and finish what was going on 0:07:01.519 here. And that can just happen. 0:07:04.59 There's nothing to acquire, no resources to get all of the 0:07:08.218 data variables have already been put in their correct situation 0:07:12.163 in the correct state. So the interesting part really 0:07:15.409 is what happens between the begin recoverable action and 0:07:18.909 until the commit finishes. And that's really what we're 0:07:22.345 going to focus on. Now in addition to commit, 0:07:25.145 there is another call that we have to explicitly think about, 0:07:28.963 and that's abort. And there's two or three 0:07:32.421 different ways in which abort may be invoked. 0:07:35.028 The first is a program that might herself or himself have 0:07:38.346 abort in their code. For example, 0:07:40.242 in that bank transfer application, if you discover 0:07:43.145 that your savings account doesn't have enough funds to 0:07:46.286 cover a transfer, you read it, 0:07:48.004 and then you maybe write something, and then you discover 0:07:51.322 that you don't have the funds to cover the transfer. 0:07:54.343 You might just abort. And the semantics of abort are 0:07:57.365 that once abort is called by the programmer, they can be 0:08:00.624 guaranteed that when the next person invokes a recoverable 0:08:04.001 action that involves the same data items, those readers will 0:08:07.497 see the same state as if this action never started. 0:08:12 So what this means is that the system must have a plan of 0:08:15.48 undoing and backing out of any changes that might have occurred 0:08:19.333 before this abort is called. Another reason an abort might 0:08:22.875 occur is that you're in a, for example, 0:08:25.237 database complication, and you're booking all sorts of 0:08:28.531 things like plane tickets, and air tickets, 0:08:31.141 and hotel reservations, and so on. 0:08:34 And you book a few of them and then you discover you can't get 0:08:37.588 one of the reservations that you want. 0:08:39.764 You might as a user might abort the whole transaction. 0:08:42.882 And that causes all the individual things that are in 0:08:45.941 partial state to abort. Another reason why abort might 0:08:49.058 happen is that, and we'll see this the next 0:08:51.529 time when we talk about locking, anytime you have locks, 0:08:54.764 we already saw that anytime you have locks you have the danger 0:08:58.352 of deadlock. In one way in which the system 0:09:01.781 implementing these atomic actions, both for isolation in 0:09:05.047 particular, deals with deadlocks is when two or more actions are 0:09:08.789 waiting for each other, waiting on locks that the 0:09:11.639 others hold, you just abort one of them, or abort as many of 0:09:15.143 them as needed for progress to happen. 0:09:17.34 So the system might unilaterally decide to abort 0:09:20.132 certain actions. And, what that means is that 0:09:22.745 the systems' abort had better have a plan to undo all partial 0:09:26.308 changes that might have occurred before it returns from abort. 0:09:31 OK, so that's the general model. 0:09:33.48 So what we're going to do today is to understand what happens 0:09:38.279 when data variables are written inside one of these recoverable 0:09:43.24 actions: how come it's implemented, and how abort is 0:09:47.32 implemented. And that's the plan. 0:09:49.879 And, once we do that, we will have implemented 0:09:53.48 recoverability. So we're going to study two 0:09:56.84 solutions to this problem. And the first solution uses an 0:10:01.32 idea called version histories. And version histories really 0:10:06.643 build on an idea that we did see last time when we talked about 0:10:10.282 recoverable sector, which is this rule that we call 0:10:13.217 the golden rule of recoverability, 0:10:15.154 which says never modify the only copy because if you modify 0:10:18.558 the only copy of something and a failure occurs, 0:10:21.317 then you don't really have a way of backing it out because 0:10:24.663 you don't know what the original value was. 0:10:27.128 Version histories generalize the idea to say, 0:10:29.71 never modify anything. So the idea is anytime you want 0:10:34.074 to write a variable, you don't actually overwrite 0:10:37.286 anything. You create another version of 0:10:39.828 the variable and somehow arrange for the set of pointers that, 0:10:43.91 for a variable to point to all of the versions of any given 0:10:47.791 variable. And to understand that, 0:10:49.933 we need to understand the difference between conventional 0:10:53.68 storage, like a conventional variable that is also called a 0:10:57.561 cell store or a cell storage item, and a variable that allows 0:11:01.576 you to implement versions which we're going to call a journal version 0:11:05.791 based storage. So, cell storage is traditional 0:11:10.075 storage. So if you have a variable, 0:11:12.35 X, that's cell storage and you set X to some value, 0:11:15.697 V, what ends up happening is that the cell that contains X is 0:11:19.713 you write the value, V, into X. 0:11:21.722 In other words, you overwrite whatever there 0:11:24.667 is you know, and replace it with V. 0:11:27.077 And, this overwriting really is what causes the problem if you 0:11:31.16 don't have another copy of this variable somehow maintained, 0:11:35.109 overwriting means that this rule of recoverabilities is 0:11:38.724 being violated. We're going to use the word 0:11:43.2 install for these writes. So we'll be installing items 0:11:47.519 into cell stores. So what that means is assigning 0:11:51.431 a value to a cell store variable. 0:11:54.039 And the problem is this gets in the way of the golden rule. 0:11:58.766 So what were going to do is use these cell storage items that we 0:12:03.9 know how to build that's the memory abstraction to build an 0:12:08.627 expanded version called a journal storage of generalized 0:12:13.11 storage in which nothing is ever overwritten. 0:12:18 The way this works is that if you have X, the very first time 0:12:23.513 you set X to some value, you end up creating a data 0:12:28.108 structure in cell storage that looks like this. 0:12:32.335 You have a value of V1. And you also keep track of the 0:12:36.952 identifier of the action that created that. 0:12:39.685 And, that'll turn out to be useful for us to know the 0:12:43.069 identifiers of the actions that created any given variable. 0:12:46.843 And how you get these identifiers? 0:12:48.991 When begin RA is called, it returns an ID, 0:12:51.854 OK, and the system knows that. And this ID is available to the 0:12:55.824 program as well. Then the next version, 0:12:58.297 if X gets set by any action to a different value, 0:13:01.42 what you do is you created that as V2. 0:13:05 And, you keep track of the identifier that maintains that. 0:13:08.329 And then you got V3, and so on, all the way. 0:13:10.841 And the current version, the latest version might be VN 0:13:13.995 that was written by IDN. Now if the same action 0:13:16.682 repeatedly writes the same variable, you just create new 0:13:19.894 versions. So it isn't like there's one 0:13:22.056 version per action. It's just that there's one 0:13:24.684 version every time you write something.

