PROFESSOR: So a couple of things I want to say about the final project. You guys should start thinking about it. So of course, you guys should think about the teams first, and submit team information. Say who you're going to team up with. By when? Team information, Josh, by when? Team information has to be in--

AUDIENCE: By tomorrow.

PROFESSOR: By tomorrow. OK, good. You should know your teams. Get them together, use the same mechanism that we used before to get the team information before. So we are going to add one small thing this year I think that would be useful. Once you have submitted your design documents, we are going to get a design review done by Masters. So what that means is next week we are going to schedule a design review with your Masters. And you should send mail to your Masters, hopefully in the middle of next week, to schedule a design review. The design review will happen the week after Thanksgiving.

So you'll submit your design doc next week, and then the week after Thanksgiving, you'll schedule your design review. The earlier the better, because you can hopefully get some really good feedback before you go into implementation. So have an idea what you're doing. Of course you're write it up for your design document. And then go to your Masters and say, here's what I'm planning to do. Get some good feedback, and hopefully it will help your life to do that.

And then beforehand, performance only mattered for your grade. We did this absolute grading. This year, we are going to have actually an in-class competition on the final day of class, to figure out who has the fastest ray tracer in the class. And for that we will actually give [? you ?] a little but of a different

[UNINTELLIGIBLE] than what I have given you. And so you don't go too much into really hand coding to that teammate because that might not work.

And so here's something hot off the press. So for the winning team, there's going to be an Akamai prize for the winning team. And this prize includes a celebration/demonstration at Akamai headquarters. You're going to go visit there [UNINTELLIGIBLE] and perhaps show off to their engineers the cool ray tracer you did. And also every team member of the winning team is going to get the iPod Nano. Sorry, guys, last year it didn't happen. First time. So there's a lot at stake. So make sure your program is going to run as fast as you can.

OK, with that, let's get your slides on. So I'd like to introduce Bradley Kuszmaul. Bradley has been at MIT, in and out of MIT, for a long time doing lots of cool stuff with high-performance-- yeah, make the screen bigger-- high-performance computing. He has done some really interesting data structure work, performance optimization work, and stuff like that. And today he's going to talk about an interesting data structure that goes all the way from theory into getting really, really high-performance. Thank you, Bradley. OK, you have the mic.

BRADLEY KUSZMAUL: So I'm going to talk about a data structure called fractal trees, which in the academic world are called streaming B-trees. But the marketing people didn't think very much of that, and so a lot of these slides are borrowed from a company that I've started. So rather than redo that, I'm just going to stick to the terminology "fractal tree." I'm research faculty at MIT, and I'm a founder at Tokutek. And so that's sort of who I am. I'll do a little bit more introduction.

So I have been around at MIT a long time. I have four MIT degrees. And I was one of the architects of the Connection Machine CM-5. And Charles was also one of the architects of that machine. So at the time, that was the fastest machine in the world, at least for some applications. And after getting my degrees and being an architect, I went and was a professor at Yale, and then later I was at Akamai. So I don't know what an Akamai prize is beyond an iPod, but maybe it's like all your content delivered free for a month or something. And I'm now research faculty in the

SuperTech Group, working with Charles. And I'm a founder of Tokutek, which is commercializing some work we did.

A couple years ago, I started collaborating with Michael Bender and Martin Farach-Colton on data structures for that are suited for storing data on disk. And we ended up a bit later starting a company to commercialize the research. And basically, I'll tell you sort of what the background is, and actually go into some technical on the data structure.

So I don't know exactly what you've spent most of your time on, but a lot of high-performance work, especially in academia, focuses on the CPUs and using the CPUs efficiently, maybe getting lots of FLOPS or something. The Cilk work that Charles and I did, for example, is squarely in the category of how do you get more FLOPS, or more computrons out of a particular machine.

But it turns out often I/O is a big bottleneck, and so you see systems that look a little bit like this. You have a whole bunch of sensors somewhere, and the sensors might be something like a bunch of telescopes in an astronomy system. They're sending millions of data items per second, and they have to be stored. And disk is where you have to store large amounts of data, because disk is orders of magnitude per byte than other storage systems.

And then you want to do queries on that data, and you want to look at the data that's recent. So it's not good enough just to look at yesterday's data. You want to know what's going on right now. If your sensor array is a bunch of telescopes, and a supernova starts happening, you want to be able to find out quickly what's going on, so that you can broadcast the message to everybody in the world so they can all point their telescopes at the supernova while it's fresh.

So that's the picture. Another example of a sensor system is the internet, where you have thousands or millions of people clicking away on Facebook, for example. You could view that collection of mice as, abstractly, a bunch of sensors. And so you see it in science. You see it in internet. There's lots of applications. For example, another one would be that you're looking for attacks on your internet infrastructure

in a large corporation, or something.

So trying to reduce this big sensor system to what is the fundamental problem, basically we need to index the data. So the data indexing problem is this. Data is arriving in one order, and you want to ask about it in another order. So typically data is arriving in the order by time. When an observation is made, the event is logged. When the next observation is made, the event is logged. And then you want to do a query. Tell me everything that's happening in that particular area of the sky over the past month. So there's a big transposition that has to be done for these queries, abstractly, is the data's coming in in one order, and you want to sort it and get the data out in another order.

So one solution to this problem that a lot of people use is simply to sort the data. The data comes in. Sort it. Then you can query it in the order that it makes sense. This is basically a simple-minded explanation of what a data warehouse is. A data warehouse is all this data comes in-- Walmart runs one of these in Arkansas. All these events which are people scanning cans of soup on bar codes all over the country out in Walmart stores, all that data arrives in Arkansas, in one location. They sort the data overnight, and then the next morning they can answer questions like what's the most popular food in the week before a hurricane strikes? Because this is the kind of request that Walmart might care about, because they get a forecast that a hurricane's coming, and it turns out they need to ship beer and blueberry Pop-Tarts to the local stores, which are the things that basically you can eat even if power has failed.

The problem with sorting is that you have to wait overnight. And for Walmart, that might actually be good enough. But if you're the astronomer, that application is not so great. So this problem is called the indexing problem. We have to maintain indexes. And traditionally, the classical solution is to use a data structure called a B-tree. So do you all know what a B-tree is? [INAUDIBLE] data structures to algorithms? A B-tree is like a search tree, except it's got some fan-out, and I'll talk about it in a second. They show up in virtually all storage systems today, and they were invented about 40 years ago, and they show up in databases such as MyISAM

or Oracle. They show up in file systems like XFS. You can think of what Unix file systems like EXT do as being a variation of a B-tree. Basically they're everywhere.

