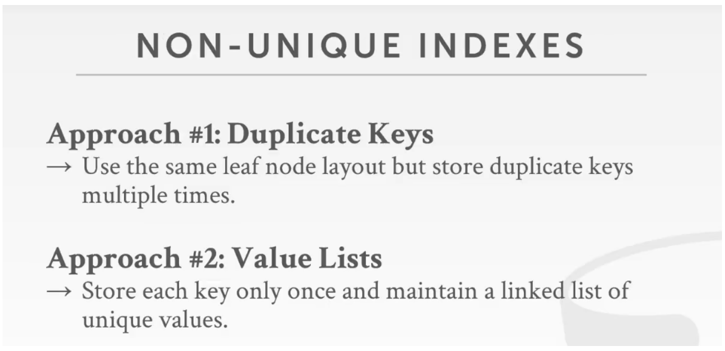
07–04

01:00:55 – 01:01:00
So his statement is that the really the complexity should be K times log n
So，他的说法是，复杂度应该是K*log(n)
01:01:01 – 01:01:06
Yes,~~ the that back ~~that’s a constant we can throw out
没错，但这个常数K我们可以丢掉
01:01:07 – 01:01:11
Because the log n is the maximum number of page IO`s I have to do to traverse
因为log(n)指的是我在遍历时对page必须要做的I/O的最大次数
1.01.11–1.01.15
that’s always orders of magnitude faster than doing the cacheline lookups here
这种做法始终要比通过cacheline进行查找要快好几个数量级
01:01:16 – 01:01:21 ！！！！
Remember I said in the very beginning here’s less storage hierarchy, anything above memory we don’t care about
还记得我在一开始所说的，我们对内存之上的存储层次结构并不关心
01:01:21 – 01:01:22
We can throw away
我们可以无须考虑
01:01:22 – 01:01:26
It`s to disk IO is the real color, we got to avoid that
磁盘I/O才是我们的重点，我们无须关心cpu cache层面的东西
01:01:29 – 01:01:30
Well, get to that a second
Well，我们稍后再来说你的问题

##################################



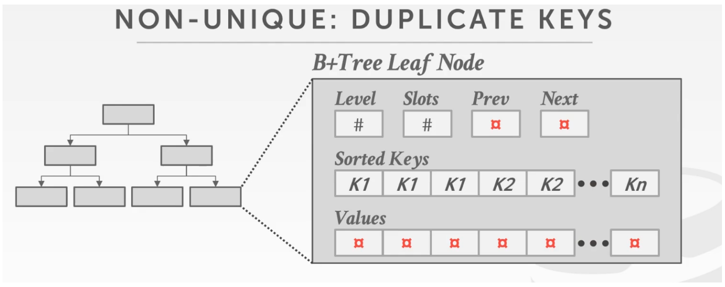
NON-UNIQUE INDEXES

Approach #1: Duplicate Keys
→ Use the same leaf node layout but store duplicate keys multiple times.

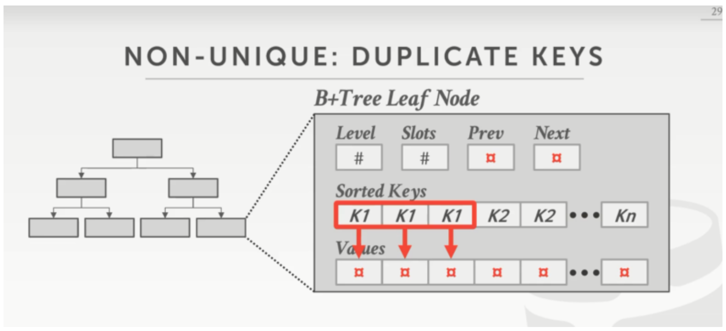
Approach #2: Value Lists
→ Store each key only once and maintain a linked list of unique values.

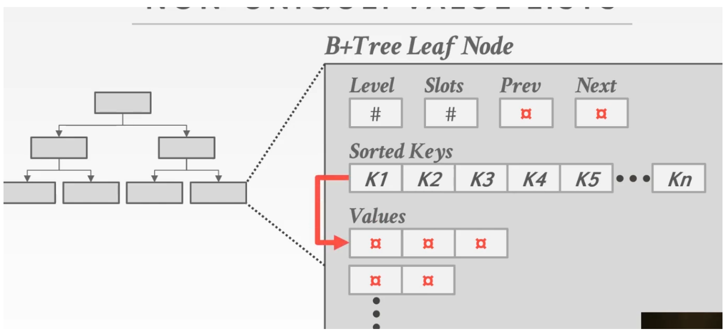
01:01:30 – 01:01:37
Okay, all right so now, I’ll get to the other thing as well is how we handle **non-unique indexes**
接下来我们要讨论的是如何处理非唯一索引
01:01:37 – 01:01:41
Well, this is the same thing we talked about in hash tables, there’s two basic Approaches

Well，这和我们所谈论的hash table是同一回事，这里有两种基本方案
01:01:41 – 01:01:43
You can duplicate the keys
你可以对key进行复制
1.01.43–1.01.50
and be mindful like in our example here the duplicate key split over to another to another node
同时也要注意，在我们的例子中，复制的key会被拆分到另一个节点中
01:01:50 – 01:01.52
We have to be mindful that that could occur
我们必须要小心这种情况的发生
1.01.52–1.01.56
and make sure we read everything we need to read
并确保我们读取了我们所要读取的一切数据
1.01.56–1.01.56
or we store a value list
或者，我们可以保存一个value列表
1.01.56–1.01.58
where we store the key once
即我们只保存key一次
1.01.58–1.02.04
and we ~~duplicate the value~~ or have a separate space in our node to store all the values for that given key
然后在我们的节点中，我们使用一个单独的区域来保存给定key的所有value



01:02:05 – 01:02:06
All right, so it looks like this
So，它看起来像这样
01:02:06 – 01:02:08
So if I duplicate the key
So，如果我对key进行复制
1.02.08–1.02.11
right I just have the key multiple times
我就会有多个相同key
01:02:11 – 01:02:15
And again it’s just like before， the offset points down to wherever the value is
再说一遍，这里和之前一样，这些offset值会指向下面的value



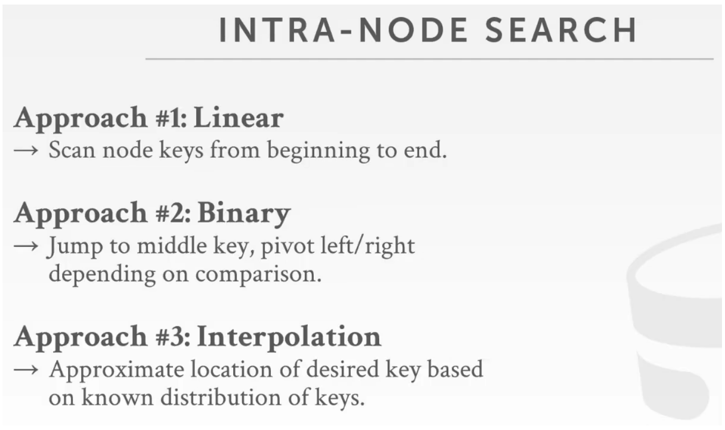
01:02:16 – 01:02:18
And you know if I insert a new one
如果我再插入一个新的
1.02.18–1.02.21
then I know I inserted here and certainly new k1
然后我就知道这里插入了一个新的K1
1.02.21–1.02.25
I inserted here and move everything over ,and everything still works
我把K1插入到这里，并将Sorted Keys中的其他元素重新移动排序，所有的东西依然能正常工作（知秋注：里面存放的offset值并没有发生改变）


