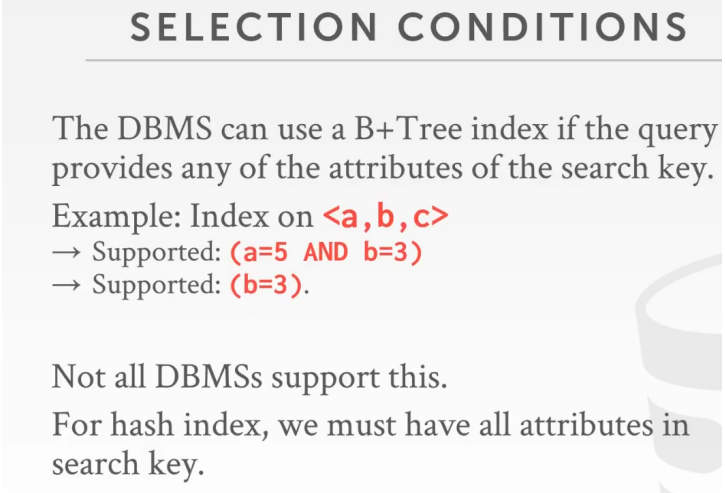
07–03

40:35 – 40:38
So this is useful for certain things like
So，这对于某些事情会比较有用
40.38–40.41
you know if I'm doing a lot of lookups within is that range is the primary key
比如，如果我要进行一大堆根据主键来进行范围查找的任务
40:41 – 40:45
If I know my tuples are stored in the same order that primary key
如果我知道我的tuple所保存的顺序是跟主键一致
40:45 – 40:50
Now when I you know traverses the leaf node within a small number of pages, I can find all the data that I need
~~当我在一小部分page中遍历叶子节点时，我可以找到所有我需要的数据~~
我只需要遍历某个叶子节点下的所包含的一小部分pages，就可以找到所有我想要的数据
40:51 – 40:54
If I'm not sorted on the key I'm doing my lookup one
~~如果我所查找的key并进行排序~~
如果我所根据查找的这个key并没有进行排序（知秋注：即没有根据该key做索引）
40.54–40.58
then every single record id I could have could point to another page
那么我所拿到的每个单个record id（代表table中每条数据）都有可能指向另一个page
40:58 – 41:00
And I could be doing a bunch of different random IO`s to go read the data that I want
~~那我可能就得做很多随机I/O，以此来读取我想要的数据~~
那么我在读我想要数据的时候，就要进行大量随机I/O
41:02 – 41:05
So not all database system supported this
So，并不是所有的数据库系统都支持这个
41.05–41.07
some database system you get this by default
某些数据库系统会默认使用它
41.07–41.11
like MySQL by storing the tuples in the leaf–nodes themselves
比如，MySQL将tuple保存在叶子节点上
41:11 – 41:13
it is a clustered index
这就是聚簇索引（clustered index ）
41:13 – 41:18
So it's guaranteed to have on the pages on disk, the tuples are sorted in the primary key order

So，这保证磁盘上page中的tuple都是以主键顺序来排序的（知秋注：聚簇索引会将tuple也保存在索引中，所以这里说的page是索引在磁盘中的保存）
41:18 – 41:22
In case MySQL if you don’t define a primary key, they’ll make one for you
在MySQL中，如果你没有定义主键（primary key），那MySQL会帮你定义一个
41:23 – 41:26
Right, they’ll have a synthetic like row ID, a record ID that’s transparent to you
它们会使用row id或者record id之类的东西作为主键，对于你来说，它们是透明的，你是看不见它们的
41:27 – 41:31
But that’s how they use to to figure out you know what where your tuple is actually located
但这就是它们用来弄清楚你tuple实际位置的方法
41:32– 41:33
Case of Postgres
至于PostgreSQL，
41.33–41.35
we can do a demo next time
我们会在下节课中，对其进行演示
41.35–41.37
but they have clustered indexes
但它们使用聚簇索引（clustered index）
41.37–41.41
you can define one you say cluster my table on this index
对于聚簇索引，你可以这样说，请把我的表聚集在这个索引之上
41:41 – 41:43
But it won’t actually maintain it in that order
但实际上PostgreSQL并不会按照这个顺序来进行维护
41.43–41.50
meaning does the sorting once stores on disk, but then over time it can get out of order
这也就是说，磁盘上所保存的表中的数据一开始是排好序的，但随着时间的推移，它就乱序了
41:50 – 41:52
Because I won’t do it for you automatically
因为我不会自动帮你做这个
41:52 – 41:56
And when we talk about multi version to concurrent issue it’ll become very clear why this is the case for them
当我们讨论在并发时所遇到的多版本问题，我们就能弄清楚其中的原因是什么了



41:59 – 42:03

So let's talk well how we can do some lookups on the our B+tree

So，现在我们来讨论下我们该如何在B+ Tree上进行查找

42:04 – 42:06

So again because things are in sorted order

So，再说一遍，因为所有东西都是排好序的

42.06–42.12

you know we can do fast traversal to find I think we're looking for

我们可以通过快速遍历来找到我们正在查找的东西

42:12 – 42:15

But ~~we made~~ one advantage you can do with a B+tree that you can't do with a hash table

其中有一个优势，我们可以在B+Tree中所使用，但我们在hash table中无法使用

42.15–42.20

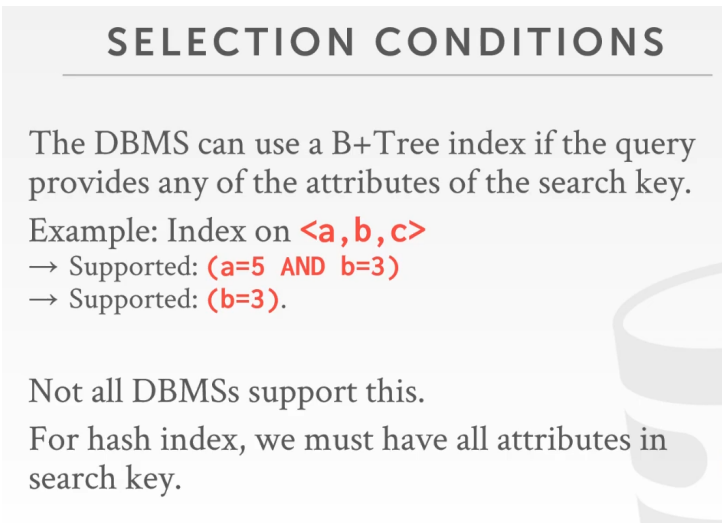
it`s that you don't need to have the exact key in order to do a lookup