Mike might drew this picture of a B-tree. And I said, I don't get it. He said, well, there's a tree, and there's bees. And I said, but those are wasps. So anyway--

So a B-tree looks like this. It's a search tree, so that means everything is organized. It's got left children and right children, and there's actually many children. And like any other search tree, all the things to the left are before all the things to the right. That's sort of the property of trees that lets you do more than just a hash table. A hash table lets you do get and put. But a tree lets you do next. And that's the key observation, why you need something like a tree instead of a hash table.

A lot of database queries-- if you go and click on Facebook on somebody's page, there's all these things that have been posted on somebody's wall. And what they've done when they organized that data is that they've organized it so that each of those items is a row in the database, and they're next to each other so that you fetch the first one, which is like the home page of the person, and then the next and next and next gives each of the messages that they want to display. And by making those things adjacent to each other, it means that they don't incur a disk I/O every time. if it were just a hash table, you'd be having to look all over the place to find those things.

So B-trees are really fast, if you have to do insertions sequentially. And the reason is you have a data structure that's too big to fit in main memory. If the data structure fits in main memory, this is just the wrong data structure, right? If it fits in main memory, what should you use to solve this problem? Any ideas? What data structure is like a B-tree except it doesn't have lots of fan-out? Does anybody know this stuff in this class? Do you people know data structures at all? Maybe I'm in the wrong place. Because it's OK. Just a binary tree would be the data structure if you were doing this in memory, right? A binary tree would be fine. Or maybe you would try to minimize the number of cache misses or something.

So for sequential inserts, if you're inserting at the end, basically all the stuff down

the right spine of the tree is in main memory, and an insertion just inserts and inserts. You have no disk I/Os, and basically it runs extremely fast. The disk I/O is sequential. You get basically performance that's limited by the disk bandwidth, which is the rate at which the disk can write consecutive blocks. But B-trees are really slow if you're doing insertions that look random. The database world calls those high-entropy. And so basically the idea is I pick some leaf at random, and then I have to bring it into main memory, put the new record in there, and then eventually write it back out. And because the data structure is spread all over disk, then each of those random blocks that I choose, when I bring it in, that's a random disk I/O, which is very expensive.

So here, for this workload, unlike the previous workload, the performance of the system is limited by how fast can you move the disk head around, rather than how fast can you write having placed the disk head. And you perhaps you can only, on a disk drive, do something like 100 disk head movements per second. And if you're writing small records that are like 100 bytes or something, you might find yourself using a thousandth of a percent of the disk I/O, of the disk's bandwidth performance. And so people hate that. They hate buying something and only being able to use a thousandth of a percent of its capacity. Right?

New B-trees. Something's wrong with that title. So B-trees are really fast in doing range queries, because basically once you've are brought a block in and you want to do the next item, chances are the next item's also on the same page. So once in while you go over a page boundary, but mostly you just reading stuff very fast.

Oh, I know what this is about. When a B-tree's new and it's been constructed sequentially, it's also very fast. When it gets old, what happens is the blocks themselves get moved around on disk. They're not next to each other. And this is a problem that people have spent a lot of time trying to solve, is that as B-trees get older, their performance degrades. This aging problem-- I saw one report that suggested that something like 2% of all the money spent by corporations on IT is spent dumping and reloading their B-trees to try to make this problem go away. So that's a lot of money or pain or something.

Well, B-trees are optimal for doing lookups. If you just want to look something up, there's an old argument that says, gee, if going to have a tree structure, which is what you need in order to do next operations, then you're going to have some path through the B-tree which is a certain depth, and you do it optimally by having the fan-out be the block size. And everything works. But that argument of optimality is not actually true for insertion workloads. And this is where the data structures work that I've done with Mike and Martin sort of gets to be an advantage.

To see that B-trees aren't optimal for insertions-- here's a data structure that's really good at insertions. What is the data structure? I'm just going to append to the end of a file. Right? So it's great. Basically, it doesn't matter what the keys are. I can insert data into this data structure at disk bandwidth. What's the disadvantage of this data structure?

AUDIENCE: Lookups?

BRADLEY KUSZMAUL: Lookups. So what is the disadvantage? Lookups aren't so good. What is the cost of doing a lookup?

AUDIENCE: Order N?

BRADLEY KUSZMAUL: Order n. Yeah. You have to look at everything. It requires a scan of the entire table. And we'll get into what the cost model is in a second. But basically, you have to look at everything. So it's order n. It turns out the number of blocks you have to read in, which is the thing you care about-- it's order n over b. So we'll get into a performance model in just a second.

So here we are. We have two data structures-- a B-tree, which is not so great at insertions. It's quite good at point queries and quite good at ranged queries, especially when it's young. And we have this other data structure, which is the append data structure, which is wonderful for insertions and really bad for queries. So can you do something that's like the best of all possible worlds? You can imagine a data structure that's the worst of all possible, but it turns out that there are data structures that do well for this. And I'll show you how one works in a minute

So to explain how it works and to do the analysis, we need to have a cost model. And we got into this just a minute ago, with what is the cost model for a table scan? Is it order N? Well, if you're only counting the number of CPU cycles that you're using up, it's order N, because you have to look at every item. But if what you really care about is the number of disk I/Os, then you just count up the number of blocks. And so in that model, the cost is order N over B. And that's the model that we're going to use to do this analysis.

So in this model, we aren't going to care about CPU cost. We are going to care about disk I/O. And that's a pretty good place to design in if you're an engineer, because right now the number of CPU cycles that you get for a dollar is going up. It's been going up. It's continuing to go up. You have to write parallel programs today to get that, but you get a lot of cycles in a $100 package. But the number of disk I/Os that you're getting is essentially unchanged. It's maybe improved by a factor of two in 40 years. So that's the one to optimize for, is the one that's not changing. And use all those CPU cycles, if you can, to do something.

So the model here is that we're going to have a memory and a disk. And you could use this, and there's some block size, B, which we may or may not know what it is. And it's actually quite tricky on real disk systems to figure out what the right block size is. It's not 500 bytes, because that's not going to be a good block size. It might be more like a megabyte. And when we move stuff back and forth, we're going to move a block at a time. We're going to bring in a whole block from disk, and when we have to write a block out, we write the whole block out. So we're just going to count that up.