01:02:25 – 01:02:28
The value list since you just looks like this, I just store the key once
value列表和我们之前所看的一样，在这里的key列表中，每个key我只保存一次
01:02:28 – 01:02:37
But then now I also have a pointer an offset to somewhere else in the node ,where I have all the values that correspond to that that giving key
但现在我通过一个指针指向该节点中某个offset值处，它里面保存了所有该key对应的value
01:02:37 – 01:02:39
That’s the first approach is more common
第一种方法更加常用
1.02.39–1.02.41
 I don’t know who actually does this one here
实际上，我并不清楚到底有谁在使用第一种方法
1.02.41–1.02.41
yes
请问
01:02:51 – 01:02:57
So her question is, can I assume for that duplicate keys will always be in the same node ,no
她的问题是，我们能否让重复的key始终都在同一个节点中？答案是No
01:02:58 – 01:03:00
So in the example I showed from that demo
So，在我所展示的demo中
01:03:00 – 01:03:03

It actually moved it over as a sibling key
实际上，我们将它变为一个同级的key（同一层）
01:03:05 – 01:03:06
That's one way to do it
这是其中一种方式
1.03.06–1.03.07
all other systems actually would have an overflow chain
实际上其他所有系统会使用overflow chain这种方式
1.03.07–1.03.11
that would say for that given leaf node
我们会这样说，对于一个给定的叶子节点
1.03.11–1.03.19！！！！
oh by the way here's some other pages or other nodes down below you that have that you have the keys that correspond to what ,you know what you're actually storing up above
在它的下方有某些其他page或者节点，它里面保存了与我们上面keys所对应的内容
01:03:25 – 01:03:27
Her question is, if I'm searching for a given key, how would I know what key to follow
她的问题是，如果我使用一个给定的key进行查找，那么我该怎么沿着我知道的那个key去查找

01:03:28 – 01:03:30
So going back to that example
So，回到那个例子中去
1.03.30–1.03.35
 I would I found looking for a greater than equal to four
如果我要找的是大于等于4的key
1.03.35–1.03.38
I had I have to go down on the on this side
那么我得沿着这个方向往下走
1.03.38–1.03.41
and find the first entry point for four
并找到4的第一个入口点
1.03.41–1.03.47
then I keep scanning along the leaf nodes, until I find something that's not equal four ,and then I know I've seen everything
然后我沿着叶子节点进行扫描，直到我找到不等于4的东西为止，那么我就知道我找到了所有我要的数据
01:03:47 – 01:03:50
Again the database system knows where the keys are unique or not
再说一遍，数据库系统知道这些key是唯一的，还是不唯一的
01:03:50 – 01:03:51
So it knows whether it has to do that
So，它知道它该怎么做
01:03:51 – 01:03:55
So it knows that, oh this is a primary key or this is a unique index
So，如果它知道这是一个主键或者是一个唯一索引
01:03:56 – 01:03.58
So the thing I'm looking for should only appear once
So，我所查找的东西就应该只出现一次

1.03.58-1.04.00
 and therefore I just get to the one leaf node that it has what I want
因此，我只需要找到那个包含我想要的数据的那个叶子节点就行
01:04:00 – 01:04:01
If it’s non unique
如果它不是唯一的
1.04.01-1.04.03
 then you have to account for that
那么我们就必须对其进行处理
1.04.03-1.04.06
and either, again you if it’s if it’s just duplicated across leaf nodes
如果它在多个叶子节点中有重复
01:04:06 – 01:04:07
I scan along siblings,
我会沿着叶子节点进行扫描（知秋注：兄弟节点）
1.04.07-1.04.08
 if it’s an overflow
~~如果是在overflow的节点上~~
如果这个叶子节点有overflow（page，或者是由overflow所引出的节点，page可以看作是节点）
1.04.08-1.04.11
 I just find that the first leaf node and then scan down its chain
~~我就会找到第一个overflow的叶子节点，并往下扫描它的chain node~~
我就会先找到这个叶子节点，然后往下扫描它的chain node（其实就是由该key对应数据放不下所引出的这些overflow page所对应的节点）
01:04:12 – 01:04:12
Yes
请问
01:04:16 – 01:04:23
The question is the size is the size of key always the same, or do the size of the key is not always the same, or the values or this always Same
他的问题是，key的大小会始终一致，还是不一致，或者value的大小是否始终一致
01:04:23 – 01:04:25
Again, I’ll show this next class
我会在下节课向你们展示这一点
1.04.25-1.04.29
if the value is just a record ID for a tuple
如果这里的value只是一个tuple的record id
1.04.29-1.04.30
always the same
那么就始终一致
1.04.30-1.04.32
 see the 32 bits or 64 bits depending on the system
根据系统的不同，它可以是32 bit，也可以是64 bit
01:04:33 – 01:04:35
If it’s the actual tuple itself
如果它是tuple自身
1.04.35-1.04.37
 like in MySQL

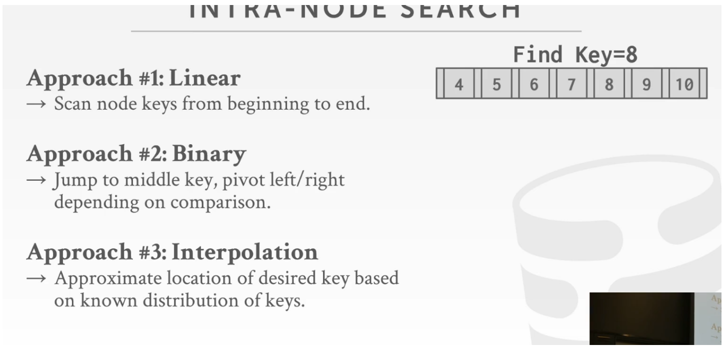
比如在MySQL中
1.04.37-1.04.39
then you got a deal overflows that way
那么你就必须处理overflow的情况
1.04.39-1.04.42
and that that's more complicated, we'll discuss that next class
这会更加复杂，我们会在下堂课进行讨论


INTRA-NODE SEARCH

Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

01:04:45 – 01:04:50
All right, again we've already discussed this briefly, but I'm gonna show that there's different ways to do searches within the node
All right，我们已经对此进行了简单讨论，但我想向你们展示几种不同的在节点中进行搜索的方式
~~一些在节点中进行搜索的不同方式~~
01:04:50 – 01:04:54
Again, I Traverse as I'm traversing the nodes traversing the tree
当我在遍历节点或者是遍历树的时候
01:04:54 – 01:05:02
I have to do a search on the key array to find a thing I'm looking for to decide whether you know there's a match that I want or whether I need to go left or right
我必须在key数组进行搜索，以此找到我要找的东西，并弄清楚这里遇到的key是否与我要找的匹配，或者是我应该往左走还是往右走