那就是我们无须通过一个具体的key来进行查找

42:20 – 42:22

You can have actually some part of the key

实际上，我们可以使用这个key的部分内容来进行查找（知秋注：比如like）



42:23 – 42:26

So the save real simple simple table I have an index on attribute <A,B,C>

So，在一个很简单的表上，我在属性<a,b,c>上进行索引

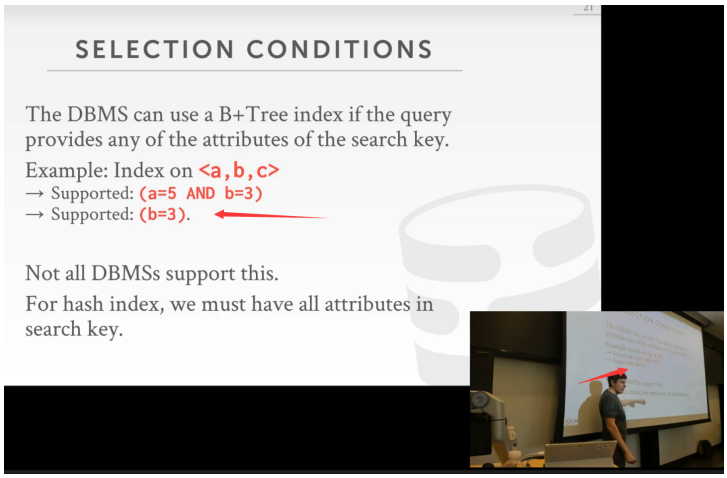
42:27 – 42:28

So I can do lookups like this

So，我可以像这样进行查找

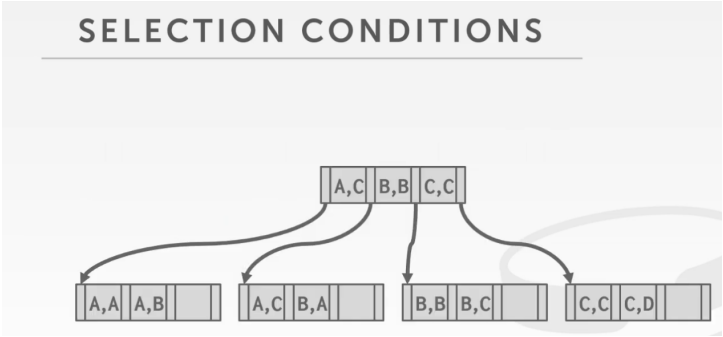
42.28–42.34

where a = 5 and b=3 where I have I don't have the C, but I have a and B
即条件为(a=5 AND b=3)，这里我并没有用到c，但我用到了a和b
42.34–42.37
and I don't need to have the C, and I can still find the things that I'm looking for
我不需要用到C，但我依然能找到我所查找的东西
42:37 – 42:39
You can't do that in a hash index
~~我们不能在hash索引中做这种事情~~
在hash索引中你是做不到这种事情的
42.39–42.40
because think what happened
思考下这会发生什么
42.40–42.44
I would take this 5 and 3 try to hash them together without the c
在不使用c的情况下，我试着将5和3一起进行hash处理
42.44–42.48
and that's gonna jump to some random location that's not just not what I'm looking for
然后，这会跳转到某个随机位置上，然而这并不是我想找的那个地方
42:49 – 42:52
You can also do queries where you only have maybe the middle guy
当你只使用中间那个属性b时，你也能进行查询
42:53 – 42:56
Right, you don't have the prefix, you dont have the suffix, you just have the middle the middle key
我们不使用前缀（a），也不使用后缀（c），我们只使用中间的b
42:57 – 42:58
Again,you can't do that in a hash table
再说一遍，我们没法在hash table中做这种事情
42:58 – 43:00
So not all database system support this
并不是所有的数据库系统都支持这个
43.00–43.08
pretty much everyone supports the prefix one where you have at least the keys in the order as they're defined for the index
许多系统都支持使用前缀进行查找，至少这些key是按照索引所定义的顺序进行排列的

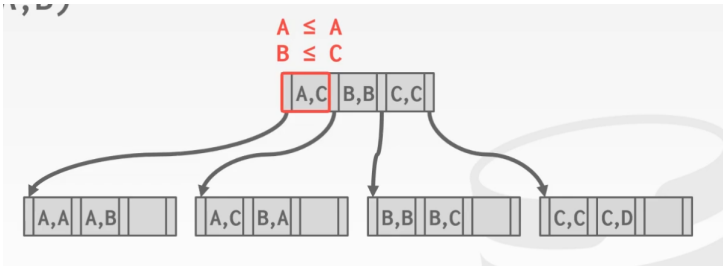


43:08 – 43:10
Now everyone can do this middle one here
所有的系统都支持使用中间这个key进行查找

43.10–43.13
actually I think maybe only Oracle , SQL server can do this
实际上，我觉得只有Oracle和SQL server能做到这点
43:14 – 43:15
So that's a little more concrete example
So，我们来看些更为具体的例子

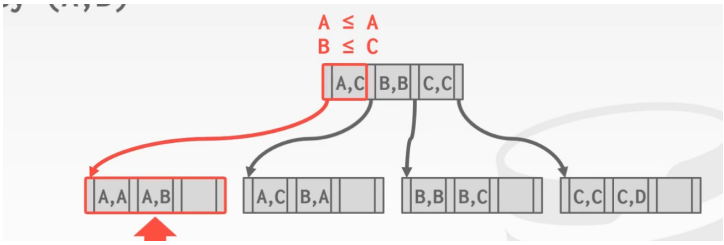


43:16 – 43:24
So let's say we have an index that is defined on two columns or two attributes
So，假设我们在两个列或者两个属性上定义一个索引
43:24 – 43:26
So this would be called like a composite key
So，这被称为复合键（composite key）
43:26 – 43:29！！！！！！！
So instead of being on for one column, it's actually two columns combined
So，我们并不是在一列上定义索引，实际上是使用两列一起来进行索引定义
43:29 – 43:35
And the order of how we define our index will determine what kind of queries we can do on them
我们所定义的索引的顺序决定了我们所能做的查询类型
43:36 – 43:40
So again if I'm trying to do look up on say, it's a trying to find key (A,B)
So，如果我试着查找key(A, B)
43.40–43.44
 well in that case I have both attributes that I've defined in my key
Well，在这个例子中，我在我的key中定义了两个属性

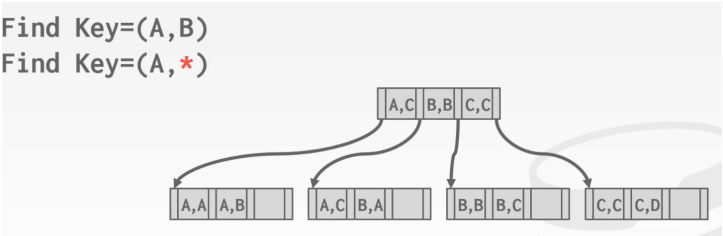


43:45 – 43:47
So now I can just do a straight comparison of look at the first key
So，现在我可以先直接对第一个key进行简单比较一下
43.47–43.49
and then look at the second key
然后，再对第二个key进行比较
43.49–43.50
 and then determine whether I want to go left and right
接着，以此来决定我是该向左走还是向右走
43:51 – 43:53

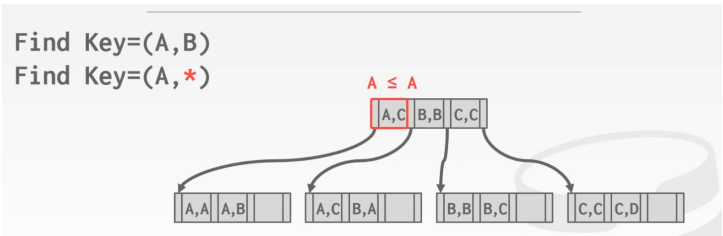
So in this case here a is less than equal to A and B is less than equals C
So，在这个例子中，A小于等于A，B小于等于C



43:54 – 43.57
So I know to find the key and I'm looking for
So，这样我就知道该如何找到我要找的key了
43.57–44.00
I go down this path do whatever search I want to do in my node
我沿着这条路线往下，在我的节点中做我想做的任何搜索（二分搜索，循序搜索之类）
44.00–44.03
and then I can find the entry that I want
然后，我就能找我想要的那个条目了