There's two parameters. There's the block size, B, and the memory size, M. So if the memory is as big as the entire disk, then the problem goes away, and if the memory's way too small-- like you can only have one block-- then it's very difficult to get anything done. So you need to be able to hold several blocks worth of storage.

The memory is treated as a cache for disk. So once we've brought a block in, we can keep using it for a while until we get rid of it. So have you guys done any cache-

oblivious data structures? OK. So you've seen this model. So the game here is to minimize the number of clock cycles and not worrying about the CPU cycles. So here's the theoretical results. We'll start with a B-tree. So a B-tree which has a block size b-- and here I'm going to assume that the things you're storing are unit sized. Because you can do the analysis, but it gets more complicated. So the cost of a lookup, which is the upper right side, is log N over log B. That's the same as-- you may not be used to manipulating these, but usually people write this as log base B of N. But that's the same as log N over log B. And I'm going to write it this way, because then it's easier to compare things.

So if B is 1,000 or something, then basically instead of paying-- just as an example, if N is, say, 2 to the 40th-- let's these all to lg's because it basically doesn't matter. So it's 40 over log base B, and if B is, say, 2 to the 10th, then that means that if you have a trillion items and you have a fan-out of 1,000, it takes you at most four disk I/Os to find any particular item.

An insertion cost is the same, because to do an insertion, we have to find the leaf that the item should have been in, and then put it there. So the append log-- well, what's the cost of insertion? Well, we're appending away, right? And once every B items, we actually have to do a disk I/O. So the cost of an insertion in the append log isn't 0. It's one Bth of a block I/O per object. And the point query cost looks really bad. It's N over B, which we already discussed.

So the fractal tree has this kind of performance. It's log N over-- it's not B, which would be really great. It's something smaller. It's maybe square root of B for the insertion cost. And the lookup cost is log N over something, which I'm going to just hide. Let's set epsilon to 1/2 and work out what that is. Because epsilon 1/2 is a good engineering point. So the insertion cost is log N over B to the 1/2, which is log N over root B. And the other one, the lookup cost-- there's all big 0's around here, but I'm not going to draw those again-- over 1/2-- so I'm going to maybe ignore that-- of the log of the square root of B. Did I do that right? B to the 1 minus 1/2. Put the 1/2 back in to make you happy. So big-O of that-- well, what's log of root B?

AUDIENCE: [INAUDIBLE]

BRADLEY KUSZMAUL: I can't quite hear you, but I know the answer. I can just say that's the same as log B, when I'm doing big 0's Get rid of the halves. So it's log N over log B. So if you sort of choose block sizes, if you set this parameter to be something where you're doing something with the square root, you end up having lookups that cost, asymptotically the same as for a B-tree. But there are these constants in there. There's a factor of 4 or something that I've glossed over. But asymptotically, it's the same.

And insertions have this much better performance. What if B was 1,000? Then we're dividing by 30 here. But B isn't really 1,000. B's more like a million in a modern system. So you actually get to divide by something like 1,000 here. And that's a huge advantage, to basically make insertions asymptotically be 1,000 times faster, whatever that means. When you actually work out the constants, perhaps it's a factor of 100, is what we see in practice. So this is basically working out those details.

So here's an example. Here is a data structure that can achieve this kind of performance. It's a simple version of a streaming B-tree or a fractal tree. And what this data structure is-- so first of all, we're kind of going to switch modes from marketoid, or at least explaining what it's good for, to talking about what a data structure is that actually solves the problem. So any questions before we dive down that path? OK. So if there are any questions, stop me. Because I like to race through this stuff if possible.

So the deal here is that you're going to have log N arrays, and each one is a power of two in size. And you're going to have one for each power of 2. So there's going to be one array of size 1, one of size 2, one of size 4 and 8 and 16, all the way up to a trillion, 2 to the 40th. The second invariant of this data structure is each array is either completely full or completely empty. And the third one is that each array is sorted.

So I'll do an example here. If I have four elements in the array, and these are the numbers, there's only one way for me to put those in that satisfy all those

requirements. Because there's four items, it has to go into the array of size four. It has to fill it up. I can't have any other way of doing that. And within that array of size four, they have to be sorted. So those four elements uniquely go there, and that's the end of the story for where four elements go.

If there's 10 elements, you get a little freedom, because, well, we have to fill up the 2 array and we have to fill up the 8 array, because there's only one way to write in binary 10, which is 0101. But we get a little choice. It turns out that the bottom array has to be sorted and the top array, the array containing 5 and 10 has to be sorted. But we could have put the five down here and, say, swapped the 5 and the 6, and that would've been a perfectly valid data structure for this set of data as well. So we get a little bit of freedom. OK? So that's the basic data structure.

So now what do we do? How do you search this data structure? Well, the idea is just to perform a binary search in each of the arrays. The advantage of this is it works, and it's a lot faster than a table scan. The disadvantage is it's actually quite a bit slower than a B-tree, because if you do the analysis here, which in this class, you probably-- you've done things like master theorem and stuff, right? So you know what the cost of doing the search in the biggest array is, right? How many disk I/Os is that in the worst case?

AUDIENCE: Log N.

BRADLEY KUSZMAUL: It's log N. It's going to be log base 2 of N, plus or minus a little bit. Just ignore all that stuff. I'll just do L-O-G. So what's the size of doing the second-biggest array? What's the cost of searching the second-biggest array? It's half as big, so it's log of N over 2, right? I can't write. So this is log N. This is equal to log of N minus 1. What's the next array? What's the cost of searching the next biggest array? Log of N minus 2-- you add that up, and what's the sum? We don't even need recurrences for this. We could have done it that way, but what's the sum? When we finally get down to 1, and you search the bottom array, you have to do one disk I/O in the worst case. So this is an arithmetic sequence, right? So what's the answer? Big-O. I'm not even going to ask for the-- pardon?

AUDIENCE: Log squared?

BRADLEY KUSZMAUL: Yes, it's log squared, which is right there in green. So basically, this thing is really expensive. Log squared N, when we were trying to match a B-tree, which is log N over log B. So not only is it not log base B, it's log base 2 or something. But it's squaring it. So if you think of having a million items in your data structure, even a relatively small one, log base 1 million is 20. If you square that, that's 400. Maybe you get to divide by 2. It's hundreds of disk I/Os just to do a look up, instead of four. So this is just sucking at this point.