0:13:26.904 So literally, nothing is overwritten. 0:13:30 And so, that's X. So, X itself points to the head 0:13:33.346 version, the very latest version that was written. 0:13:36.763 And, you could imagine that there are these pointers pulling 0:13:40.877 you back like a link list. But the nice thing about it is 0:13:44.781 this is the journal store. So, X itself is this whole 0:13:48.407 thing. And, we'll implement two calls 0:13:50.917 that when you have, this is basically a memory 0:13:54.055 abstraction. So, you need to read and you 0:13:56.844 need to write. So, for write, 0:13:58.796 we're going to come up with a call called write journal, 0:14:02.631 which in the notes I think has a slightly different name. 0:14:08 I think they call it write new value. 0:14:10.364 But write journal makes it clear that it's for journal 0:14:13.846 store. And, this is easy. 0:14:15.423 It's some data item, X. 0:14:16.868 It's some value, V. 0:14:18.051 And, it's the ID of the action that's doing the write. 0:14:21.532 And this is very easy to implement. 0:14:23.766 All you do is you create a new version. 0:14:26.262 And then you take the current thing that X is pointing to, 0:14:30.007 and make the current version's next pointer point to that. 0:14:35 And then you make X point to the new version. 0:14:39.101 So, it's just a link list thing, OK? 0:14:42.364 And, in addition to write journal, we obviously need to 0:14:47.398 implement read journal. And read journal is going to 0:14:52.152 take a data item that you wish to read, X, and for reasons that 0:14:57.932 will become clearer in a minute, it also takes the ID of the 0:15:03.432 action that wants to do the read, OK? 0:15:08 So if you want to read something, the idea is going to 0:15:11.312 be the following: the idea is going to be that 0:15:14.125 some of these actions are actions; some of these versions 0:15:17.625 are going to have been written by actions that were committed. 0:15:21.437 OK, and some of these actions were going to have been written 0:15:25.187 by actions that started writing things and then maybe failed or 0:15:29.062 aborted. So they never committed. 0:15:31.062 Now, clearly when you do read journal, you don't want to see 0:15:34.75 the results of those actions that were never committed 0:15:38.062 because what you want to see from the definition that we laid 0:15:41.812 out are once you reach the commit point, 0:15:44.25 you want to see the change is visible. 0:15:48 Before that, you don't want anything 0:15:50.193 visible. So as long as you can keep 0:15:52.324 track of which of these actions committed, and which of these 0:15:56.085 didn't commit, you can implement read journal 0:15:58.843 by starting at the most recent version, and going backwards 0:16:02.478 until you find the first version that corresponds to a value that 0:16:06.49 was written by an action that was committed. 0:16:10 So what you need to do is start from here and look at IDN. 0:16:13.825 If IDN, you need to maintain another table that tells you 0:16:17.583 whether IDN committed or not. If it committed, 0:16:20.604 then return that value. If not, go back one. 0:16:23.489 And, keep going until you find the most recent version that was 0:16:27.651 written by a committed action. If you do that, 0:16:30.671 then read journal clearly returns to you what you would 0:16:34.295 want, which is the value that was written by the last 0:16:37.785 committed action. The only other tweak that you 0:16:41.464 want to do, and the reason why ID is passed as an argument read 0:16:44.595 journal is if the current action has already written, 0:16:47.222 so let's say you are implementing an action and you 0:16:49.747 set the value of X to 17, then when you read the value of 0:16:52.575 X, you would want the value that you set. 0:16:54.595 I mean, you wouldn't want the previous committed action that's 0:16:57.676 one way of defining read journal. 0:17:00 So as you go from the most recent version to the oldest 0:17:03.258 version, you either look see whether the value that you are 0:17:06.758 reading now is a value that you set, your own action set. 0:17:10.137 And if it was, just return that. 0:17:12.008 And then, it'll return to you the last value that this action 0:17:15.629 set. Otherwise, you keep going until 0:17:17.741 you find the value set by the most recent committed action. 0:17:21.241 And since we aren't dealing here with concurrent actions at 0:17:24.741 all, right, we've already said last time that, 0:17:27.456 until next Monday, we're only going to be dealing 0:17:30.353 with one action at a time. There's no concurrent actions. 0:17:35 Clearly, this algorithm will be correct. 0:17:37.464 You start from the most recent version, keep going until you 0:17:41.191 find the first version that was either done by this action 0:17:44.982 that's doing the read, or the first version that was 0:17:48.204 written by an action that committed. 0:17:50.416 So, clearly what this means is that you need a table that you 0:17:54.206 have to maintain that stores the status of these different 0:17:57.808 actions. It needs to store which actions 0:18:00.272 committed, and which actions didn't commit. 0:18:04 And that's going to be done using a data structure called 0:18:08.119 the commit record table. And this is a very simple 0:18:11.724 table. It just has ID1, 0:18:13.343 ID2, all the way down to whatever ID's you have. 0:18:16.801 Every time somebody calls begin RA, you return them an ID, 0:18:20.994 and then you create this table that as soon as they create this 0:18:25.555 action, you set their state to pending, which I'll call P, 0:18:29.749 OK? And, any time an action 0:18:32.408 commits, you replace this P with a C, which is a commit record. 