44:03 – 44:06
Let's say though now I want to do a prefix search
假设，现在我想进行前缀搜索
44.06–44.09
where I only have the first element to my composite key, but not the second one
我现在只有复合键（composite key）中第一个元素，但我没有第二个元素



44:09 – 44:14
So again I can just look at the first key or first attribute of the key, A is less than equal to A
So，现在我可以看下这个复合键中第一个属性，这里A小于等于A
44:14 – 44:20
So I know that the starting point for what I'm looking for, it has to be down in this direction, so I go down here
So，我知道此处的(A,C)对于我所要找的东西来说是一个起点，它应该位于这个方向的下面，因此，我要往下去进行查找
44:21 – 44:24
But now I'm gonna do a sequential scan across my node
现在我要跨节点进行循序扫描
44.24–4433
and going across the the leaves to find all the entry I want up until I reach a key, that is less than or equal to the you know my key A

通过跨叶子节点去找到所有我想要的条目，直到遇见大于等于我复合键（Composite key）中的A为止，我才停止搜索

44:33 – 44:37

So in this case as soon as I find one that start to B, I know my search is done

在这个例子中，一旦我遇到了key中第一个属性等于B，我就知道我的搜索完成了

44.37–44.42

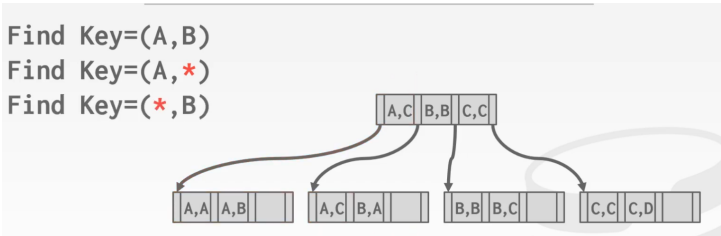
and there's not gonna be anything else remaining in the leaf nodes that would satisfy my predicate

也就是说，在我的叶子节点中不再有任何满足我条件的元素了

44:42 – 44:47

So this one's pretty easy or not easy, but a lot of database systems can support this one

So，这种方式可能简单，也可能不简单，但许多数据库系统都支持这个



44:47 – 44:48

The hard one is this

真正难的是这种

44.48–44.52

where you only have you only had the last element not the first one

即我们只有复合键（composite key）中最后一个元素，而不是第一个元素

44:53 – 44.56

So the way you actually end up implementing this is

实际上，我们最终实现这种的方法是

44.56–45.04 ！ ！ ！

you try to figure out at least in the top and in the root node, which portions of the tree do I need to look at

~~我们要试着在根节点处弄清楚树的哪一部分，才是我们需要看的~~

我们要试着在根节点处弄清楚我们需要去查看该树的那一部分

45:04 – 45:06

It could be something that there's something could be there

在那一部分中，可能有我们要找的东西

45:06 – 45:11

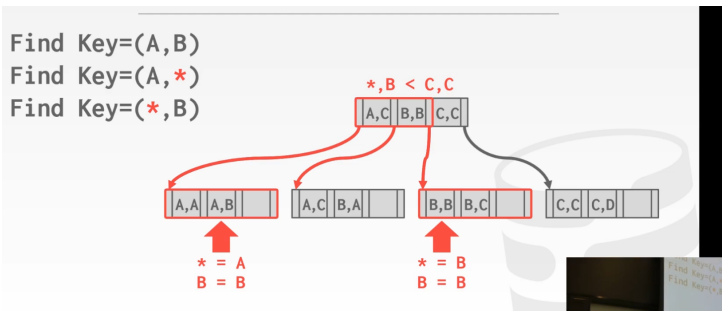
So in this case here, I know that no matter what I have for the first value

So，在这个例子中，我知道，不管我找的第一个元素的值是什么

45:12 – 45:17

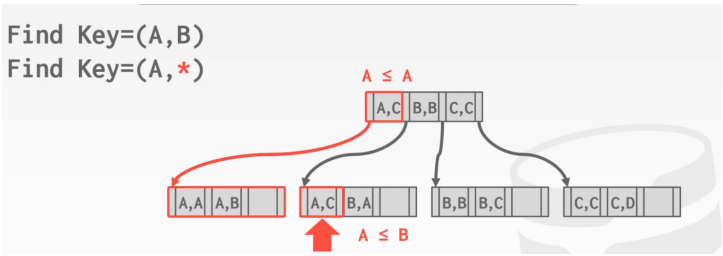
It's always gonna B less than C for the second attribute that's the second value

复合键（Composite key）中第二个元素B的值始终小于C



45:17 – 45:20

So I don't need to look at this guy over here, I only need to look at these other ones
So，我无须看这里的东西，我只需要看另一边的其他东西即可
45:21 – 45:29
So essentially what you just do is you end up doing multiple index probes or multiple traversals, and substituting different values for the thing that you don't have
So，也就是说，我们所做的就是使用多个索引探针或者是进行多次遍历，使用多个不同的值来替换我们所没有的东西（这里指的是*，通过替换它来进行查找）
45:30 – 45:31
So we look at the top and say
So，我们看着根节点时会说
45.31–45.33
well I know I have an A ,I have a B ,and I have a C
Well，这里我有一个A，一个B和一个C
45.33–45.37
well there's nothing for this C that would find over here ,so I can skip that
Well，我们在C那部分中找不到我们要找的东西，So，我可以跳过它
45:37 – 45:40
So let me now do a lookup in these guys
So，现在我在这两个部分（A和B）中进行查找
45.40–45.44
and I substitute the star with an A and each one of those is a separate lookup
我用一个A将这个*进行替换，每个查找都是一个独立的查找
45:44 – 45:47
And then you combine them all together and produce the final result
然后我们将A和B的查找结果合并在一起，并生成最终结果
45:48 – 45:51
So Oracle calls this skip scans I don't know what other systems call, yes
So，Oracle将其称为skip scan，我不知道其他系统将它称为什么。请问
46:07 – 46:11
Yeah, yes yes you're right, that's wrong
你说的没错，这里我写错了（这里没把C包括进去）
46:12 – 46:17
But yes, so you would include that, but it's each one of those use it separately to traversal
我应该将C这部分也包括进去，但这里每部分都应该单独遍历一下
46:17 – 46:19
Okay, and you're just filling in the values
Ok，你所做的就是用不同的值去替换这个*即可
46:19 – 46:21
Whereas like in this one here,
然而在这里



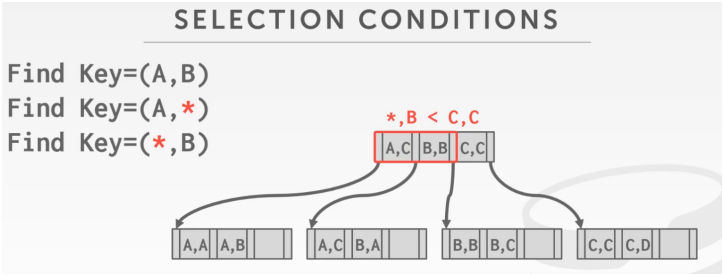
46.21–46.24
the main point I make is like this one, he like for this one and the first one