So let's put that aside and see if we can do insertion, since we are doing so badly at [? logging. ?] So to make this easier to think about, I'm going to add another set of temporary arrays. So I'm actually going to have two arrays of each size. And the idea is at the beginning of each step, after doing an insertion, all the temporary arrays are empty. I'm only going to have arrays on the left side that are going to have data in them. So to insert 15 into this data structure, there's only one place to put it. I put it in the one array, if I'm trying to be lazy about how much work I want to do. And it turns out, this is exactly what you want to do. You have an an empty one array, a new element comes in, just put it in there.

Now I want to insert a 7. There's no place in the one array, so I'm going to put it in the one array over on the temp side. And then I'm going to merge the two one arrays to make a two array. So the 15 and the 7 become 7 and 15 here. I couldn't put it there because that array already was full. And then I merge those two to make a new four array. So this is the final result after inserting those two items. Does that make sense? It's not a hard data structure.

So one insertion can cause a whole bunch of merges. Here we have sort of an animation. So here I've laid out the one array across the top, and then the temporary array just under it, and then going down, we have a sequence of steps for the data structure over time. So we have the whole arrays. We have one and the two and the four and the eight array are all full, and we insert one more item, which causes a big carry. So the one creates a two, the two twos create two fours, the two

fours and the eight create two eights, and so forth.

So here you are. You're running. You've built up a terabyte of data. Your insert one more item, and now you have to rewrite all of disk. So that also sounds a little unappealing. But we'll build on this to make a data structure that actually works.

So first let's analyze what the average cost for this data structure is. I've just sort of explained why-- there are some really bad cases where you're doing an insertion and it's expensive. But on average, it turns out it's really good. And the reason is that merging of sorted arrays is really I/O efficient, because the merge is essentially operating on that append data structure. We're reading two append data structures and then writing the answer into another append data structure. And that does hardly any I/O.

So if you have two arrays of size X, the cost to merge them is you have to read the two arrays and you have to write the new array. And you add it all up, and that's order X over B I/Os. Maybe it's 4x over b or something. But big-O of X over B. So the merge is efficient. The cost per element for the merge is 1 over B, because order X elements were merged when we did that. And we get to spread the cost. Sure, we had to rewrite a trillion items when we filled up our disk, but actually, when you divide that out over the trillion items, it's not that much cost per item. And so the cost for each item of that big operation is only 1 over B disk I/Os. And each item only has to be rewritten log N times.

So the total average cost for an insertion of one element is log N over B, which is actually better than what I promised here. But this data structure's going to be worse somewhere else. So this is a simplified version. I'll get to within-- ignoring epsilons and things, it'll be good enough. So does that analysis makes sense? It's not hard analysis, so if it doesn't make sense, it's not because of you. It's got to be because I didn't explain it, because it's too easy to not understand.

So if you're going to build something like this, you can't just say, oh, well, your database is great except once every couple days it hangs for an hour while we resort everything. So the fix of this is that we're going to get rid of the worst case.

And the idea is, well, let's just have a separate thread that does the merging of the arrays. So we insert something into a temporary array and just return immediately. And as long as the merge thread gets to do at least log N moves every time we insert something, it can keep up.

You could actually do a very careful dance, where I insert something, and part of the insertion is I have to move something from this array and something from this array and something from this array, and I can keep everything up to date that way. So it's not very hard to de-amortize this algorithm-- that is, to turn the algorithm from a good average-case behavior to good worst-case behavior. The worst-case behavior just becomes that it has to do log N work for an insertion, which isn't so bad. Does that make sense? Yeah.

AUDIENCE: Does that work if these are [INAUDIBLE] items [INAUDIBLE]? What if somebody wants [INAUDIBLE]?

BRADLEY KUSZMAUL: Ah. Well, OK, so the question-- let me repeat it and see if-- so you're in the middle of doing these merges and you have a background thread doing that, say, and somebody comes along and wants to do a query.

AUDIENCE: Yeah. [INAUDIBLE]

BRADLEY KUSZMAUL: So the trick to there is that you put a bit on the array that says, the new arrays is not ready to query. Keep using the old arrays, which are still there. Just don't destroy the old ones until the new one's ready. So basically you have these two megabyte-sized things. You're trying to make a two megabyte-sized one. You leave the one-megabyte ones lying around for a while while you're incrementally moving things down. And then suddenly, when the big ones done, you flip the bits, so in order one operations, you can say, no, those two are no longer valid, and this one's valid. So queries should use this one instead of those. So that's basically the kind of trick you might do. Or you would just search the partially constructed arrays, if you have locks. There's lots of ways to do it.

So that's a pretty good question. Yes. That's one that we had to think about a little.

So it sounds glib, but it's like, how do we do this? Any other questions? OK.

So now we've got to do something about the search, because the search is really bad. Well, it's not as bad as the insertion worst-case thing. I'm going to show you how to shave off a factor of log N, and I don't think I'm going to show you how to shave off the factor of 1 over log B. so we'll just get it down to log N instead of log squared N. Because if I actually want to get it down, then I have to give up-- remember, the performance that I had was log of N over B. If I actually want to get rid of things, I have to do something else. There's a lower-bound argument.

So the idea here is we're searching-- I'm going to flip those. We've got these arrays of various sizes. And I've just done a binary search on here and then here and then here, and I found out the thing I'm looking for wasn't here and it wasn't here and it wasn't here. That's where it would have been, if it had been there. It should have been there but it wasn't. It should have been here but it wasn't. And then I'm going to start searching in this array. And the intuition you might have is that, gee, it's kind of wasteful to start a whole new search on this array when we already knew where it wasn't in this array. Right?

So for example, if the data were uniformly randomly distributed, and the thing was, say, 1/3 of the array here, I might gain some advantage by searching at the 1/3 point over here to see if it's there. Now, that's kind of an intuition. I don't know how to make that work. But I do know how to make something work. But the intuition is, having done some search here, I should in principal have information about where to limit the search so that I don't have to search the whole thing on the next array. OK?