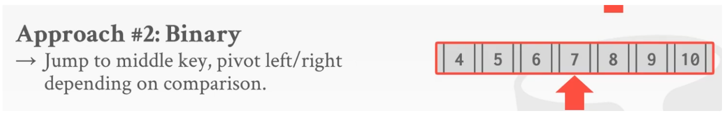
INTRA-NODE SEARCH

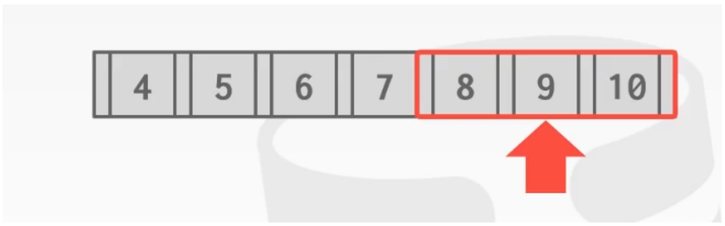
Approach #1: Linear
→ Scan node keys from beginning to end.

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

Approach #3: Interpolation
→ Approximate location of desired key based on known distribution of keys.

01:05:02 – 01:05:05
So the most basic way to do this, it's just a linear search
So，最简单的做法就是使用线性搜索
01:05:05 – 01:05:06
So I'm trying to find key 8
So，如果我试着找到值为8的key
1.05.06-1.05.13
I just start at the beginning of my sort of key array scan along to I find one and I'm done
我会从我的key数组的开头开始扫描，以此找到我要的那个key，找到之后，扫描就结束了
01:05:13 – 01:05:15

Right worst case scenario, I have to look at all K keys
这是最糟的情况，因为我得去查看所有的key
01:05:16 – 01:05:17
Binary search
二分查找
1.05.17–1.05.18
if it’s sorted
如果key数组是已经排好序的

Approach #2: Binary
→ Jump to middle key, pivot left/right depending on comparison.

1.05.18–1.05.20
then I just find the middle point
那么我只需找到它的中点
1.05.20–1.05.21
jump to that
跳到中点处
1.05.21–1.05.25
figure out is that less than or greater than the key I’m looking for or the one I am looking for
与它进行比较，看看我要找的key比它大还是小
1.05.25–1.05.28
and that tells me whether I go left or right
这样我就知道该往左边查找还是往右边查找了
01:05:28 – 01:05:29
In this case here, I’m looking for 8
在这个例子中，我要找的是8
1.05.29–1.05.31
middle is 7
中点是7
1.05.31–1.05.32
I know 8 is greater than 7
我知道8比7大


1.05.32–1.05.33
so I jump over here
So，我跳转到这里
1.05.33–1.05.36
then I think the halfway point of that ,I get 9
这里的中点值为9
1.05.36–1.05.37
then I go to this direction
那我就往左边走
1.05.37–1.05.41
and I get 8 and I find what I want

我找到了我想要的8

01:05:41 – 01:05:43

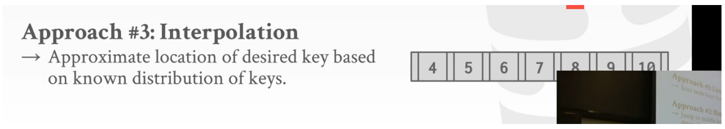
One thing though that is kind of cool you can do

你还可以使用一种更酷炫的方法

1.05.43–1.05.47

if you know what the values actually look like for your keys actually what the keys look like

如果你大概知道你的key值是多少的话



Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.

1.05.47–1.05.50

it`s that you can use an interpolation technique

那么你就可以使用interpolation这种技术

01:05:50 – 01:05:54

We can approximate the localcation of the key

我们可以估计该key所在的大概位置

1.05.54–1.06.00

by doing some simple math to figure out what your starting point should be for your Linear search

通过某些简单的数学方式，以此来弄清楚我们线性搜索的起点在哪

01:06:00 – 01:06:03

So rather in case of linear search you start for the beginning and go all the way to the end

So，这里我们并不是在key数组的起点处使用线性扫描一路扫到尾

01:06:03 – 01:06:07

If I know that my keys or in this case integers

如果我知道我的key的大致大小

01:06:08 – 01:06:10

And I know something about their distribution

我知道它们的分布信息

1.06.10–1.16.13

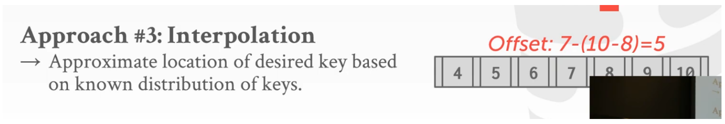
then I can do a really simple you know simple math and say

那么我就可以使用简单的数学方法来进行处理，我表示

1.16.13–1.16.16

well I know I have 7 keys in my array

Well，我知道在我的数组中有7个key



Approach #3: Interpolation

→ Approximate location of desired key based on known distribution of keys.

Offset: 7-(10-8)=5

1.16.16–1.16.21

and the max key is 10, and I’m looking for 8

key值最大为10，我所找的是8

01:06:21 – 01:06:23

So if I take 10 minus 8 and get 2

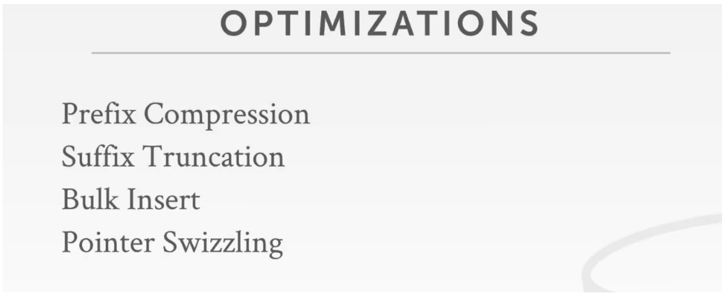
So，10–8我得到2

1.06.231.06.27

and then 7keys– minus 2 and get 5

然后，7–2得到5

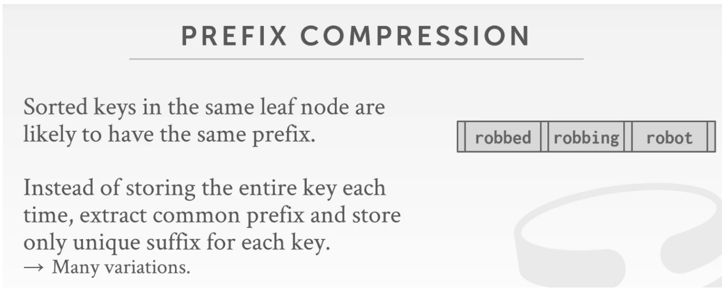
01:06:27 – 01:06:32
I know I can just jump to the fifth position and that’s at least a starting point for what I’m looking for
这样我就知道我可以跳到第5个位置，这就是我所寻找的最小起点
01:06:33 – 01:06:35
So this obviously works
So，很明显这种方式可行
1.06.35–1.06.38
because they’re always increasing the monotonic order
因为这里它们的值始终是按照单调递增的顺序排列