0:18:36.204 OK, and once it's replaced with a C for an action, 0:18:39.204 this item is called the commit record for an action. 0:18:42.326 So now, when you want to do read journal and you're looking 0:18:45.877 to see whether for any given action, things were committed, 0:18:49.428 the corresponding action is committed or not, 0:18:52.122 you look at this. You see its IDN. 0:18:54.142 You look for IDN in this table, C, if it's committed or not. 0:18:57.755 If it's not committed, then you go to the previous 0:19:00.755 version and you do the same thing. 0:19:04 If it's committed, then you return it. 0:19:06.665 Now, it's not actually clear why you need this pending thing 0:19:10.916 here. But it'll turn out that you 0:19:13.221 will require the pending thing when you deal with isolation on 0:19:17.616 Monday. So for now, you don't have to 0:19:20.21 worry about the fact that these pending things are there, 0:19:24.244 OK? Now, suppose an action starts, 0:19:26.622 and then it aborts. So I mentioned here that when 0:19:30.692 an action starts and it aborts, the system has to do some kind 0:19:34.253 of undoing of data in order for abort to be correctly 0:19:37.288 implemented. So, the state of the system's 0:19:39.682 restored to the state before the action even started. 0:19:42.717 The nice thing about this way of implementing version 0:19:45.752 histories and read journal is you don't have to do anything on 0:19:49.313 an abort. If the application or the 0:19:51.298 system called abort, nothing has to be done because 0:19:54.216 read journal basically is just going scanning this backward, 0:19:57.66 looking for whether the version was written by itself, 0:20:00.754 that same action or looking for whether the version was written 0:20:04.373 by a committed action. So as long as you can find for 0:20:09.148 any given ID whether it was committed or not, 0:20:12.3 that's all you need. OK, but just for completeness,

0:20:15.882 and this will become useful the next time, all we'll do when 0:20:20.108 abort is called on an action, so abort takes the ID of the 0:20:24.191 action as an argument, all we'll do is we'll replace, 0:20:27.916 if ID7 aborts, we'll just replace the pending. 0:20:32 We'll replace that with an abort, OK? 0:20:34.448 So, this commit record table contains the status of the 0:20:38.122 actions. And that status could either be 0:20:40.775 committed, pending, or aborted. 0:20:42.816 When it starts, it's pending. 0:20:44.721 And then it's pending as long as either it aborts, 0:20:48.054 in which case it aborted, or it's committed. 0:20:50.979 Now, if it just fails and you don't do anything about it, 0:20:54.789 and there's no abort call, it'll continue to remain in the 0:20:58.666 pending state. But that's OK because we're 0:21:01.455 never really going to read the value of anything that's the in 0:21:05.605 the pending state that was set by an action that's in the 0:21:09.414 pending state. So is this clear? 0:21:13.185 OK, this approach is actually quite reasonable except that it 0:21:17.368 has a few problems. The first problem it has is, 0:21:20.645 well, it has two related problems. 0:21:22.946 And that's the first class of problems that it has is that 0:21:26.92 although it looks like we've really nailed this problem of 0:21:30.894 achieving recoverable storage using this journal storage idea, 0:21:35.147 building general recoverable actions so that for any variable 0:21:39.33 that's read inside here or read inside a recoverable action, 0:21:43.444 you use this general storage idea. 0:21:47 It's not quite correct because you have to ask, 0:21:50.46 what happens if the system fails while the system is 0:21:54.296 writing this commit record? So, the application calls 0:21:58.207 commit. The system's starting to write 0:22:00.991 this commit record and it fails. Or you might more generally 0:22:05.822 ask, what happens if I create this new version in write 0:22:09.467 journal, and as I'm creating a new version of a variable, 0:22:13.247 the system crashes. So some garbage got written 0:22:16.352 here. Or more likely, 0:22:17.702 some garbage got written not in here but as I was changing this 0:22:21.887 pointer for X to point to the most recent version, 0:22:25.194 some garbage got written. So, all subsequent reads of X 0:22:28.839 don't quite work. The answer to this question is 0:22:32.703 that we know how to solve this problem because that question is 0:22:36.223 basically identical. Both of these are identical. 0:22:38.948 If we know how to solve the problem of writing a single, 0:22:42.071 recoverable sector, a single, small item of data, 0:22:44.796 then we know how to solve these two problems because both of 0:22:48.146 these are writing recoverably a small amount of data. 0:22:51.098 In one case, a pointer that takes X to point 0:22:53.54 to the most recent version, in another case it's a single 0:22:56.719 data item that corresponds to the commit record in this commit 0:23:00.183 record table. And so this shows this idea of 0:23:04.535 bootstrap, that in order to build this atomic action, 0:23:08.929 this recoverable action, we end up [SOUND OFF/THEN ON] 0:23:13.408 and then you bootstrap on something that we know already 0:23:18.056 how to do because there are these cases where you have to 0:23:22.788 make sure that it writes to certain pointers, 0:23:26.507 and some table items are done [commonly?]