And here's basically what you do, is every element, you get a forward pointer to where that element would go in the next array. So for example, you have something here and something here, which are the two things that are less than and greater than the thing you're looking for. And it says, oh, those should have gone here in the next array. So if you maintain that information, it's almost enough. But let's gloss over the almost part. If those two destinations of those two pointers are close

together, then you've saved a lot of search in the next array. Does anybody see a bug in this? There is one. The almost part. You don't have to see it, because I've been thinking about this a lot.

The problem is, what if all of these items are less than all of these items, for example? In which case, these pointers all point down to the beginning, and we've got nothing. That's a case where this fails. And that's allowed, right? In particular, if we were inserting things-- yeah.

AUDIENCE: Then we know the element is in the biggest array, because the element was supposed to go between the two.

BRADLEY KUSZMAUL: Ah, but in this array, we found out that it's above the last element, when we did our search, right? That's one of the possible ways-- the worst-case behavior is we've got something where this array is less than this array. We're looking for that item. So we do a binary search and find out, it's over here. And it doesn't help to special case this or something, because they could be all to the right or they could be all bunched up in funny ways. There's lots of screwy ways that this could go wrong. But the simple version, it's easy to come up with an example, which is everything's to the left. Yeah.

AUDIENCE: Can you still save time by, when you do the binary search on the smallest array-- but I guess you'd want [INAUDIBLE] search. It will help reduced cost which gives you the next one and so on?

BRADLEY KUSZMAUL: Yeah. So there is a way to fix it so that the pointers in the smaller array do help you reduce the cost in the next array. And that is to seed the smaller array with some values from the next array. Like, suppose I put in every 20th item, and I stuck it in that array with a bit on it that says, oh, this is a repeat, it's going to be repeated again. So then I could guarantee that there's these dummies that I throw in here, which are evenly spaced, plus whatever else is in there. So put the other things in there, and they have forward pointers too. And now I'm guaranteed that the distance between two adjacent items is guaranteed to be a constant.

Does that make sense? The trick is to make it so that having found two adjacent items that the thing you want-- then on the next array, the image of those two items is separated by at most 20 items. And so that gets you down to only log of N instead of log squared of n, because you're searching constant items in this array, and there's only log N arrays. Yeah.

AUDIENCE: Doesn't that slow down the merging of the rays?

BRADLEY KUSZMAUL: Not asymptotically. Because asymptotically, what this means-- if I'm going to build that array, so I'm going to merge two arrays to make this array, I have to do an additional scan of this other array as I'm constructing this one. So the picture is I have two arrays, and I'm trying to merge them into this array. And I'm trying to also insert these dummy forward pointers from the next array, which is only twice as big. So the big 0's are, if it's X, instead of it being 1, 2, 3, 4X, it's 8X.

So it's only a constant. So basically, I can read all three of these. I can read an array and the next one and the next array, which is twice as big, and the next array which is four times as big. It all adds up to 8 times the size of the original array. So at least the asymptotics aren't messed up. Maybe the engineer in you goes, bleh, I have to read the data eight times.

But remember, the game here is not to get 100% of the disk's insertion capacity. That's not the game, going back to the marketing perspective. The competition is only getting 0.001% of the disk's capacity. That's what a B-tree gets in the worst case. And so we don't have to get 100% to be three orders of magnitude better, which is where we are. So it turns out that for this kind of thing, we end up getting 1% of the disk's capacity, and everybody's jumping around saying that's great, because it's 1,000 times faster.

And why do we only get 1%? Well, there's a factor of two here and there's a log N over there, and you divide all that, and it's a constant challenge, because the engineers at Tokutek always are having ideas for how to make it faster. And right now, making this data structure faster is not the thing that's going to make people buy it. Because it's already 1,000 times faster than the competition. What's going to

make it faster is some other thing that adds features that make it so it's easy to use. So I keep having to-- no, you really need to work on making it so that we can do backups, or something.

It turns out, if you're selling a database, you need to do more than just queries and insertions. You need to be able to do backups. You need to be able to recover from a crash. You need to be able to cope with the problem of some particularly heavy query that's going and starving all the other queries from getting their work done. All those problems turn out to be the problems that, if you do any of them badly, people won't buy you. And so I suspect that there's another factor of 10 to be gotten over this data structure, if you were to sit down and try to say, how could I make it be the fastest possible thing. And someday, that work will have to be done, because the competition will have it and we won't.

So let's see. I mentioned some of these just now. So some of the things you have to do in order to have an industrial strength dictionary are you need to cope with variable-size rows. Now we assumed for the analysis that the rows were all unit size. In fact, database rows vary in size. And some of them are huge. Some of them are megabytes. Or sometimes people do things like they put satellite images into databases. So they end up having very large rows.

You have to do deletions as well as insertions. And it turns out we can do deletions just as fast as insertions. And the idea there is basically, if you want to do a delete, you just you insert the thing with a bit on it that says, hey, this is really a deletion. And then, whatever you get a chance, when you're doing a merge, if you find something that has the same value, you just annihilate it. And the delete has to keep going down, because there might be more copies of it further down that were shadowed. And eventually, when you finally do the last merge, that tombstone goes away.

You have to do transactions and logging. You have to do crash recovery. And it's a big pain to get that right, and a lot of companies have foundered when they tried to move from one mode to the other. How many of you have experienced the

phenomena that your file system didn't come back properly after a crash? You see the difference in age here. They're all using file systems that have transactional logging underneath them. When's the last time it happened?

AUDIENCE: Tuesday.

BRADLEY KUSZMAUL: Tuesday. So the difference is you're paying attention and they're not, right?

AUDIENCE: [INAUDIBLE] disk failure.

BRADLEY KUSZMAUL: Disk failure. That's a different problem.

AUDIENCE: Cacheing is not [? finalized. ?]

BRADLEY KUSZMAUL: Yeah. You say everybody's running with their disk cache turned on. And on some file systems, that's a bad idea. So we're still suffering that it's been difficult to switch from the original Unix file system, which is 30 years old and wasn't designed to recover from crash. You have to run fsck, and it doesn't always work. We still have file systems that don't recover from crashes. So you can see why that could be difficult. It turns out that one common use case is that the data is coming in sequentially, and this data structure just sucks compared to a B-tree in the case where you're inserting things and the data actually is already sorted as it's inserted. Because this is moving things all around and moving things all around. And it's like, why didn't you just notice that it's sorted and put it in?