对于这个例子和第一个例子而言，它的重点在于

46.24–46.27

I had to do one traversal, and then I found the thing I was looking for

我只需做一次遍历，然后我就能找到我要的东西



46:27 – 46:30

This one is you have to probe down multiple times ,and you fill in the values

对于这里，我们必须将*替换成不同的值，并对数据进行多次遍历才行

46:31 – 46:32

Thank you I'll fix that

感谢你指出这个问题，我会将它修复的

46:35 – 46:38

Okay, so let's get to the good stuff

Ok，我们来看个好东西

46:38 – 46:40

So we know what a B+tree is now

So，现在我们知道了B+ Tree是什么了

46:40 – 46:43

Let's talk about actually how you want to build it it makes it this thing actually useful

现在让我们讨论下，我们该如何构建它，并让它变得非常有用

46:44 – 46:51

So there's this great book which I think is free list ,if you google it it shows up free, I don't know whether that's true or not

So，图上所展示的是一本非常好的书，如果你在谷歌上搜索它，你们应该能免费观看，虽然我并不确定这是真是假

46:51 – 46.54

There's great book written a few years ago by Goetz Graefe

这是有Goetz Graefe在几年前所编写的一本好书

46.54–46.56

who's a famous database researcher

他是一位非常著名的数据库研究人员

46.56–47.00

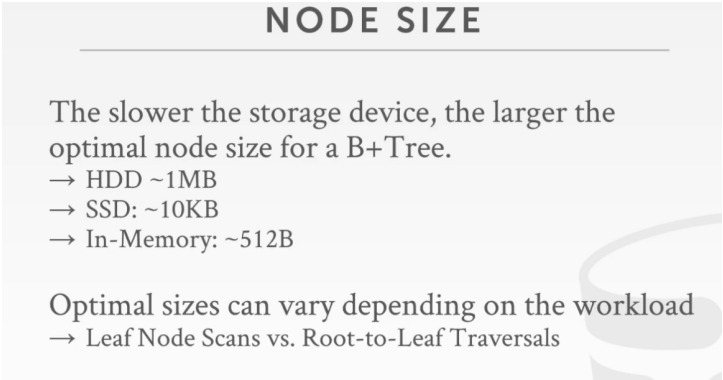
he's gonna talk about a lot of this stuff he's done for query optimization later on

他之后会对他所做的查询优化相关工作进行大量讨论

47:00 – 47:07

But he basically he wrote this book is like all the modern techniques and peaks and optimizations you can do in a B+tree in a real system

但基本上来讲，他在这本书中所写的所有现代技术和优化方法，你们都可以在真实的系统中或者是B+ Tree中进行使用
47:08 – 47:13（这段话有屏蔽词）
So we're gonna cover some of these things and actually it's a really light read ... and it was
So，我们会对其中部分内容进行介绍
47.13–47.17
and like it covers all the really important topics, and in a way that's easy to read
书中介绍了所有很重要的话题，并且对于我们来说很容易去理解
47:17 – 47:25
So which I had a handle Node Size, how to do Merging, how to hand Variable Length keys the Non–Unique keys what they asked about and then Intra–Node Search,how to do better searches inside the node
这里面介绍了，如何处理node size，如何进行合并，如何处理可变长度的key，以及非唯一键，接着就是Intra–Node搜索，以及如何更好地在节点中进行搜索

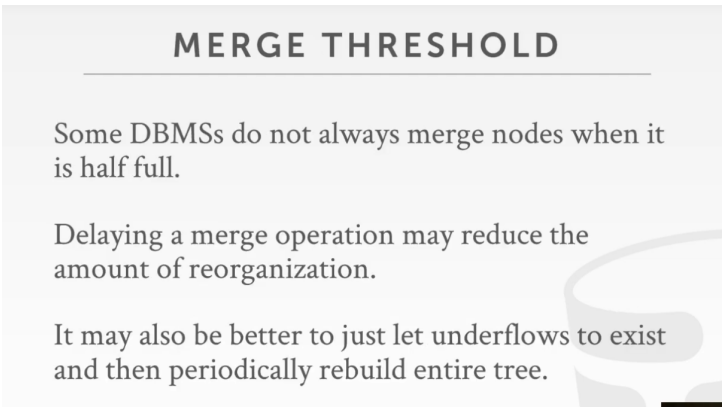


47:26 – 47:34
So in general the you can think of a node in our B+tree, it's just a like a page in our table
So，一般来讲，你可以将B+Tree中的一个node当做我们表中的page来思考
47:34 – 47:37
Right, so the size of the node could be the same as a page size
So，一个node的大小可能等同于一个page的大小

47:38 – 47:40
In practice though it doesn't have to be
在实战中，尽管我们无须这样做
47.40–47.43
and depending on what kind of hardware we're storing our database on
取决于我们的数据库所存放的硬件类型
47.43–47.50
we actually may want to have even larger page sizes or smaller node sizes are smaller node sizes
实际上，我们想要的可能是更大的page或者是更小的node
47:50 – 47:52
So it turns out the research shows that
So，根据研究表明
47.52–48.00
the slower the disk you have, you're just drawing your index on your tree on the larger the node size you want
如果你使用的磁盘速度很慢，那么当你在构建你树上的索引时，你会希望你的节点大小会更大一点

48:00 – 48:01
And you know it should be obvious, right
显而易见
48:02 – 48:03
The for every disk I/O I do
对于我所进行的每次磁盘I/O来说
48.03–48.08
I'm bringing I can read the the nodes sequentially all the pages for it
我可以按顺序读取节点上的所有page
48:08 – 48:14
And that's gonna be much faster than you went to random I/O two different different nodes, if my node is is a smaller size
~~如果我的节点的大小更小，那么就要比在两个不同的节点上进行随机I/O速度来的更快~~
如果跳到不同节点间随机I/O的速度非常快，你的节点就可以使用更小的size(知秋注：其实看的是寻址的速度，内存大于固态硬盘大于机械硬盘，寻址不需要加锁，但读和写都是要有锁存在的)
48:15 – 48:16
So if you're in a spinning hard drive
So，如果你是用的是一个机械硬盘
48.16–48.19
you have node sizes up to one megabyte that's usually a good number
~~你所使用的节点的大小最多是1MB，通常情况下，这就很好了~~
通常情况下，你的节点大小为1MB就足够了
48:20 – 48:22
SSDs roughly 10 kilobytes
如果是SSD，那么10kb左右就可以了
48.22–48.27
which roughly corresponds to the node sizes or page sizes that real database systems use
这大致对应了在真实数据库系统下所使用的node大小或page大小
48:27 – 48:30
But then if you're an in–memory database, you actually want to go low as 512 bytes
但如果你使用的是内存型数据库，那么你所使用的node大小或page大小只需要512 bytes就够了
48:31 – 48:36
And so the this is another good example what we talked about
So，这是我们所讨论的另一个很好的例子
48:36 – 48:43
how in our buffer pool We could have one buffer pool our system for index pages, and one buffer pool set for data pages
在我们的buffer pool中，我们可以使用一个buffer pool来管理我们的index page，另一个buffer pool用来管理data page
48.43–48.44
and we could set them to be different sizes
我们可以将它们设置为不同的大小
48:44 – 48:52
So I could set if I'm going to slow spinning this hard drive, and I can have a buffer pool ,my B+tree pages and have them be one megabyte