. 0:23:29.971 And we know how to do that because we just told you how to 0:23:34.788 do recoverable sectors. And you could just take 0:23:39.561 [UNINTELLIGIBLE] objects for these items, and [UNINTELLIGIBLE 0:23:43.635 ] to get this bootstrap. So that's the first thing, 0:23:47.438 the first step problem. There's another problem, 0:23:50.833 not so much a correctness problem, but a problem in 0:23:54.228 general using these version histories in order to build 0:23:57.895 recoverable actions. Any ideas on what that might 0:24:01.624 be? Like, why would we want to use 0:24:03.539 this? Is this a space? 0:24:04.757 Well, you kind of can't really get around that. 0:24:07.426 I mean, it's true that there are these older versions that 0:24:10.733 you keep forever. But, there are organizations 0:24:13.344 you can bring to bear that's [UNINTELLIGIBLE] beneath these 0:24:16.709 old version that you can't really care about anymore 0:24:19.668 because really the [UNINTELLIGIBLE] requires, 0:24:22.221 at least for the [UNINTELLIGIBLE PHRASE] about this when we talk 0:24:25.645 about isolation tomorrow. But really, the 0:24:27.965 [UNINTELLIGIBLE] only requires for a single action case the 0:24:31.331 last committed version. So, you could garbage collect 0:24:36.188 this stuff if you want. 0:24:38 0:24:42 Yeah, it's really slow. So, for applications where you 0:24:44.917 care about performance, a reasonable performance, 0:24:47.559 [UNINTELLIGIBLE PHRASE] this is really slow. 0:24:49.926 And naturally, it's not to say that this is a 0:24:52.348 bad idea, an idea that shouldn't be used at all. 0:24:54.935 In fact, it's a perfectly good idea for many cases where you 0:24:58.183 might, for various reasons, want to store restorative 0:25:01.045 records of old data and you don't care about fast read or 0:25:04.128 write performance. So it's perfectly good for 0:25:07.803 certain applications. But it's not good for 0:25:10.716 applications that want reasonably high-performance. 0:25:14.184 And the reason that this thing is small is because if you think 0:25:18.485 about it, it actually optimizes what you might think of as 0:25:22.439 uncommon case because what it ensures is that when you fail 0:25:26.462 and you recover, you have to do no work. 0:25:30 So crash recovery is really fast in this approach because 0:25:33.517 there's nothing to be done for crash recovery. 0:25:36.343 But reads and writes are slow because a read involves 0:25:39.609 traversing the list. A write involves 0:25:42.059 [UNINTELLIGIBLE PHRASE]. And so, it almost optimizes the 0:25:45.514 opposite of what you would want. If you want to write 0:25:48.78 performance, you want to form the principle of optimizing the 0:25:52.548 common case. And in order to optimize the 0:25:55.061 common case, what it means, what you want to do here is to 0:25:58.641 make the reads and writes really fast, and maybe pay the penalty 0:26:02.598 of a little bit of extra turning in doing [UNINTELLIGIBLE 0:26:06.115 PHRASE]. It's working now? 0:26:10.914 [LAUGHTER] Hello? 0:26:14 0:26:21 All right, thanks. OK, so what you want to do is 0:26:27.064 optimize, whoa, it's loud. 0:26:30.29 The integral of the volume over time is correct. 0:26:37 0:26:43 OK, so the solution to this problem where we want to 0:26:47.18 optimize the common case of reads and writes, 0:26:50.786 but we are OK taking a bunch of time to do crash recovery is an 0:26:55.868 idea called logging. 0:26:58 0:27:04 So the way to think of a log is it's like a version history 0:27:07.949 except you don't have a version for each variable. 0:27:11.286 You think of it as an Interleaf version data structure that 0:27:15.236 interleaves all the version histories for all of the data 0:27:19.05 that was ever written during an action, during all of the 0:27:22.863 actions that ran. So what this means is that you 0:27:26.064 can write the log sequentially. And you've seen this in 0:27:29.741 yesterday's paper where they use logs for a different application 0:27:34.1 for high performance in a file system for a system where writes 0:27:38.322 normally would incur a lot of

seeks. 0:27:42 But you can use the same idea. In this case, 0:27:44.609 we're going to use a log for crash recovery. 0:27:47.219 But the fundamental property of a log data structure is that it 0:27:50.982 needs be written only sequentially. 0:27:53.046 And we know that disks do that pretty fast. 0:27:55.595 It's only when you have to seek that and read small chunks of 0:27:59.236 data with seeks that you end up being really slow. 0:28:03 So we're going to use cell storage to satisfy our reads and 0:28:07.873 writes. So all of those are going to go 0:28:11.067 to cell stores. You don't read means you 0:28:14.596 just read a variable. You don't traverse any link 0:28:18.63 lists and writes. You don't create any new 0:28:22.075 versions. You just write into cell store. 0:28:25.436 But then the log is going to be stored on a nonvolatile medium 0:28:30.563 such as a disk. And it's written sequentially. 