You have to get rid of the log base B to get it down to log base B of N instead of log base 2 of N for search costs. Because people in fact do a lot more searches than-- if you have to choose which to do better, you want to generally do searches better. And compression turns out to be important. I had one customer who had a database which was 300 gigabytes. He has a whole bunch of servers, and on each server, he had a 300 gigabyte database. And with us, it was 70 gigabytes, because we compress.

And we just do simple compression of, basically, large blocks. When we do I/Os, we do I/Os of like a megabyte. So when we take one of those megabytes, we compress it. And it's a big advantage to compress a megabyte at a time, instead of what-- a lot of B-trees, they have maybe 16 kilobytes. And gzip hardly gets a chance to get anywhere when you only have 16 kilobytes. And it gets down to 12 kilobytes. But if you have a megabyte to work with and you compress it, particularly if it's sorted-- so this is a megabyte of data that's sorted, so compression works pretty well on sorted data. So you get factors of 5 or 10 or something.

And so we asked him to dump the data without the indexes, so just the primary table with no indexes, and then run that through gzip. And it was 50 gigabytes. So the smallest he could store the raw data was 50 gigabytes, and we were giving him a useful database that was 70 gigabytes that had a bunch of indexes. So he was like, yeah.

And you have to deal with multithreading and lots of clients and stuff.

So here's an example. We worked with Mark Callahan, who was at Google at the time-- he's now at Facebook-- on trying to come up with some benchmarks, because none of the benchmarks out in the world do a good job of measuring this insertion performance problem. So iiBench is an insertion benchmark. And basically what it does is it sets up a database with three indexes, and the indexes are random. So it's actually harder than real workloads. This workload, you basically have a row and then you create a random key to point into that from three different places. Real databases, it turns out, probably have more of a Zipfian distribution. Have you talked at all about Zipfian distributions of data?

So this is sort of an interesting thing. If you're dealing with real-world caches, you should know that data ain't uniformly randomly distributed. That's a poor model. So in particular, suppose I have memory and disk, and this is 10% of the disk. Very simple situation. Very common ratio. You'll see, this is how Facebook sets up their databases. They'll have a 300-gigabyte database and 30 gigs of RAM. If the queries that you wanted to do were random, it wouldn't matter what data structure you were

using. Let's suppose that God tells you where it is on disk, so you don't have to find it. You just have to move the disk head and move it.

If they're random, then basically, no matter what you've done, 90% of the queries you have to do are going to do a random disk I/O. 10% are going to already be there, because you got lucky. So that is not reflecting what's going on on any workload that I know. What they'll see is more like 99% of the queries hit here, and 1% go out here, or maybe 95% here and 5% go out there.

And it turns out that, that for a lot of things, there is a model of what's going on called a Zipfian distribution. This would a random uniform distribution. It's like every item has equal probability of being chosen for a query. It turns out that for things like what's the popularity of web pages, or if you have a library, what's the frequency at which words appear in the library-- so words like "the" appear frequently, and words like "polymorphic" are less frequent. So Zipf came up with this model, and there's a simple version of the model, which says that the most popular word has probability proportional to 1. It's not going to be 1. It's going to be proportional to 1. The second most popular word is going to have 1/2. The third most popular word is 1/3. And the fourth most popular word is 1/4 the probability of the first word, and so forth.

So if you plot this distribution, it kind of looks like this, right? It's like 1 over x. And what would you tell me if I told you that I had an infinite universe of objects that had a probability distribution like this. Does that seem plausible? Why? You're saying no.

AUDIENCE: [INAUDIBLE PHRASE] try adding them all together.

BRADLEY KUSZMAUL: If you add them all together, it doesn't converge. So it's a heavy-tailed distribution.

So it turns out that if you sum up these up to 1, the sum from 1 to n of 1 over i, it's the nth harmonic number. And that grows over time. It's basically like the integral under this curve, from 1 to n. It's close to that. And what is the integral of that? It's like something you learned seven years ago, and now you've forgotten, right? You

learned it when you were sophomores in high school, or something. So it's approximately log of n. It's actually like log of n plus 0.57, is a very good approximation for the nth harmonic number.

When you're doing this kind of analysis, boy, it depresses people, because you say, oh, that's H of n, and if you have a million items in your database, then the sum of all those things is H of 1 million, and what's H of 1 million? Well, the log base 2 of 1 million, I know that, because I'm a computer scientist. So it's going to be like 20, and because we're doing log base e in that formula, maybe it's 15 or something. So if you have 1 million items, then the most popular item is going to-- you have to divide by H of n here.

So the most popular item is going to appear 1/15 of the time. And the next most popular item is going to appear-- emergency backup chalk. Somebody's been burning both ends of this chalk. 1/30 of the time, and 1/45 of the time, and those add up. When you go up to 1 over 1 million-- another zero in there-- times 15, that finite series will add up to 1, approximately. Except to the extent that I've approximated.

So this is what's going on. So the most popular Facebook page-- they might have 1 billion pages, so how does that change things? Well, that means the most popular one has a probability 1 in 20, and the second most is 1 in 40. And this explains why cache works for this kind of workload. Nobody really knows why Facebook pages and words in libraries and everything else have this distribution, which is named after a guy named Zipf. But they do. Everything has this property. And so you can sort of predict what's happening.

So iiBench should have a Zipfian distribution and it doesn't. So this is painting a worse picture. Or a better picture. It's making us look better than we really are, because the real world is going to have more hits on the stuff that's in memory for a B-tree than this model, where basically you're completely hosed all the time because it's random. So this is an example in the category of how to lie with statistics. And it's a pretty sophisticated lie. If you're going to lie, be sophisticated.

So these measurements were taken in the top graph. Up is good. It's how many rows per second we could insert. And this axis is how many rows have been inserted so far. And the green one is a B-tree. According to Mark Callahan, who's essentially a disinterested observer, it's the best implementation of a B-tree ever. And you can sort of see what happens, is that as you insert stuff, the system falls out of main memory, and the performance was really good at the beginning-- 40,000 per second-- and then boom, you're down to 200 down here at the end, by the time you've inserted a billion rows.

Whereas, for the fractal tree, you can sort of see this noise. That's because some insertions are a little cheaper than other insertions. Every other insertion's completely free, right? You had a free spot. You just put it in. One out of four insertions, the ones that weren't free, half of them only had to do a little operation in memory. So you see this high frequency noise, because some things are cheaper than others. And that's like a factor of 30 or something.