01:06:38 – 01:06:40！！！
Right if there floats this is hard to do, if there’s strings I don’t think you can do this
如果这里是浮点数，这种方法做起来就有点困难。如果是字符串，那我不觉得你们能用这个
01:06:40 – 01:06:44
But this is another technique you could do to make that search go faster
但你们可以使用另一种技术来让这种搜索变得更快
01:06:47 – 01:06:51
This one I don’t know how is calm how common it is,the binary search I think what everyone does
这种方法我并不清楚是否普遍使用，但我觉得二分查找应该是每个人都用的
01:06:51 – 01:06:53
But again there’s this trade-off now
但再说一遍，这里面存在了取舍问题
1.06.53*–1.06.56
an ordinary binary search I had to make sure my keys are in sorted order
在普通的二分查找中，我必须得保证我的key都是按顺序排列
01:06:56 – 01:06:58
If I’m doing the linear search, I don’t have to do that
如果我使用的是线性搜索，那么我就不用进行排序
01:06:58 – 01:07:01
So therefore as I update the nodes
So，在我对节点进行更新时
1.07.01–1.07.03
I don’t pay that penalty of maintaining the sort order
我无须为维护顺序而付出代价

OPTIMIZATIONS

Prefix Compression
Suffix Truncation
Bulk Insert
Pointer Swizzling

01:07:06 – 01:07:11
All right, so let’s finish up real quickly to let some optimizations, we can do it to make it go better
So，让我们快速讲下一些优化上的内容，它能让搜索变得更好
01:07:11 – 01:07:18

So these are the kind of things that like again a real database system would actually do to make B+tree go faster
在一个真实的数据库系统中实际上会去使用这些东西，来让B+Tree变得更快



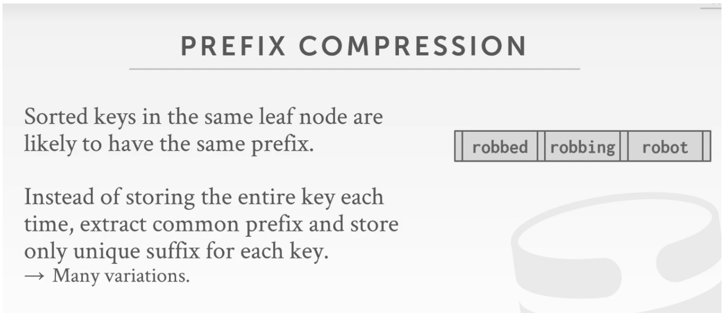
PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.

robbed	robbing	robot

01:07:19 – 01:07:23
So the first type first we think about we’re told is to compress the data
So，第一种我们要讨论的技术就是关于压缩数据这方面的
01:07:23 – 01:07:25
So the first kind of question scheme we can do is called prefix compression
So，第一种技术我们称之为前缀压缩（prefix compression ）
01:07:26 – 01:07:30
And this is based on the observation that, because we’re keeping the keys in sorted order
它是基于我们的key都是有序排列的
01:07:30 – 01:07:33
It’s very likely and a lot of data sets
在许多数据集中，很有可能会有这种状况
01:07:34 – 01:07:38
The keys that are stored in a in a single node are actually going to be very similar to each other
即保存在同一个节点上的key实际上彼此之间会非常相似
01:07:39 – 01:07:41
All right, because we end up sorting them right
这是因为我们对它们进行了排序



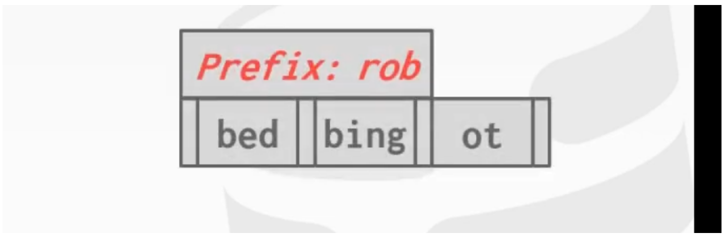
PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.

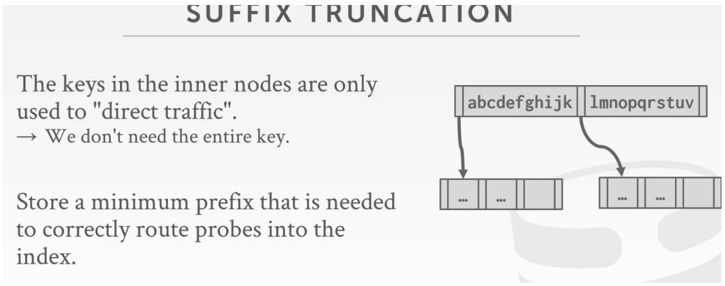
robbed	robbing	robot

01:07:42 – 01:07:46
So in this case here I have know that has three keys **robbed,robbing** and **robot**
So，在这个例子中，我已经知道我有了3个key，即robbed，robbing以及robot
01:07:46 – 01:07:50
Well all three of them share the same prefix ROB
它们三个都有相同的前缀，即rob
01:07:50 – 01:07:56
And so rather than me duplicating or storing that redundant ROB over and over again for every single key
So，对每一个key而言，我无须一遍又一遍的复制或存储这个冗余字符串rob



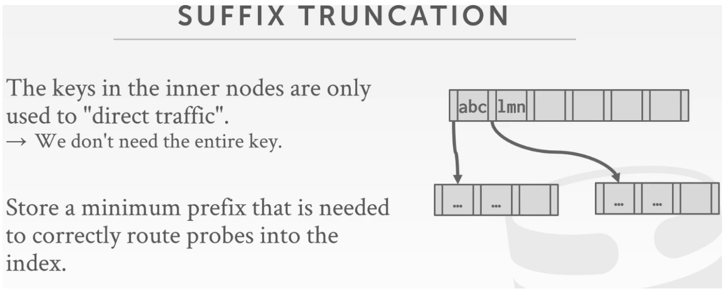
1.07.56–1.08.00
what if I extract that out to store the prefix once ROB
如果我将这个前缀rob提取出来
01:08:00 – 01:08:04
And then for the the keys, I destroy the remaining parts of that’s actually different
然后对于这些key来说，我将公共的部分删除后，每个key所剩下的部分实际上是不同的
01:08:06 – 01:08:10
So this is very very common this is these are called sometimes prefix compression, prefix trees
这非常常见，有时它被称为前缀压缩（prefix compassion）或prefix tree（Trie Tree）
1.08.10–1.08.17
 this is why you use in a lot of high–end or a lot of large database systems
这就是为什么你会将它用在许多高端或大型数据库系统中的原因了
01:08:17 – 01:08:20
Because you know because there’s so much duplicate data
因为，这里面有太多太多的重复数据了
01:08:20 – 01:08:23
So Facebook uses this for all their internal MySQL stuff
So，Facebook在他们所有的内部MySQL中都使用了这个
1.08.23–1.08.25
 and it makes a big difference for them as they save a lot of space
这引起了性能上的巨大差异，因为这样做节省了大量的空间
01:08:26 – 01:08:29
So this is sort of one way to do this those other optimizations you can do like
So，这只是优化的其中一种方式，你还可以使用一些其他的优化
1.08.29–1.08.32
if again if I’m doing a clustered index
如果我使用了聚集索引
1.08.32–1.08.36
 where I know all my tuples, the tuples are on on disk or on pages
我知道我所有的tuple都存放在磁盘或page中
1.08.36–1.08.38
in the same way that they’re sorted in my index
~~在我的索引中，它们以相同的方式进行排序~~
同样，在我的索引中，它们是以排序存在的
1.08.39–1.08.41
 then it’s very likely that tuples in the same node
那么tuple很有可能都在同一个节点上
1.08.41–1.08.44
 their record ID will have the same page ID
它们的record id会使用相同的page id
01:08:44 – 01:08:46
Because they’re all going to land on the same page
因为它们都会落在同一个page上