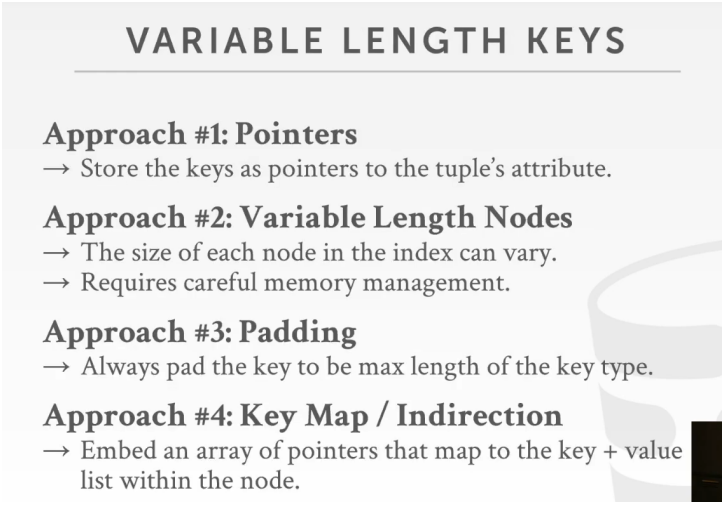
~~So，我可以在我的机械硬盘上的buffer pool中用来存放B+ Tree page，并将page的大小设置为1MB~~
so，如果我使用的是速度很慢的机械硬盘，我可以将用来存放B+ Tree page的buffer pool大小设置为1MB
48.52–48.55
whereas my data pages I'll keep them at 8 kilobytes or 16 kilobytes
~~然而，我将我的data page设置为8kb或16kb~~
而将存放 data pages的buffer pool的大小保持在8kb或16kb
48:56 – 49:00
The optimal size can also vary depending what kind of operations or queries you're doing on it
最佳大小同样是根据你所进行的操作或查询来决定的
49:01 – 49:03
So leaf node scans we're doing long sequential reads
So，对于叶子节点上的扫描来说，我们使用的是耗时长的循序扫描
49.03–49.06
those are typically better to have larger node sizes
通常情况下，这种更适合于大小更大的节点
49.06–49.08
because I can do more sequential I/O
因为我可以进行更多的循序扫描
49:08 – 49:12
where if I'm doing a lot of lookups a lot of traversal that's a lot of random I/O
如果我所进行的查找、遍历需要进行大量的随机I/O
49.12–49.17
so therefore I want to have smaller node sizes
因此，我想要的是体积更小的节点



49:17 – 49:28
So the next thing we can do is actually violate the very thing, that said in the beginning about how the we always have to merge anytime the anytime we're less than half full
So，正如我一开始所讲的那样，当节点没有达到半满状态，我们就得随时进行合并操作
49:29 – 49:32
And the demo I did it was sort of simple it would do exactly that
我之前展示的那个demo很简单，它就是按照这个来做的
49:33 – 49:38
But in practice you may actually not want to do this immediately when you're less than half full
但在实战中，当你的节点并没有达到半满状态的情况下，我们可能实际上并不想立即进行合并操作

49:39 – 49:44
Because it’s just like when we saw in the hash table, we do leaf nodes with linear hashing at the end
因为当我们在hash table中看到这个的时候，在最后我们会使用linear hashing来处理我们的叶子节点
49:45 – 49:46
I may compact something
我可能会去压缩某些东西
49.46–49.47
I may merge something
我可能会合并某些东西
49.47–49.49
because I went less than half full,
因为某些节点并没有处于半满状态
49.49–49.56
but then the next operation inserts into that node and now I have to just split all over again
但然后在下次操作的时候，我又往这个节点中插入了些数据，现在我得将它进行拆分开来
49:56 – 49.59
So the the merging operation is expensive
So，合并操作的代价是很昂贵的
49.59–50.02
~~splits off the top~~ splits are also expensive
拆分的代价同样也很昂贵
50.02–50.04
but it splits we have to do ,because we ran out of space in our node
但我们必须进行拆分，因为我们耗尽了我们node中的空间
50:04 – 50:10
The merge is we can actually relax that requirement and not merge things right away
实际上我们可以将要求放宽，我们不立即将这些东西进行合并
50:10 – 50:12
So it gets slightly unbalanced over time
So，随着时间的推移，二叉树会逐渐变得略微不平衡
50.12–50.17
and then in the background we can have like a garbage collector or something go through and do rebalancing
接着，在后台，我们可以使用比如垃圾回收器之类的东西来对它进行重新平衡
50.17–50.21
or what’s often times the case people just rebuild the entire tree from scratch
或者，有的时候，人们会直接从头开始来重建这棵树
50.21–50.23
and that fixes all these issues
这就会修复所有问题
50:23 – 50:27
So these a lot of times you see this in you know high–end commercial enterprise systems
So，你会在许多高端商用企业级数据库系统中经常看到这个
50.27–50.30
you know though they’ll shut the database down over the weekend
他们会在周末的时候关闭数据库

50:30 – 50:31
Because they’re gonna rebuild all their indexes
因为他们要去重建他们的所有索引
50.31–50.34
and that’s essentially what they’re doing there their rebalancing everything
简单来讲，这就是他们重新平衡二叉树的方式
50.34–50.35
because it wasn’t always merging correctly
因为这并不会一直正确合并
50:37 – 50:41
Anytime you see like a bank says they’re down at 3 a.m on a Sunday in the morning
有时候你们会看到，银行发通告表示它们会在周日早上3点的时候停止服务
50.41–50.44
it’s probably this is one of the things they’re probably doing
这就是他们所可能做的其中一件事（重构索引）



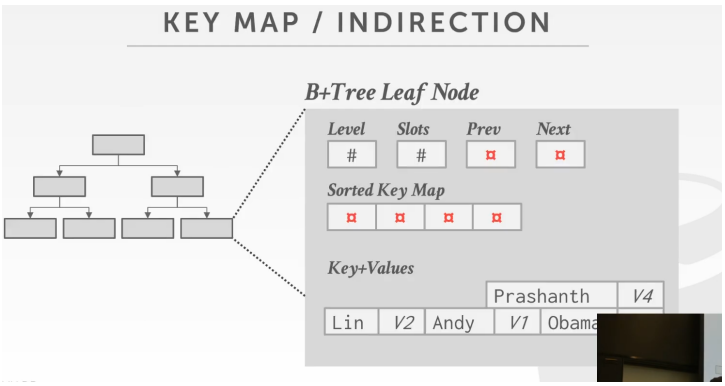
50:46 – 50:50
All right, so now we want talk about how we actually want to handle variable length keys
So，现在我们想讨论的是我们实际该如何处理可变长度的key
50:51 – 50:52
So again everything I’ve shown so far
So，在目前为止我所展示的东西中
50.52–50.54
we assume that the key is a fixed length, and the value is always fixed length
我们假设key是固定长度的，value也始终是固定长度的
50:55 – 50:57
And in practice the values will always be fixed–length
在实战中，value始终是固定长度的
50:58 – 51:00
So there’s four different ways we can handle this
So，我们有四种方式可以处理这个问题
51:01 – 51:05
So the first approach is that rather than storing the key itself in the node
So，第一种方式是，我们并不会将key本身存放在节点中
51:05 – 51:13
We store a pointer to the actual attribute or the tuple, where we can do a lookup to find what the key actually is