It turns out it even works on SSD, solid state disk. You might think-- all this time I've been talking about disk drives. Solid state disk has a complicated cache hierarchy inside it, and we were surprised to see that basically we're faster on this workload on a rotating disk than a B-tree is on an SSD, which is orders of magnitude faster in principle, but turns out that for various reasons it's not. One question I get often is, the world is moving away from rotating disk to solid state disk. A lot of applications-- how many of you have solid state disks in your laptops? That's a really good application for a solid state disk, because it's not sensitive to being knocked around. So it's worth it to have a solid state disk even if it were more expensive, which it is. It turns out it's not that much more expensive for a laptop. It's a couple of hundred dollars more or something.

But the advantage of it is that if you go up in an airplane and you're sitting and trying to type in the middle of a thunderstorm, flying across-- it doesn't care. Disk drives, if you do that-- disk drives do not like flying at high altitude, because they work by having a cushion of air that the head is flying on, and in airplanes, which pressurize the cabin to the same altitude as 8,000 feet, that's half an atmosphere. So there's

only half as much air keeping it off. So if you travel a lot, that's when your disk drive will fail, is when you're flying.

OK. So it looks like, however that rotating disk is getting cheaper faster then solid state disk is. So rotating disk is an order of magnitude cheaper per byte than solid state disk today. Maybe two orders of magnitude cheaper. It's hard to measure fairly. But rotating disk, according to Seagate-- they're saying, by the end of the decade, we'll have 70 terabyte drives that are the same form factor. And so you figure out what the Moore's Law is for that, and it's better than for lithography. Lithography is not going to be that much more dense in that timeframe. So at least for the next 5 or 10 years, it looks like disk drives are going to maintain their cost advantage over solid state storage, and maybe even spread that cost advantage.

So for any particular application, for storing your music, SSD will be cheap enough, but for those people that have really big data sets, like these new telescopes they're putting up-- these new telescopes are crazy. These people are putting up these telescopes. They're putting up 1,500 telescopes across the Australian Outback. And each of those telescopes in the first 15 minutes live is going to produce more data than has come down from the Hubble, total. And there's just no way for them to-- I don't know what they're going to do. But it's a huge amount of data, and they're going to have to use disks to store whatever it is that they want to keep. And they don't like throwing away data, because it's so expensive to make. So if I were a disk maker, I'd make sure that my salesmen had an office somewhere out there.

So the conclusion is you're not going to be able to, at least for those applications, just have an index in main memory. You're going to have to have a data structure that works well on disk.

The speed trends-- well, seek time is not going to change. It hasn't changed. It's not going to change. The bandwidth of a disk drive grows with the square root of its capacity. So if you quadruple the storage on the disk because you've made the bits twice as dense in each dimension, then one spin of the disk sees twice as many disks, not four times as many disks. So that projects out to something like disks that

are 500 megabytes per second. So how long is it going to take to back up a 67 terabyte disk drive?

So there remain systems problems. And I was explaining to my son that there's all these problems in systems. Data structures aren't suited, and all these systems suck. He said, well, isn't that horrible if you're computer scientist? I said, no, because we make our living off of these problems. So here are some problems. There's plenty of living to be made yet.

Power consumption is also a big issue for these things. If you fill up a room a Google data center, a room which is probably bigger than this room, full of machines. The Facebook data center is probably a room about this size, full of machines. And power and cooling is something like half the cost of the machines. The machines for something like Facebook, the hardware might cost them $10 million or $20 million a year, and the heating and cooling is another $10 million or $20 million a year, which is why they go off and they build these data centers in places like North Carolina, where I guess they're willing to give them power for free or something. So making good use of disk bandwidth offers huge power savings, because basically you can use disks which are cheaper than solid state for power. And you want to use that well.

CPU trends. Well, you've probably talked about this, right? CPUs are going to get a lot more cores. I actually have a 48-core machine that cost $10,000 that I bought about a month ago. And our customers mostly use machines that are like $5,000 machines. So when I provisioned this machine, I said, well, I should spend and buy a machine that's twice as good as what they're buying, because I'm developing software that they're going to use next year. So I bought a $10,000 machine, which is 48 cores. And we're having all sorts of making a living with that machine. The memory bandwidth and the I/O bus bandwidth will grow. And so I think it's going to get more and more exciting to try to use all these cores. Fractal trees have a lot of opportunity to use those cores to improve and reduce the number of disk I/Os.

So the conclusion is, basically, these data structures dominate B-trees

asymptotically. And then B-trees have 40 years of engineering advantage, but that will evaporate eventually. These data structures ride better technology curves than B-trees do, and so I find it hard to believe that in 10 years that anybody would design a system using a B-tree, because how do you overcome those advantages. So basically all storage systems are going to use data structures that are like this, or something else.

There's a whole bunch of other kinds of indexes that we haven't attacked, things like indexing multi-dimensional data or indexing data where you have very large keys, very large rows. Imagine that you're trying to index DNA sequences, which are much bigger than a disk block. So there's a whole bunch of interesting opportunities. And that's what I'm working on.

So any questions or comments? Arguments? Fistfights? OK.

AUDIENCE: Where's the mic?

BRADLEY KUSZMAUL: Where is the mic?

AUDIENCE: That's OK. I can [INAUDIBLE].

BRADLEY KUSZMAUL: It's on my coat.

PROFESSOR: So actually, this is a very interesting point, because if you think where the world is leading, I think that big data is something that's very, very interesting, because all these people are gathering huge amounts of data, and they're storing huge amounts of data. And what do with data, accessing them, is going to be one big problem. I mean, if you look at what people like Google are doing, they're just collecting all those. Nobody's throwing anything out. And I believe if you to kind of look at them, analyze them, do cool things with the data, it's going to be very, very important.

So I think that would be very interesting, high-performance end. It's not just doing

number crunching. Until now, when people look at high-performance, it's about CPU. It's about how many FLOPS per second can you do? TeraFLOPS, petaFLOP machines and stuff like that. But I think one thing that's really interesting is it's not petaFLOPS. How many terabytes of data can you process through to find something.

BRADLEY KUSZMAUL: So I was at a talk by Facebook, and they serve 37 gigabytes per data per second out of their database tier. And that's a lot of serving. Out of one little piece of whatever they're doing. Those guys have three or five petabytes. And in the petabyte club, they're small potatoes. There's people who have hundreds of petabytes, people with three-letter acronyms.