01:08:46 - 01:08:51
So rather than storing that page ID over and over again for every single tuple in a node
So，我们无须将每个单个tuple的page id一次又一次地存放在一个节点中
01:08:51 - 01:08:54
I store the page ID once and then store their offset or slot separately
我可以只需保存一次page id，然后将它们的offset值或slot分别存放
01:08:55 - 01:08:56
Right, yes
请问
01:09:10 - 01:09:11
Yes
没错
01:09:12 - 01:09:14
The question is how do we actually decide what to do
她的问题是我们实际该如何决定到底怎么做呢
01:09:14 - 01:09:18
Right, so you basically can say every single time I insert
So，简单来讲，当我每次进行插入时
1.09.19-1.09.21
I figure out what the common prefix is and that's what I'll store
我会弄清楚公共前缀是什么，我之后会将它保存起来
01:09:22 - 01:09:27
You could say anytime I do like a compaction or do like a real optimization
当我每次进行数据压缩或者优化的时候
1.09.27-1.09.33
then I decide what the best is right that you know for the for my keys right then and there
~~那么我会去选择对我的key来说最好的优化方式~~
那么我会为我的key去选择属于它最适合的优化方式
01:09:33 - 01:09:36
In practice also think of it like in a lot of database systems
我们来思考下，在实战中，很多数据库系统的做法
1.09.36-1.09.41
the the newer keys might get inserted in always on the one side of the tree
新插入的key可能会一直放在树的一侧
1.09.41-1.09.43
like this is always increasing in value
即key的值一直在增加
01:09:44 - 01:09:45
And so therefore
因此
1.09.45-1.09.50
the a lot a large portion of the tree on the other side is going to be static
树中很大一部分数据都会是静态的
1.09.501.09.53
it's gonna be you know mostly read-only
大部分数据都是只读
01:09:53 - 01:09:57

So at that point I can make it a hard decision like here’s how I want to do compression or compaction
此时我就可以做出一个艰难的决定，我该如何进行压缩
01:09:58 – 01:09.59
Different trade-offs
这其中有不同的取舍
1.09.59-1.10.01
you do it online or offline
你可以在线上做，也可以在线下做
1.10.01-1.10.02
yes
请问
01:10:07 – 01:10:10
So this question is what happens to someone insert the words sad
So，他的问题是当我们插入sad这个单词时，会发生什么
01:10:11 – 01:10:15
Right and ends up in this node, yeah you have that you have to account for that, you have to maintain it on the fly
你必须对它进行统计，并且得在运行时对它进行维护
01:10:17 – 01:10:18
Correct,yeah
没错
01:10:19 – 01:10:24
Or you could say source mesh and distro metadata to say, this prefix is only used for the first three keys not the other ones
或者你可以这样做，此处的这个前缀只适用于这3个key，并不适用于其他的key
01:10:25 – 01:10:27
Right, there’s the bunch of tricks you can do
你可以使用许多技巧来做到这点

SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

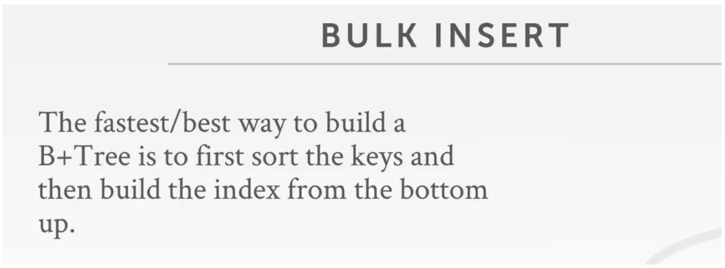
Store a minimum prefix that is needed to correctly route probes into the index.
abcdefghijk
lmnopqrstuv
…
…
…
…
01:10:29 – 01:10:33
So the the opposite of prefix compression is **suffix truncation**
So，与前缀压缩相对的一种方法就是后缀截断（**suffix truncation** ）
01:10:33 – 01:10:44
And the basic idea here is that, we can recognize that, we don’t maybe need to store the entire key in our inner nodes to figure out whether we want to go left and right
它的基本思路是，我们无须在我们的inner node中存储完整的key值，以此来弄清楚我们是该往左走还是往右走
01:10:44 – 01:10:50
So in this case here we have ABCD up to K for one key and IMNO up to V for another key
在这个例子中，我们的key如图所示
01:10:50 – 01:10:53

But if I'm just trying to see whether I want to go left or right
但如果我试着去弄清楚我该往左走还是往右走
1.10.53–1.10.56
 I can probably get by just looking at the first you know in this case here first character
在这个例子中，我们可以从第一个字符就能判断出来
01:10:57 – 01:11:00
So instead of storing the entire key in the inner node
So，我们无须将整个key都存放在inner node中
1.11.00–1.11.06
our store a uniquely distinguishing prefix of it
我们可以在inner node中存储能够区分key的唯一前缀即可
1.11.06–1.11.08
and then throw away the remaining suffix
然后，将剩余的后缀抛掉

SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic".
→ We don't need the entire key.

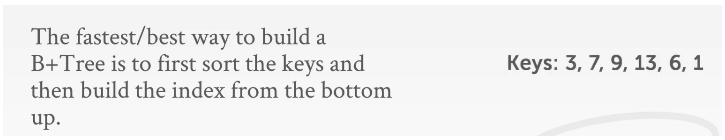
Store a minimum prefix that is needed to correctly route probes into the index.