我们所保存的是指向该属性或tuple的指针，这样当我们进行查找的时候，我们就能找到这个key是什么了
51:14 – 51:20
So again if I have you know if I have an attribute that's a varchar,
So，再说一遍，如果我的属性的类型是varchar
51.20–51.24
instead of storing that varchar in the node,I have its record ID
我并不会将这个varchar保存在节点中，而是将它的record id保存在节点中
51:24 – 51:29
And then when I want to figure out whether the key,I'm doing a look–up on matches that key that's stored in that B–tree
然后，当我想要弄清楚我所查找的key是否与B–Tree（B树）上所保存的key匹配的时候
51:29 – 51:34
I follow the record ID get the page, and go look at them with a real value actually is
我会通过record id来拿到这个page，并看下它里面实际所保存的值是什么
51:35 – 51:37
So this is obviously super slow
So，显而易见，这样做的速度会非常慢
51.37–51.40
it's nice, because we're storing less data
这样做很nice的原因是我们所保存的数据量变少了
51.4–51.44
because now we just store the pointer instead of the actual key in the node
因为我们现在节点中所保存的是指针，而不是key
51:44 – 51:47
But it's expensive to do that lookup you know as we're traversing
但当我们遍历的时候，进行查找的代价会很昂贵
51:48 – 51:51
People tried this in the 1980s for in memory databases
在1980年代的时候，人们试着在内存型数据库中使用这个
51.51––51.53
because memory was really expensive
因为内存当时真的很贵
51:53 – 51:55
But nobody actually does this anymore
但实际上没人再会这么做了
51.55–51.59
everybody stores the keys always in the node
所有人都将key始终存放在节点中
52:00 – 52:02
The next you could you have variable length nodes
我们所能使用的第二种方法就是使用可变长度的节点
52.02–52.08
this is basically allows the the size of a node can vary based on what's stored in it
简单来讲，这允许一个节点的大小根据它所保存的东西来变化
52:08 – 52:09
But we've said this is a bad idea
但我们已经说过这是一个糟糕的想法
52.09–52.14

because we want our page sizes to be always the same in our buffer pool and on disk
~~因为我们想让buffer pool和磁盘上的page大小始终一样~~
因为我们想让我们的page大小在buffer pool和磁盘中始终是一样的
52:14 – 52:20
so we don't have to worry about doing the thin backing problem to decide how about you know find free space to put in what we want to store
这样我们就无须去担心该如何找到空闲的空间将我们的数据放进去了
52:21 – 52:22
so nobody does this one as well
So，同样也没人使用这种方式了
52:23 – 52:25
The next approach is do padding
下一个方式就是使用填充（padding）
52.25–52.29
or basically we say you look at what the attribute is, and you're trying to index on
在我们所试着进行索引的属性上
52:29 – 52:33
And we say that whatever the max size it could be, no matter what key you give us
不管这个属性的最大大小是多少，也不管你给我们的key是什么
52.33–52.41
we will Pat it out with either null bits or, you know zeros to make it always fit exactly our our node size
我们会使用null或者0对其进行填充，以此让它始终完全适合我们的节点大小
52:41 – 52:43
So everything is always nice and nicely aligned
So，所有数据始终都完美对齐
52:44 – 52:46
So some systems actually do this
So，实际上有些系统采用了这种方式
52.46–52.48
I think Postgres does this and we can look at that next time
我觉得PostgreSQL采用了这种方式，我们下节课的时候可以看下
52:50 – 52:54
But again itsthe trade off, I'm wasting space in order to store things
但再说一遍，这是一种取舍，为了保存数据，我得浪费空间
52:54 – 52.59
So this why is also – it's super important to make sure that you define your schema correctly,
So，这就是为什么要说确保schema的正确定义是非常重要的
52.59–53.04
like if I'm storing email addresses which are you know maybe 32 characters or 50 characters
比如，如果我存的是email地址，它的长度可能是32个字符或者是50个字符
53:04 – 53:06
But I set the varchar' size to be 1024
但我将varchar的大小设置为1024
53.06–53.09
if I'm padding it out up to 1024
如果我将数据填充到1024

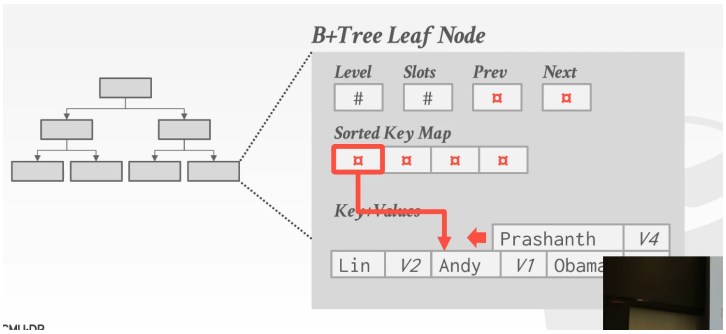
53.09–53.12
even though most of my emails aren't that big,but then I'm wasting a lot of space
虽然我大部分的email并没有那么大，但我仍然浪费了大量空间
53:12 – 53:13
Yes
请问
53:23 – 53:25
~~So when you~~ say again sorry
能再说一遍吗?
53:30 – 53:36
Correct,yeah so when you call Create table, you can define varchar,you define the length in it
正确，当你创建表时，你可以在表中定义varchar的长度
53:36 – 53:40
~~You don't have to put it in I and I I don't~~ different systems to deal with different things
不同的系统会做不同的事情
53:40 – 53:43
But in practice you always want to say this is the max size of what I actually can store
但在实战中，你总是想说，这是我实际所能存储的最大体积了
53:44 – 53:47
Right, and then so varchar supposed to be variable length,
So，varchar的长度应该是可变的
53.47–53.49
so even though I say the max size could be 32
So，即使我说，最大的大小是32
53:51 – 53:54
If you give it a 16 16 you know character string
如果你给出的字符串长度是16个字符
53.54–53.57
it could in theory store that more compactly
理论上来讲，它能更紧凑的保存这个数据
53:57 – 53.59
Some systems do different things
有些系统会做些不同的事情
53.59–54.05
some systems actually say it's a char and where it's always gonna be that size that's always padded out
实际上，某些系统会说它是一个char，但它们始终会将这个数据的大小填充到我们所设置的大小为止
54.05–54.07
they actually just still that store that as a varchar
这些系统实际上依然将它作为varchar来保存
54:07 – 54:12
So logically you don't know ,you don't care, underneath the covers they can do different things
从逻辑上来讲，你不知道，也不关心它们内部所做的不同的事情
54:13 – 54:15
And MySQL was always the worst offenders
并且，MySQL始终是最糟糕的玩意

54:15 – 54:17
So if you say the max size of our string is like 16
So，如果你说我们字符串的最大大小为16
54.17–54.22
and you give it a 32 character string, it'll store it just truncates it silently for you
然后，你往里面塞了一个32个字符的字符串，它会将它保存进去，但MySQL会偷偷摸摸将这个字符串的一部分给切掉
54:23 – 54:26
All right, so Postgres and all the system will throw an error
So，PostgreSQL和其他所有系统则会抛出一个错误
54.26–54.30
but the database system should enforce that correctly ,same thing for index
但数据库系统会强制让它正确，对于索引来说也是同样如此
54:30 – 54:32
~~We like~~ to build an index we have to be told
当我们说要构建一个索引时
54.32–54.36
you know here's the attributes and our tables your indexing
假设这里是我们要进行索引的属性和表
54:36 – 54:37
So we know what their type is
So，我们知道它们的类型是什么
54.37–54.44 ！ ！ ！
we know what their max sizes, and we can pad out as needed
我们也知道它们的最大大小，我们可以根据需要对数据进行填充
54:44 – 54.48
All right, with probably more common is to use in an indirection map
更为常见的方式可能是使用一个间接映射（indirection map）
54.48–54.54
where store pointers for our keys inside of our sort of key array
我们将我们key的指针存放在key数组中
54.54–55.01
but we're still the pointers are just actually two offsets in our node themselves rather than to some arbitrary page
我们这里的指针实际上是两个在我们这个node中对应的offset值，而不是指向其他任何page