0:28:37 0:28:45 So once we have those two, our plan is going to be as 0:28:49.391 follows. And this plan is the same plan 0:28:52.599 that's adopted. Although there is dozens of 0:28:56.146 ways of doing log based crash recover, they all essentially 0:29:01.044 follow the same basic plan. You read and write normally to 0:29:05.848 cell storage. And you also write a copy of 0:29:08.375 what you're reading and writing. You write an encoding of what 0:29:12.135 you're writing, any updates that you make into 0:29:14.908 the log. OK, and we'll talk in more 0:29:17.003 detail about what you're exactly right into the log and when you 0:29:20.886 write into the log, OK? 0:29:22.242 So that allows us to follow this golden rule of 0:29:25.077 recoverability. It'll turn out that the log is 0:29:27.85 a copy of the data. So you always have two copies 0:29:30.809 of the data: one in cell storage, one on the log. 0:29:35 So what happens when you fail? Well, when you fail, 0:29:38.536 unlike in the version history case where you could fail and 0:29:42.639 restart, and you don't have to do anything, here when you fail, 0:29:47.024 the system runs a recovery procedure. 0:29:49.57 And that recovery procedure recovers from the log that we 0:29:53.531 have conveniently arranged to write in the non-volatile 0:29:57.351 storage. So, it remains even after a 0:29:59.826 crash, and it remains after a crash recovers. 0:30:04 And there are two things to do while recovering from the log. 0:30:08.347 For actions that didn't get to finish the commit, 0:30:11.826 for actions that were uncommitted, which is this 0:30:15.231 commit never return, what we have to do is to look 0:30:18.782 carefully to see whether the corresponding cell store had any 0:30:23.13 updates that were made to it. And it'll turn out that the log 0:30:27.478 is going to help us keep track of what items were updated by 0:30:31.753 any given action. And what we're going to end up 0:30:36.228 doing is for uncommitted actions, we're going to back 0:30:40.092 out. In other words, 0:30:41.504 we're going to undo any changes that it made, 0:30:44.773 and the log is going to help us do that. 0:30:47.67 And conversely, for committed actions, 0:30:50.419 because the semantics we want are that once committed, 0:30:54.357 you would like the changes to be visible to other people. 0:30:58.518 For committed actions, what you would like to do are 0:31:02.307 to make sure that the changes made by all committed actions 0:31:06.616 are in fact installed in the cell store. 0:31:11 And what this means is that if they turn out to not have been 0:31:15.816 installed, and we're going to use the log to tell us whether 0:31:20.551 they've been installed or not, we will redo those actions. 0:31:25.127 And, the second thing we need to do is what happens if an 0:31:29.622 abort is called either by the application or by the system. 0:31:35 Well, in this case, what we have to do is to use 0:31:37.793 the log, and to keep track, the log is going to help us 0:31:41.002 keep track of the changes made by this action to the cell 0:31:44.33 store. The cell store itself doesn't 0:31:46.41 have an ocean of old or new because it's overwritten. 0:31:49.501 So the log is going to tell us that. 0:31:51.581 And when abort is called, we just want to back out by 0:31:54.671 undoing the changes of the current action. 0:31:58 0:32:04 And that's the plan. So the first thing we need to 0:32:08.105 figure out is what this log looks like. 0:32:11.289 So as we saw from this discussion, the log is going to 0:32:15.729 be required for us to do two things. 0:32:18.662 We're going to be undoing things from the log, 0:32:22.432 and we're going to be redoing things from the log. 0:32:26.537 So what that suggests is that any time you update cell store, 0:32:31.564 you change X from 17 to 25. What you'd really like to 0:32:36.47 maintain is what the value was before the change was made so 0:32:39.941 that you can undo if you need to, and what the value is after 0:32:43.47 the change was made so that you can redo if you have to if by 0:32:47 chance the actual cell store didn't get written at the right 0:32:50.47 time. So really the way to think 0:32:52.294 about logging base crash recover is that the log is really the 0:32:55.882 authoritative version of the data. 0:32:57.823 The cell store itself is you should think of as a cache. 0:33:02 And we've seen this idea before. 0:33:03.853 The cell store you should think of as a cache. 0:33:06.543 If a failure happens, you really have to be careful 0:33:09.532 about trusting the cell store. And, you don't trust what's in 0:33:13.119 the cell store. You start with a log, 0:33:15.271 and by selectively undoing certain changes that were made 0:33:18.619 and redoing certain changes, you produce a more pristine, 0:33:21.967 correct version of the data, which corresponds to the 0:33:25.076 changes made by all the committed actions being visible, 0:33:28.364 and the changes made by all the uncommitted actions being wiped 0:33:32.07 away to the previous version. OK, so what does the log look 0:33:36.479 like? Well, as I've already said, 0:33:38.112 a log is like a version history except it interleaves 0:33:40.765 everything, and it's sequential. So it's really an append-only 0:33:43.877 data structure. 0:33:45 0:33:55 And there's a few different kinds of records that the log 0:33:58.672 maintains. In particular, 0:34:00.245 two are going to be interesting to us. 0:34:03 0:34:08 So there are two types of records that we care about. 0:34:12.178 The first type are update records, which are written to 0:34:16.517 the log whenever a cell store item changes. 