01:11:08 – 01:11:12
So in this case here, I could store A and L and that would been enough
So，在这个例子中，我可以只存a和l，这就足够了
1.11.12–1.11.13
 but I'm showing ABC LMn
但这里我所展示的是abc和lmn
01:11:14 – 01:11:17
Right, and again down below I still have to store the entire key
在下面的节点中，我依然得保存整个key
1.11.17–1.11.21
because I need to go be able to have it be it'll say, you know is the key I'm looking for here
因为我需要能够找到它，并表示这是我所寻找的key
01:11:21 – 01:11:25
But it might in my inner guideposts, I don't need to have this have the full key
但在我的inner node中，我无须保存完整的key用来当做路标
01:11:25 – 01:11:30
And of course again you have to maintain this if something and somebody insert something that would violate these this
如果某人插入了某些数据，那可能就会违反这个，那我们就得去对它进行维护
1.11.30–1.11.34
 we have to know we order it or reorganize it
我们必须对此进行排序或重新整理
01:11:34 – 01:11:37
But in practice that you know if the data is is not changing a lot
但在实战中，如果数据并没有发生太多改变
1.11.37-–1.11.37
 then this could be another big win

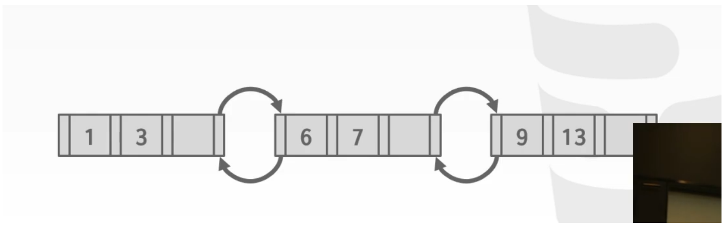
那么这种做法就可能非常棒
01:11:40 – 01:11:44
So as far as you know prefix compression is more more common than the suffix truncation
So，据我所知，前缀压缩（prefix compre）要远比后缀截断来得更常用

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

01:11:46 – 01:11:50
All right, the last few things I want talk about is how to handle **bulk inserts** and **pointer swizzling**
最后我想谈论的东西就是如何处理**bulk inserts和pointer swizzling**
01:11:51 – 01:11:53
So in all the examples I showed so far
So，目前为止，在我所展示的例子中
1.11.53–1.11.59
 we assume that we’re incrementally building out our index, we’re inserting keys one by one
我们假设，随着我们一个接一个插入我们的key，以此逐步构建出我们的索引
01:11:59 – 01:12:02
But in a lot of cases you have all the keys ahead of time
但在大多数情况下，你会提前拥有所有这些key
01:12:03 – 01:12:06
So it’s a very common pattern that people do in databases that say
So，这是一种人们在数据库中非常常用的方法
1.12.06–12.08
I want to bulk load a new data set
比如说，我想批量加载一个新的数据集
1.12.08–1.12.10
now I’ve collected data from some other source
现在，我已经从其他数据源收集到了一些数据
1.12.10–1.12.12
and I want to put it into my database
我想将它放入我的数据库
01:12:12 – 01:12:16
A lot of times what people do is they turn up all indexes, bulk load the data
许多时候，人们所做的就是打开所有索引，并批量加载数据
1.12.16–1.12.18
insert it into the table
并将数据插入表中
1.12.18–1.12.19
and then they go back and add the indexes
然后，他们回过头来，添加索引
01:12:19 – 01:12:21
Right, so that way you as you as your insert the new data
当你插入新的数据时

1.12.21*-1.12.24
you're not trying to maintain the index which is expensive
你不会去试着维护索引，因为这样做代价太高了
01:12:24 - 01:12:25
So in this case here
So，在这个例子中
1.12.25-1.12.27
if you have all the keys ahead of time
如果你提前就拥有了所有的key
1.12.27-1.12.34
a really simple optimization to do to build the indexes rather than building it top-down like we've done so far
我们可以使用一种超简单的优化方式来构建索引，这里我们并不会像我们之前所做的那样，即自上而下地构建索引
01:12:35 - 01:12:37
You actually build it from the bottom up
实际上，你可以自下而上去构建索引

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

01:12:37 - 01:12:41
So let's say these are the keys I want to insert, the very first thing I do is just sort them
So，这里是我想要插入的key，首先我要做的就是对它们进行排序
01:12:42 - 01:12:44
And we'll see in a few weeks
我们会在几周后看到
1.12.44-1.12.53
We there's an efficient algorithm we can use tech that can sort data in such a way that maximize the amount of sequential i/o, we have to do
我们可以通过一种高效的算法来进行排序，它能得到我们必须做的最大循序I/O的数量
01:12:53 - 01:12:54
So we can sort it
So，我们可以对其进行排序
1.12.54-1.12.59
and that lets me way more efficient than actually building the index one by one
实际上，这比我一个接一个地去构建索引来说，这种做法高效的多


01:12:59 - 01:13:02
And then we just lay it out along leaf nodes have everything filled out correctly
然后，我们将它们排列在叶子节点上，将它们正确填入到叶子结点中
01:13:03 - 01:13:05
And then going from the bottom to the top
接着，我们从下往上走
1.13.05-1.13.11
we just fill in the inter nodes and generate our pointers

我们只需要（使用中间key）来填充inter node，并生成我们的指针
01:13:11 – 01:13:16
So again this is this is a pretty standard technique that any major database system we will support
So，再说一遍，这是任何主流数据库系统都支持的一种标准技术
1.13.16–1.13.18
when you call create index and a large dataset that already exists
当你对一个已经存在的大型数据集进行索引创建时
01:13:18 – 01:13:20
And then once it's already built once it's built
一旦索引构建完成
1.13.20–1.13.23
then I can you know maintain our do any changes I want just like before
那么我就可以像之前一样对我们所做的任何改变进行维护
1.13.23–1.13.25
there's no real difference to it
它和之前所做的，并没有什么区别
01:13:25 – 01:13:30
The database system doesn't know whether you did the bulk insert versus the incremental build to build an index
数据库系统不知道你是批量插入，还是逐个构建索引
01:13:31 – 01:13:32
Everything's still the same
所有东西都一模一样
1.13.32–1.13.33
in the back ,yes
后面那位同学，请说出你的问题
01:13:39 – 01:13:42
So this question is what happens if you want to merge a small B+tree into a large B+tree
So，他的问题是，如果我们想将一个小型B+Tree合并到一个大型B+Tree中，这会发生什么
1.13.42–1.13.44
let's take that offline
我们课后再讨论这个问题
1.13.44–1.13.46
and we have a paper that does something like this
我们有一篇paper提到了这个
01:13:46 – 01:13:51
But I would say in general building index is very building indexes with bulk inserts very fast
但总的来说，我想说的是使用批量插入（bulk insert）来构建索引，速度会非常快
01:13:51 – 01:13:53
It`s a very very hard problem
这是一个非常非常难的问题
1.13.53–1.13.57
and it's at least in academia it's under appreciated
至少在学术界，这是一个值得赞赏的问题
01:13:57 – 01:13.59
This is very very common
它非常非常普遍

1.13.59–1.14.03

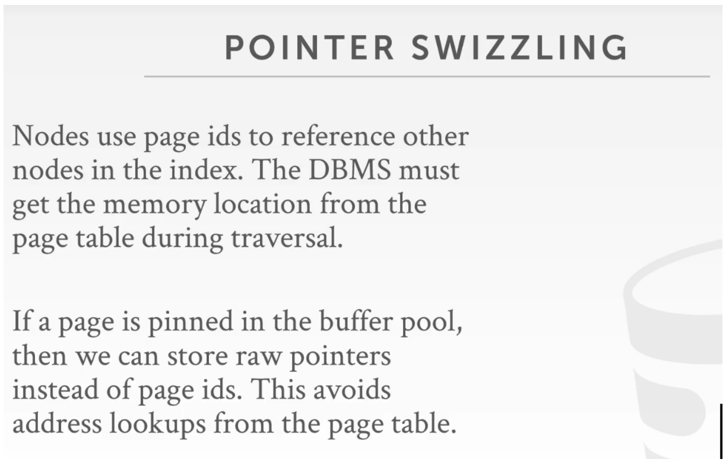
so how do you database system this as fast as possible is super important