55:01 – 55:05
So it would look like this, so we have a sort of key map
So，它看起来应该像这样，我们有这样一种key map
55:05 – 55:08
So again this is sorted these are just pointers are offsets to down here

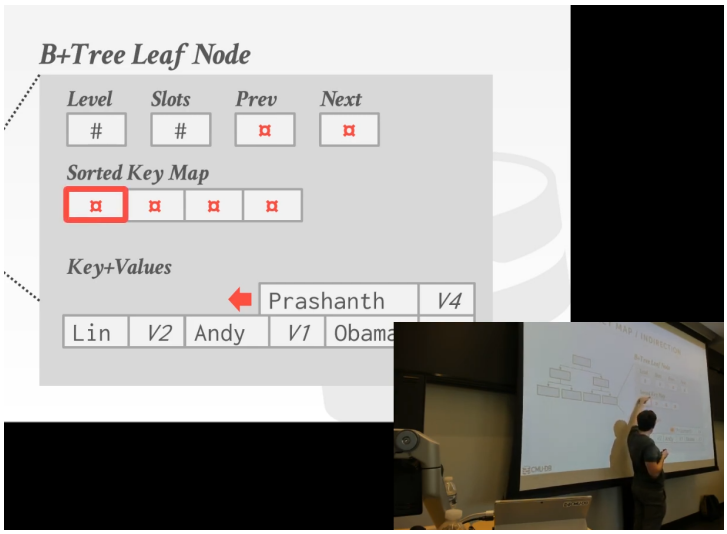
So，再说一遍，这里是有序的，这下面所存放的指针，其实存放的是offset值（知秋注：node所在的地址+数据在该node内的offset，就能获取到对应想要的数据，有点类似于我们前面所学的slotted-page）
55:09 – 55:13
But these are sorted based on the values of the keys
但它们是根据key的值来排序的
55:13 – 55:14
So to be very clear
So，为了让你们更明白些
55.14–55.17
the keys themselves not the keys corresponding value
这里我说的是key自身的值，而不是它所对应的value值
55:17 – 55:19
But the actual string that we're trying to store
但这是我们实际所试着保存的字符串
55:19 – 55:23
Right, and so just like in the slotted page layout for tuples
So，这和slotted page中tuple的布局很像
55:24 – 55:26
We're going to grow from the end to the beginning
这里的key+value是从后往前进行存储
55.26–55.31
and this side grows from the you know from from beginning to the end
这里的sorted key map则是按照从前往后的顺序进行存储



55.31–55.32
and at some point we get to full
在某一时刻，我们的节点会变成全满状态
55:32 – 55:35
Actually, I think this has to be fixed size, so we have to set a degree at a time
实际上，我觉得节点的容量必须是固定的。So，我们得给它设置一个度
55:35 – 55:38
So but if I don't run out of space for what I'm trying to store here
~~但如果我所保存的东西并没有耗尽这个节点的空间~~
但如果这个节点没有足够空间来存储要存的数据
55.38–55.42
then I can have an overflow page that's that's chained to this
~~那么我就可以使用一个链接到该page的overflow page~~
那么我就可以使用一个overflow page来链接到这里
55:43 – 55:47
So again this is just just a an offset to whatever the key is
So，再说一遍，这只是一个key在该节点内对应的offset值

55.47–55.49
so now if I'm doing binary search as I'm jumping around this array,
~~So，此时如果我进行二分搜索，当我跳到这个数组中时~~
So，此时，当我跳到这个数组中并进行二分搜索时
55.49–55.51
I jump down here to see what the actual key value is
我会根据该offset值跳到这里来看下，这个key实际值是什么
55:53 – 55:56
So what's a really simple optimization we can do to make this go faster
So，我们应该使用怎样的优化才能让这个变得更快呢?
55:58 – 55:58
Into that
那位同学请回答
56:07 – 56:13
It statement is is it a statement or a question, do we store this as an array or a linked list
So，他的问题是，我们是以数组的形式还是以链表的形式对它进行保存
56.13–56.14
it's always stores in an array
它始终保存在一个数组中
56:22 – 56:27
Okay, so his statement is, I'm storing this as an array or vector u vectors as a wrapper
an array
Ok，他的问题是，我是以数组的形式保存还是以vector来保存（vector其实是包装后的数组）
56:28 – 56:33
If I now do insertion or deletion that's gonna take O N or, yes
如果我想在进行插入或删除，它的复杂度就是O(n)
56:34 – 56:37
But again like this is this is just within the node itself
但这只是对节点自身而言
56:38 – 56:40
So the size is not that big
并且它的大小也没有那么大
56:41 – 56:45
Right,so you know fan out of like maybe 32
假设它里面有32个元素
56:46 – 56:48
So I have 32 elements I need I need to keep sorted
So，我有32个元素要进行排序
56:48 – 56:50
I can do that in cache that's very fast
我可以在缓存中来做，这样会非常快

57:07 – 57:10
Correct, okay so say I'm doing binary search
没错，假设我在进行二分搜索
57:11 – 57:14
So in this case your binary search is just you just do the linear search
So，在这个例子中，我们只做线性搜索

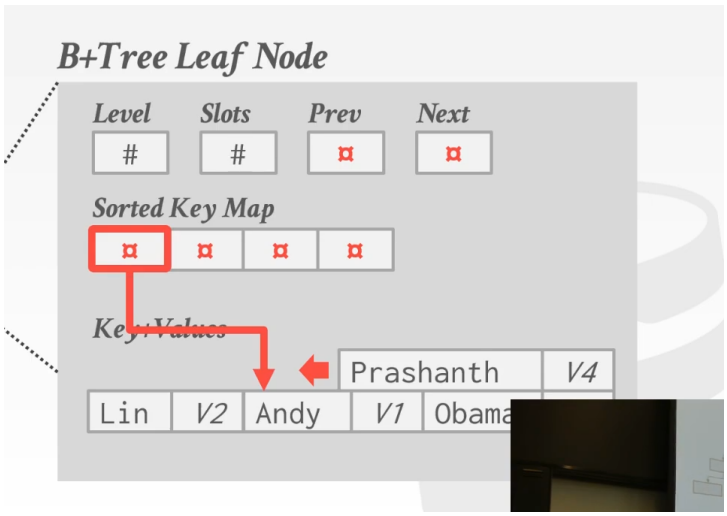


57:15 – 57:19

So I've just fought scan along and I want to see is there's the key I'm looking for a match what what I have

~~So，我沿着这里进行扫描，我想看下我所查找的key与我所拥有的key是否匹配~~

So，我沿着这里进行扫描，我想从中找到一个可以与我手里这个key相匹配的存在



57:19 – 57:23

So I have to follow this pointer,but again it's just an offset ,it's in the same page