0:34:19.892 So, if X goes from 17-25, what you would write is an 0:34:23.991 update record that looks like this. 0:34:26.723 You store the ID of the transaction, sorry, 0:34:30.098 ID of the recoverable action that did the update. 0:34:35 And then, you store two items. One of them is an undo item or 0:34:41.293 an undo action, actually. 0:34:43.811 And, an undo that might save, and a redo action. 0:34:50 0:34:54 So what this means here is that let's say that the actual step 0:34:58.65 of this action said X is assigned to some value, 0:35:02.234 new. In the log, what you would 0:35:04.521 write is keep track of old value, the current value of X, 0:35:08.79 and make that the undo step. And then, keep track of the 0:35:12.984 change that was made and make that the real step. 0:35:16.643 So now, after doing this, if the system were to fail, 0:35:20.608 and this action 172 were to never commit then you can 0:35:24.572 systematically start with the log, start with the latest item 0:35:29.147 in the log and go backwards, and undo any changes made by

log, start with the latest new checkpoint A in the log and go backwards, and undo any changes made by 0:35:33.416 actions that didn't commit. And conversely, 0:35:38.048 and you might need to do this as well, you might want to look 0:35:42.244 at all the actions that committed, and make sure that 0:35:45.881 all those actions, those individual steps in those 0:35:49.307 actions are redone so that once the crash recovers, 0:35:52.804 you have a correct version of the data. 0:35:55.461 Now the other thing that you will need, and you'll see why in 0:35:59.657 a moment, is another kind, a record and a log, 0:36:02.804 which we're going to call the outcome record. 0:36:07 And this outcome is the thing that keeps track of whether an 0:36:09.889 action committed or not. Remember I said you're going to 0:36:12.583 look through the log and figure out which actions committed, 0:36:15.473 and which didn't commit. You need to store that 0:36:17.726 somewhere. In particular, 0:36:18.902 what that means is that when an action commits, 0:36:21.155 you had better make sure that there is in it them in the log 0:36:24.044 because the log really is the only correct version of the 0:36:26.787 data. So you have an outcome record, 0:36:28.502 and this has an ID of the action. 0:36:31 It might be 174. And, there's a status that 0:36:35.34 might stay committed. And other values for the status 0:36:40.714 might be aborted is a possible value of the status. 0:36:45.881 Another is pending. So for various reasons, 0:36:50.221 what we will have is when begin recoverable action returns with 0:36:56.629 an ID, we will create a log entry that says that this action 0:37:02.726 has begun. So you might have a begin 0:37:06.845 record. It's not that important to 0:37:09.28 worry about for now. But the status of a committed 0:37:12.897 record and an aborted, and the update type are 0:37:16.219 important to understand. So once you have this log 0:37:19.835 structure understood, or the log data structure 0:37:23.23 understood, what you have to think about our there are two 0:37:27.438 questions that you end up spending a lot of time thinking 0:37:31.571 about in designing these log-based protocols. 0:37:36 The first one is when to write the log. 0:37:38 0:37:44 And the second one is, you know, I sort of said you 0:37:47.409 just look through the log and undo the guys who didn't commit, 0:37:51.568 and redo the people who committed. 0:37:53.818 But you have to be very careful about doing that. 0:37:57.09 And that corresponds to this question of exactly how to 0:38:00.772 recover, how to systematically recover so the state of the 0:38:04.659 system is as I have described before. 0:38:08 So those are the questions we're going to deal with for the 0:38:11.95 next few minutes. Let's do this with a specific 0:38:15.083 example. And it will turn out and to 0:38:17.467 answer doesn't really depend on the example. 0:38:20.396 But the example is good to give you the right intuition. 0:38:24.142 And this example is actually pretty common example of a 0:38:27.82 disk-bound database. 0:38:30 0:38:34 So a disk bound database is one where you have applications 0:38:40.692 writing to a database, which is where the cell storage 0:38:46.807 is implemented. And the cell storage is on 0:38:51.538 disk. So, you might have writes of 0:38:55.346 cell items, X, and they go to a database. 0:39:01 And similarly, in any disk bound database that 0:39:04.074 you want crash recovery for, you need to maintain a log. 0:39:07.832 And for various reasons having to do primarily with dealing 0:39:11.795 with failures of the disk hardware itself, 0:39:14.596 it's very often useful to an experience to maintain the log 0:39:18.559 on a different disk. So we'll maintain for this 0:39:21.701 example the log on a different disk. 0:39:24.093 So whenever write X is done, just looking at the log data 0:39:27.919 structure, you need to write an update record and append that to 0:39:32.223 the log. So at some point you would need 0:39:35.425 to write this to the log. You need to log the update -- 0:39:38 0:39:43 -- that says that X change from something to something else. 0:39:46.722 So the question is, when do you write both of 0:39:49.498 these? So one approach might be that 0:39:51.706 it really doesn't matter. As long as the log gets the 0:39:54.987 data, you're fine. But that has a couple of 0:39:57.637 problems. In particular, 0:39:59.088 suppose you write X without writing the log entry. 