PROFESSOR: I mean, some of those three-letter acronym places, the amount of data they are getting and they are processing is just gigantic. And I think to a point that even some of the interesting things about-- if they keep growing their data centers at the rate they keep growing in the next couple of decades, they will need the entire power of the United States to power their data centers, because they are at that kind of thing at this point. Even in these big national labs, the reason they can't expand is not that they don't have money to buy the machines, but they don't have money to pay for the electricity, and also they don't have electricity-- that much electricity-- [UNINTELLIGIBLE] them to basically feed it.

BRADLEY KUSZMAUL: I've run into people for whom the power issue was a big deal. I look at it and say, eh, you bought a $5,000 machine. You spend $5,000 in power over the lifetime of the machine. It doesn't seem like it's that big a deal. But they've filled up their data center, and the cost of adding one more machine has a huge incremental cost, because they can't fit one more in. So that means they have to build another building. And so almost everybody's facing that problem who's in this business. And then they try to build a building somewhere where there's natural cooled-- Google's written these papers about, oh, it turns out if you don't air condition your computers, most of them work anyway. So, well, air conditioning is a quarter of the cost over the lifetime of the computer. So if you can make more than 3/4 of them give you service, you come out ahead.

GUEST SPEAKER: On that note, MIT is part of a consortium that includes Harvard, Northeastern, Boston University, and University of Massachusetts Amherst, to relocate all of our high-performance computing into a new green data center in Holyoke, Massachusetts. So the idea is that rather than us locating things here on campus, where the energy costs are high and we get a lot of our energy from fuels that have a big carbon footprint, locating it in Holyoke-- they have a lot of hydro power and nuclear power there. And they're able to build a building that is extremely energy-efficient. And it turns out that a bunch of years ago when they were digging up the Route 90, the Mass Pike, they laid a lot of dark fiber down the length. And so what they're going to do is light up that fiber, which comes right back here to the Boston area.

And so for most people who are using these very high-performance things, it doesn't really matter where it's located anymore, at the level of that. So instead of just locating some piece of equipment here, we just will locate it out there, and the price will drop dramatically. And it'll be a much greener way for us to be doing our high-end computing.

Yeah, question?

AUDIENCE: Isn't someone talking about water-cooled offshore floating data centers?

GUEST SPEAKER: Sure. Sure. So the question is, are people talking about water-coooled offshore floating data centers. Yeah. I mean, locating things in some area where you can cool things easily makes a lot of sense. Usually, they tend to want those near rivers rather than in the middle of the ocean, just because you get the hydropower. But even in the ocean, you can use currents to do very much the same kind of thing. So for some of these things, people are looking very seriously at a whole bunch of different strategies for containing large-scale equipment.

PROFESSOR: So one that's very counterintuitive is people are trying to build data centers in the middle of deserts, when it's very hot. I mean, why do you think people want to build a data center in the middle of the desert?

AUDIENCE: Solar power?

PROFESSOR: Solar power is one thing. No, it's not solar power.

AUDIENCE: It gets really cold at night.

PROFESSOR: No, it's not really cold at night. That's not it.

GUEST SPEAKER: Cheap property.

PROFESSOR: No, the biggest thing about cooling is either you can do air conditioning, where you're using power to pull heat out, or you can use just water to cool. And what happens is most of other places is the humidity is too high. And when you go to the desert, humidity is low enough that you can just pump water through the thing and get the water evaporation going by, and then use that to cool the system. So sometimes they're looking at data centers in places where it could be 120 degrees, but very low humidity. And they think that is a lot more efficient to cool than that. So there are a lot of these interesting nonintuitive things people are looking at. So what [UNINTELLIGIBLE] they say is that humidity's the killer, not the temperature.

AUDIENCE: What if you're located on the South-- if you're located on the South Pole, then that's both cold and really low humidity.

GUEST SPEAKER: Yeah. I mean it'll be interesting to see how these things develop. It's a very so-called hot topic these days, is energy for computing. And the energy for computing of course matters also not only at the large scale but also at the small scale, because you want your favorite handheld to use very little battery. So your batteries last longer. So the issue of energy, using that as a measure-- we've mostly been looking at how fast we can make things run in this class, but many of the lessons you can use to say, well, how can I make this run as energy-efficient as possible?

And what you'll learn is that many of the lessons we've had in the class during the term we focused, as I say, on performance. But there are many resources in any given situation that you might want to optimize. And so understanding something about how do I minimize energy, how do I minimize disk I/Os, how do I minimize

clock cycles, how do I minimize off-chip accesses-- which tend to be much more energy intensive than on-chip-- all those different kinds of measures end up being part of the mix of what you have to do when you're really engineering these systems.

PROFESSOR: So I think another interesting thing is, because we are in this time where some stuff grows at exponential rates and stuff like that, some of those ratios that made sense at some point just suddenly start making really bad things. Like, for example, in this, at some point the seek times were normal enough that you didn't care. And at some point, because the rest of the things took off so fast, suddenly it becomes this really, really big bottlenecks.

BRADLEY KUSZMAUL: B-trees were a really good data structure in 1972. Because, well, the seek time and the transfer time and the CPU-- the CPUs actually couldn't read in the data in one rotation, so people didn't even read consecutive blocks, because the CPU just couldn't handle data coming in that fast. You would stagger blocks around the disk, so that when you did sequential reads, you'd get this one and then this one and this one. There was this whole thing about tuning your file system. It's like--

AUDIENCE: By the way, back when disks were--

BRADLEY KUSZMAUL: Yeah. Washing machine.

AUDIENCE: Washing machine size.

BRADLEY KUSZMAUL: For 20 megabytes.

PROFESSOR: Oh, yeah, that's the big disk.

So hopefully you guys got a feel for-- we have been looking at this performance on a small multi-core and stuff like that, how it can scale in different directions and the kind of impact performance can have. And in fact, if anybody has read books on why Google is successful, one of the biggest things for their success is they

managed to do a huge amount of work very cheaply, because the amount of work they do, if anybody did in the traditional way, they can't afford that model, to give it for free or give it supplied for advertising. Because they can get it done because it's about optimization. Performance, Performance basically relates to cost. And if the cost is low enough, then they don't have to keep charging a huge amount of money for each search.

GUEST SPEAKER: So let's thank Dr. Kuszmaul for an excellent talk. And can you hang out for just a little bit, if people want to come down? OK. Thanks.