So，你该如何让你的数据库变得尽可能得快，这一点非常重要

01:14:03 – 01:14:04

So let’s talk about efforts

So，让我们来谈论下该如何努力吧



POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

01:14:05 – 01:14:07 ！！！！

All right, the last thing I’ll talk about is called **pointers swizzling**

All right, 最后我想谈论的东西叫做**pointer swizzling**

01:14:08 – 01:14:14

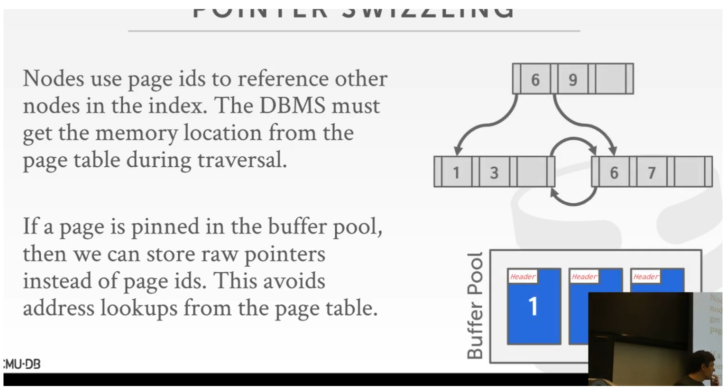
So again I talked about how the way we figure out how to traverse the index is

So，我已经谈论过我们该如何遍历索引

1.14.14––1.14.16

by having these pointers from one node to the next

即通过这些指针从一个节点跳到下一个节点



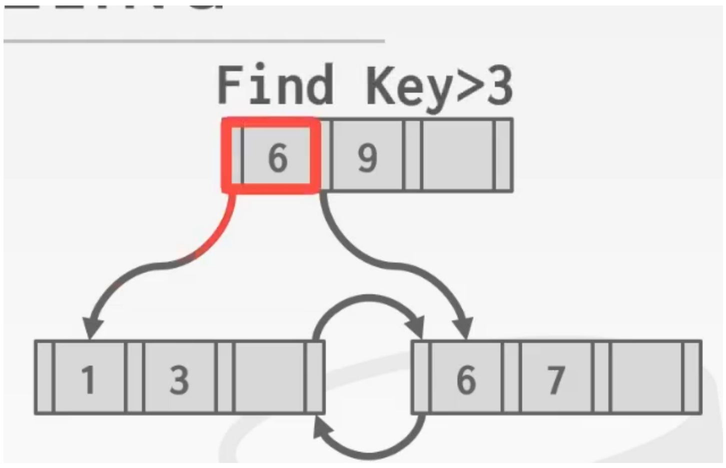
Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

01:14:17 – 01:14:22

In actuality what we’re storing is not you know raw memory pointers, we’re storing page ids

实际上，我们所保存的并不是原始的内存指针，而是page id



01:14:23 – 01:14:27

And whenever we want to do traversal the same when I find key greater than 3

当我们想进行遍历的时候，比如当我想找到比3大的key的时候

01:14:28 – 01:14:28

We start here

我们从这里开始遍历

1.14.28–1.14.31

and we say ,oh I want to go to this this node down here

我们会说，Oh，我想跑到下面的这个节点处

1.14.31–1.14.32

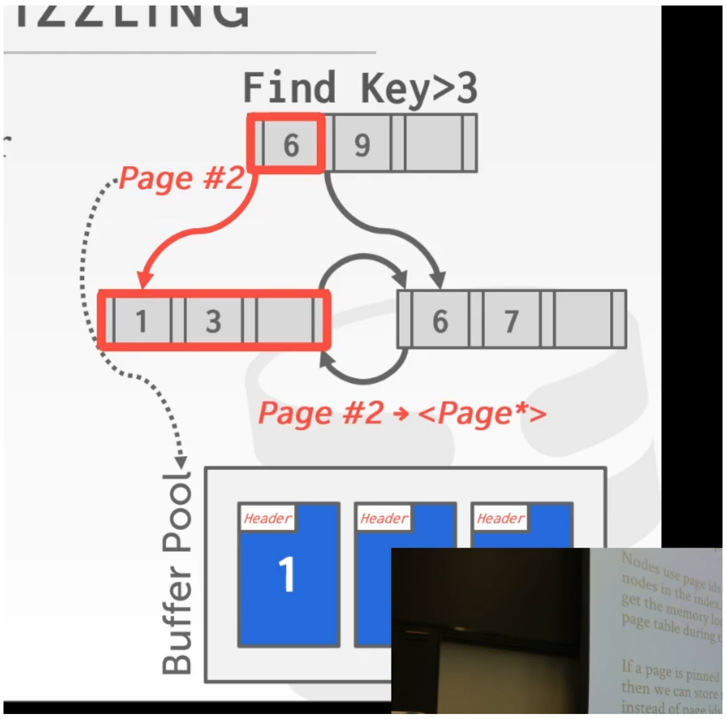
well how do we actually get there

Well，我们实际该如何到达那里呢?

01:14:33 – 01:14:38

Well in the root node ,I'm storing a page ID for this index

Well，在根节点处，我们保存了该索引的page id



1.14.38–1.14.40

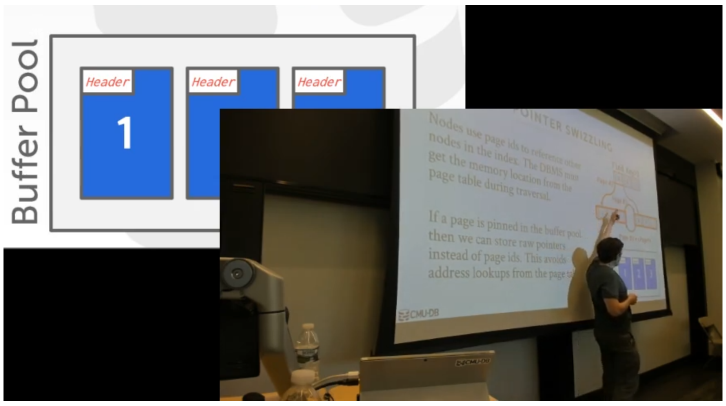
and now I've got to go down to the buffer pool and say

现在，我们跑到下面的buffer pool中去，并表示

1.14.40–1.14.41

hey I have page #2

hey，我想要page #2

1.14.41–1.14.47
~~if it's in memory,~~ if it's not a memory，go get it for me I'll and and then give me back a pointer to it
如果它不在内存中，那么请帮我找到它，并给我一个指向它的指针
01:14:47 – 01:14:50
So then I go get now my pointer to it and now I can do my traversal
So，现在我拿到了指向它的指针，于是我就可以进行我的遍历了
01:14:50 – 01:14:52
Same thing as I'm scanning along here
当我沿着这里扫描的时候，也是做一样的事情