0:40:03 And as soon as you write X, before you have a chance to 0:40:06.849 write to the log, you crash, or the system causes 0:40:10.27 this program to abort, or the program itself aborts. 0:40:13.906 It writes X and then it does some calculation and the it 0:40:17.826 decides to abort. Now you are in trouble because 0:40:21.177 the log hasn't kept track yet the log hasn't had a chance of 0:40:25.382 keeping track of what the old value was, which means that if 0:40:29.588 you really want to restore this database by undoing this write 0:40:33.936 to X, you have to do a whole lot of work. 0:40:38 And it might be impossible to do it. 0:40:40.393 If you didn't know, for example, 0:40:42.512 what the current value was, there was absolutely no way for 0:40:46.478 you to restore to the old value. So what this suggests is that 0:40:50.649 you better not write to the cell store before you write to the 0:40:54.82 log because if you wrote to the cell store log write, 0:40:58.376 and the system crashed right after or failure about it, 0:41:02.068 you won't really have a way in general of reverting to the 0:41:05.965 version of the data item before this write. 0:41:10 And you do need to revert because it just aborted or 0:41:13.805 fails. So you need to back out of all 0:41:16.492 changes that were made. So that suggests the first part 0:41:20.522 of our protocol which we are going to call the wall protocol. 0:41:25 Actually, that is the wall, I mean, not the first part. 0:41:29.029 This suggests this wall protocol. 0:41:31.417 Wall stands for write-ahead logging. 0:41:35 0:41:40 And the protocol says update the log or append to the log 0:41:44.666 before you write to the cell store. 0:41:47.5 It's what it says. Write ahead log says write the 0:41:51.5 log before you write the cell store. 0:41:55 0:42:01 The advantage of writing the log before you write to the cell 0:42:05.339 store is that suppose now you set X to some value and then you 0:42:09.75 crashed. Then you're guaranteed that if 0:42:12.498 the cell store got written, the log got written, 0:42:15.897 which means that if this action didn't commit, 0:42:19.152 you can go through the log and undo that action because you 0:42:23.346 know that the log entry got written correctly before the 0:42:27.324 cell store got written. And if the log entry didn't get 0:42:31.905 written, then you know the cell store didn't get written, 0:42:35.348 which means you don't have to undo anything for that 0:42:38.483 particular data item. So either way you're fine. 0:42:41.372 There is another part of this protocol that we're going to 0:42:44.877 need to meet the semantics of a recoverable action that we 0:42:48.381 wanted, which is that once you reach commit, 0:42:51.024 you want the changes made by that action to be visible to all 0:42:54.713 the other people, all of the other actions that 0:42:57.54 are subsequent actions. And what that means is that 0:43:01.72 before you return from the commit, you had better make sure 0:43:05.415 that the commit record for this action is logged to the disk, 0:43:09.238 is logged, because if you didn't do that, 0:43:11.787 and you just returned, then you can't be guaranteed 0:43:14.973 that all of the writes that were done

to the cell item were 0:43:18.669 actually put on to the cell store. 0:43:20.771 There's no guarantee that these writes to the cell store 0:43:24.276 actually got written to the cell store because all you are doing 0:43:28.29 in this protocol is ensuring that the writes to the log are 0:43:31.985 being written before the writes to the data. 0:43:36 Nobody is saying when the writes of the cell store really 0:43:39.583 are happening and finishing, which means if the action 0:43:42.976 commits, and you return committed to the user to the 0:43:46.239 application, then you had better have a way of making sure that 0:43:50.208 if the failure now happened, the system when it recovers 0:43:53.728 knows that this action committed, which means it 0:43:56.735 follows, then, that if you want those 0:43:59.039 semantics that you'd better write the commit record, 0:44:02.304 the fact that this action committed to the log before the 0:44:05.887 commit returns. And really the only reason you 0:44:10.163 need that is that we've established; we've decided that 0:44:14.059 we wanted the semantics the different action commits, 0:44:18.026 you want the results to be visible to everybody else. 0:44:21.777 And later on, we'll see that this is related 0:44:24.878 to this notion of durability. So write commit record before 0:44:29.062 -- 0:44:30 0:44:35 -- returning for commit. 0:44:36 0:44:46 So two main ideas: write ahead logging means make 0:44:48.65 sure that you write the log, append to the log before you 0:44:51.742 write to the cell store. And in order to make sure that 0:44:54.723 committed actions, the results of committed 0:44:57.042 actions are visible even after failure to subsequent actions, 0:45:00.355 log the commit record before you return from the commit. 0:45:04 0:45:11 So now we are actually in good shape to specify this recovery 0:45:15.488 procedure that I've alluded to before because the log is going 0:45:20.051 to contain these update records and these outcome records. 0:45:24.314 And that's going to allow us to decide what to do upon crash 0:45:28.728 recovery. And actually the only other 0:45:31.673 piece we need is to decide what happens on an abort. 