So，我必须由这个指针入手，但再说一遍，它只是一个offset值，它在同一个page中

57:24 – 57:26

So it's gonna be your maybe 16 bits

So，它的大小可能是16bit

57:27 – 57:31

I follow that offset to jump to where this is ,and then I do my comparison

我按照这个offset值跳到它所在的位置，然后进行比较

57:31 – 57:34

And if it doesn't match， then I jump back and do the same in jump down here

如果它不匹配，那我就跳回去，然后再重复做同样的事情，然后跳到下面来进行比较

57:34 – 57:42

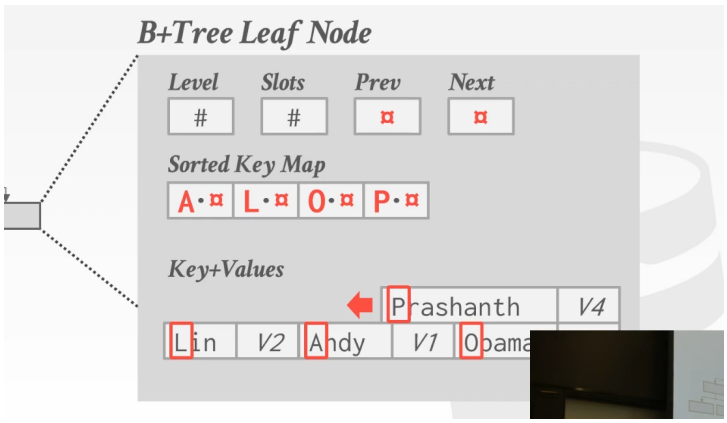
And so just like in slotted pages where the tuples want to be sorted in the order as they're laid out in the page and the same way that are sorted at the slot array

~~So，就像slotted page中那样，tuple想按照它们在page中的顺序进行排序，并且顺序与slot array中相同~~

So，就像slotted page中那样，tuple在page中的存储布局顺序与slot array中对应的顺序相同

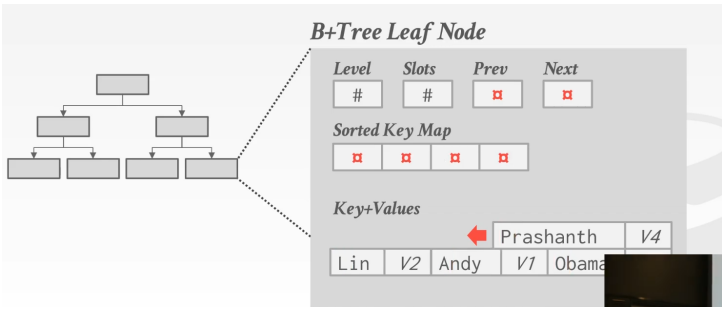
57:42 – 57:45！！！！！

This verbling data about at the bottom down here can be any order that at once
下方这一行数据可以是以任意顺序进行排列
57:45 – 57:48
I just know how to you know I know how to jump to it based on this
基于此，我知道该如何跳转
57:52 – 57:54
His question is for non leaf nodes do you do the same thing, yes
他的问题是，对于非叶子节点来说，我们是否会做同样的事情。是的
57:55 – 57:57
It's and this is for very well aligned data and any node
~~这是用于对齐好的数据以及任意节点的~~
无论什么节点，只要数据对齐好
57:59 – 58:03
So this is sort of micro optimization
So，这是一种微优化
58.03–58.05
and going to disk is always the most expensive thing
访问磁盘所付出的代价始终是最昂贵的
58:06 – 58:08
But a really simple thing we could do is
但我们可以做的一件非常简单的事情是
58.08–58.12
~~just recognize that before~~ since this is only 16 bits in general
通常情况下，因为这里仅有16 bit大小的空间
58.12–58.14
I have a lot of space up here
在上面我有大量的空间



58:14 – 58:20
So maybe I just take the first character of every string, and just embed it inside upper here
So，我所做的可能是将每个字符串的首字母放在上面这里
58:21 – 58:23
So now when I'm scanning along and trying to find the thing I'm looking for
So，现在当我开始扫描，并试着找我想找的东西的时候
58.23–58.30
if my key doesn't match exactly you know the first character that I know I don't need to traverse down and find it
如果我的key无法和这里第一个字符准确匹配，那么我就无须往下遍历来查找这个key了
58:30 – 58:33

Again, this is all going to be in memory
再说一遍，这些都是在内存中做的
58.33–58.35
this is like avoiding cache misses,
这样可以避免cache miss
58.35–58.38
in making the binary search and making the search on this run faster
这样可以让二分查找或其他查找变得更快
58:38 – 58:40
Okay, this is a micro optimization
Ok，这是一种微优化
58.40–58.43
a voting disk is always the major thing that we care about in this course
在这门课中，我们始终最关心的就是Voting Disk（它里面记录着节点成员的信息）
58:43 – 58:45
But this is a really simple trick that you can do to speed this up
但你可以使用这种很简单的技巧，来让这些操作变得更快
58:46 – 58:46
yes
请问
58:50 – 58:53
Again, it looks like what if there's two persons that name is start the same letter
他的问题是，如果两个人的名字开头的字母相同的话，会怎么样呢
58:54 – 58.56
Again, you'd have to depending what you're looking for
这取决于你查找的是什么东西
58.56–59.01
if you want to find exactly one, you find the first one you're done
如果你想找到一个，那么你找到第一个结果的时候，搜索就结束了
59:02 – 59:04
If you need to find anybody you have to go to both of them
如果你想找的是其中任何一个人（首字母是相同的两个），那么你就得去找这两个了



59:05 – 59:08
Alright, means same way here right
和此处的情况相同
59.08–59.14
for this one here, ~~I'd have to I'd have to if I'm trying to find~~ everyone's different here
对于这里的情况而言，这里每个人都不一样
59:13 – 59:16
But if there's like Paul and Prashanth who's my PG student
比如这里的Paul Prashanth，他是我负责的研究生
59:17 – 59:19

I would scan down here find Prashanth
我会先向下扫描，找到Prashanth
59.19–59.23
then actually go to the next one just make sure that that one doesn't have the same, you know doesn't have the same thing as well
然后，再跑到下一个位置，以确保这里并没有和它相同的元素存在
59:36 – 59:38
Okay,so he said collision
Ok，他说了这里会有碰撞
59:39 – 59:42
~~So there's dip~~ so I'm showing lexical graphical ordering alphabetical ordering for this
So，这里我所展示的是按照词汇顺序或者是字~~母~~表顺序进行排列的
59:43 – 59:49
in high–end database systems you can actually define arbitrary sort orders, everything still works the same
在高端的数据库系统中，实际上你可以对顺序进行任意方式的排序，但它们的工作方式都是相同的
59:55 – 59.58
You're talking like dictionary codes for not talk about that
你说的是字典相关的东西，但我们讨论的并不是那个
59.58–1.00.02
I could have different you know sorting based on whether it's you know code or what language I'm using
根据我所使用的语言或者代码，我可以定义不同的排序方式
01:00:03 – 01:00:07
~~For that one you have to begin~~ the database system would know ,this is how the sort order is
数据库系统会知道这里的排序规则是怎么样的
01:00:07 – 01:00:10
So what you know it would know what what prefix if it wants to store up in here
So，数据库就会知道它该如何保存前缀
01:00:12 – 01:00:14
Again, high level ideas that's still the same
再说一遍，高级层面的思想都是一样的
01:00:16 – 01:00:16
yes
请问
01:00:24 – 01:00:35
Correct, sir he said so he said, if you have K keys, you have at most k plus one pointers to other things
正确，如果你有k个key，那么你最多会有k+1个指向其他东西的指针
01:00:42 – 01:00:47
~~Not necessarily~~ for simplicity, yes, you just scan along the keys do linear search
为了简单起见，这没错，你可以沿着key进行线性搜索