0:45:34.834 And that's actually pretty straightforward. 0:45:37.438 If the system calls abort, or if the user application 0:45:40.661 calls abort on an action, what abort has to do is to look 0:45:44.132 through the log. Remember that all of the rights 0:45:47.045 have been written. Any time a write happens, 0:45:49.71 you don't actually care about when the write actually happens 0:45:53.429 at the cell store. What you care about is that the 0:45:56.466 write happens to the log before the write happens to the cell 0:46:00.185 store. So, if an abort were called, 0:46:02.293 all you have to do is to ensure that before abort returns, 0:46:05.826 all of the actions done by, all of the steps taken by this 0:46:09.359 action around done, and the corresponding cell 0:46:12.148 values are on done. And that's all you have to do 0:46:17.577 when you implement abort. So one thing that I haven't 0:46:22.199 really specified very clearly is when the actual writes happen to 0:46:27.888 the disk or to any cell store. And it turns out that it really 0:46:32.882 doesn't matter. If there's no failure, 0:46:35.129 as long as you ensure, you could have caches in the 0:46:38.165 middle. You could have anything else. 0:46:40.351 So, as long as you ensure that if there's no concurrency, 0:46:43.751 we'll deal with that next time. But as long as you ensure that 0:46:47.455 when you have actions that come one after the other that are 0:46:51.037 recoverable that the values that are read are only the values 0:46:54.681 that were written by previously committed actions, 0:46:57.656 then it really doesn't matter when those were actually written 0:47:01.36 to disk. But, for recovering to work, the main thing 0:47:05.021 that matters is make sure the log keeps track exactly of all 0:47:09.439 the things to undo for uncommitted actions. 0:47:12.584 And for things that got committed, to make sure that the 0:47:16.702 log keeps track of the commit record before the commit 0:47:20.671 returns. So given the story, 0:47:22.693 the way the recovery procedure works as the following. 0:47:26.661 The first step is the system fails, and that it recovers. 0:47:30.855 You scan the log backwards. 0:47:34 0:47:39 And as you are scanning the log backwards, you keep track of two 0:47:43.952 kinds of actions. You keep track of actions that 0:47:47.647 were either committed or were aborted, OK? 0:47:50.871 And what that means is that for actions that were committed or 0:47:55.666 aborted, the cell store for those actions is in a certain 0:48:00.069 state or needs to be in a certain state. 0:48:04 For committed actions, it needs to be in a state 0:48:06.565 that's the result of finishing the committed action. 0:48:09.348 And for the aborted actions, what it means is that when the 0:48:12.514 abort returned and there was an aborted action, 0:48:15.025 abort already undid the state of the cell store by definition 0:48:18.3 by the definition of the abort procedure. 0:48:20.483 So what that means is for log records that contain a type 0:48:23.539 outcome and the status abort that you don't have to do 0:48:26.432 anything because the changes are already on done before that 0:48:29.653 abort record was written. So what you do in scanning the 0:48:34.243 log backwards is you build up two kinds of actions. 0:48:37.983 You build up winners, which are actions that were 0:48:41.573 committed or aborted. And you build up a list of 0:48:45.088 losers that were none of these. In other words, 0:48:48.529 they were pending actions that kind of just during a failure 0:48:52.941 they were pending, so they didn't commit. 0:48:55.933 And they were never aborted. 0:48:59 0:49:06 And so the plan now is to make sure that the cell store is 0:49:09.463 correctly restored to the state that was before the crash where 0:49:13.23 all of the committed actions' results are visible, 0:49:16.207 and none of the uncommitted actions, you know, 0:49:18.941 all of those are blown away. All you have to do is to 0:49:22.101 redo the winnders were committed. You don't have to do anything 0:49:25.807 for the aborted winners because they were already undone. 0:49:30 So you have to redo committed winners, and you have to undo 0:49:35.105 any changes made by losers, right, because these losers by 0:49:40.122 definition were things that didn't commit or didn't abort. 0:49:45.139 And the reason you only redo the committed winners rather 0:49:50.068 than all winners is it makes no sense to redo aborted winners. 0:49:55.437 And you don't need to undo them because they were already undone 0:50:00.982 when the abort record was written to the log. 0:50:06 So this is the basic idea for dealing with one of these 0:50:09.145 databases. But there's five or six 0:50:11.067 optimizations that end up making this kind of system go faster. 0:50:14.679 You'll see some of these optimizations buried inside the 0:50:17.883 system R paper, which is the discussion for 0:50:20.33 tomorrow. But what I'll do on Monday, 0:50:22.427 I'll spend five minutes talking about the most important 0:50:25.631 optimizations, and I think the whole story 0:50:28.019 will become clear. So the plan for the subsequent 0:50:31.734 lectures on this topic are: on Monday we'll deal with 0:50:34.742 isolation, and on Wednesday we'll continue to talk about 0:50:37.922 isolation, and then talk about a different issue of

and on Wednesday we'll continue to talk about client-side isolation, and then talk about a different kind of consistency.