01:14:52 – 01:14.53
I want to get to my sibling
我想拿到我的兄弟节点
1.14.53–1.14.56
this is my sibling is page three
我的兄弟节点的page id是page #3
1.14.56–1.14.58
because that's what's stored in my node
因为这是我节点中所保存的东西
1.14.58–1.15.01
I got to go down to the buffer pool and say,give me the pointer for page three
我会跑到下面的buffer pool中，让它把page #3的指针给我
01:15:02 – 01:15:04
So as I'm traversing
So，当我在遍历时
1.15.04–1.15.11
I keep going back to the buffer pool manager and saying do this conversion from page ID to pointer
我会去回到buffer pool管理器这边，并让它将page id转换为指针
01:15:11 – 01:15:13
And this is really expensive
其实这样做的代价真的很高
1.15.13–1.15.19
because I got it you know I had to protect my hash table in my and my buffer pool with latches
因为我必须使用latch来保护我buffer pool中的hash table
1.15.19–1.15.22
and therefore I'm going to much the steps just to get this pointer
因此，我要花很多步才能拿到这个指针
01:15:23 – 01:15:25

So with pointers swizzling the idea is that
So，pointers swizzling的思路是
1.15.25–1.15.27
if I know all my pages are pinned in memory
如果我知道所有固定在内存中的page的话
1.15.27–1.15.31
meaning I know it’s not going to be going to be evicted
这就意味着，我知道它们不需要从内存中移除出去
01:15:31 – 01:15:35
Well instead of storing the the page ID
Well，这里我并不会保存page id
1.15.35–1.15.36
I’ll just replace it with the page pointer
而是我将它用page指针来替换
1.15.36–1.15.42
because I know if it’s pin it’s never gonna move to a different memory address
因为我知道，如果page被固定住，那么它永远不会移动到另一个内存地址上去
01:15:42 – 01:15:45
So now when I do traversals instead of doing that lookup to the buffer pool
So，现在当我进行遍历时，这里并不是在buffer pool中进行查找
1.15.45–1.15.48
,I have exactly the page ID or the page pointer that I want
我拥有具体的page id或者是我想要的那个page指针
01:15:48 – 01:15:50
And I can go get exactly the data that I want
那我就能准确地拿到我想要的数据
1.15.50–1.15.52
and I don’t have to go ask the the buffer pool
这样我就无须去询问buffer pool了
01:15:53 – 01:15.57
Of course now I’m gonna make sure that if i evict this thing, I write it out the disk
当然，现在我要确保，如果我将它从内存中移除，也就是将它写出到磁盘上
1.15.57–1.15.58
I don’t store the page pointer
我不会保存page指针
1.15.58–1.16.00
because when it comes back in that’s gonna be completely different
因为当它再放入内存的时候，它的地址就完全不同了
01:16:00 – 01:16:02
So you don’t blow away the page id entirely
So，我们无须将page id完全删除
1.16.02–1.16.06
it’s just you have a little extra metadata say, here’s the pointer you really want not the page number
你可以通过一点额外的元数据来表示，这里是我们想要的指针，而不是page id
01:16:08 – 01:16:13
So you may say alright~~ when would we actually pay~~ when would actually would be pinning these pages in memory
So，你们可能会说，当我们将这些page固定在内存中时

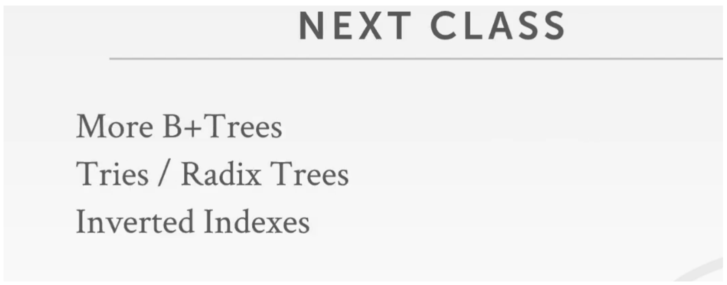
01:16:14 – 01:16:16
Well maybe not for the leaf nodes
Well，我们可能固定的并不是叶子节点
1.16.16–1.16.18
but at least for the upper levels on the root and maybe the second level
但至少会是二叉树的上层部分，比如根节点，或者也可能是树的第二层处的节点
1.16.18–1.16.20
those things are gonna be super hot
这些节点使用的频率超级高
1.16.20–1.16.22
because I'm always gonna have to go through them to get down to the leaf nodes
因为我始终得通过它们才能到达叶子节点
01:16:23 – 01:16:25
So maybe it's not that big of a deal for me to pin those pages
So，对于我来说将这些page固定住可能并不是什么大问题
1.16.25–1.16.30
and they're gonna be relatively small compared to the size of the entire tree
比起整个树的大小来看，相对而言，它们的体积还是比较小的
01:16:30 – 01:16:34
and then I can use this optimization stratgy, because I know my pointers are always gonna be valid
那么我就可以使用这种优化策略，因为我知道我的指针始终有效
01:16:35 – 01:16:39
So this one is actually very common, pointers swizzling is used in in pretty much every major system
So，这种方式实际上非常普遍，很多主流数据库系统中都使用了这种pointer swizzling
01:16:41 – 01:16:41
okay

CONCLUSION

The venerable B+Tree is always a good choice for your DBMS.

01:16:44 – 01:16:44
all right

1.16.44–1.16.45
so to finish up
So，总结一下
1.16.45–1.16.47
the b+ Tree is awesome
B+ Tree是一种非常棒的数据结构
1.16.47–1.16.54
Hopefully I've convinced you that it's a good idea to use this for you you know, if you're building a database system
如果你要去构建一个数据库系统，那希望我已经说服你去使用B+ Tree，这真的是一个很棒的想法



NEXT CLASS

More B+Trees
Tries / Radix Trees
Inverted Indexes

01:16:54 – 01:16.58
next class we'll see some additional optimizations for this
在下一节课中，我们会看到对它的一些额外优化
1.16.58–1.17.01
and maybe do some demos with Postgres and MySQL
我们可能会使用PostgreSQL和MySQL来做一些demo
01:17:01 – 01:17:04
but then we'll also you talked about two other types of tree based indexes we may want to use
但之后，我们也会去讨论两种我们可能想使用的基于树的索引
01:17:06 – 01:17:09
Tries/Radix trees which are gonna look like b–trees were slightly different
在我们看来，Tries/Radix Tree看起来像是B–Tree（B树），但实际有所不同
1.17.09–1.17.10
because that we're not store entire keys
因为我们不会去保存完整的key
01:17:10 – 01:17:13这段之后字幕文件就没了。
And inverted indexes will allowed you to do key searches
接着，倒排索引（inverted indexes）能让我们对key进行搜索
01:17:15 – 01:17:15
Any questions
有任何问题吗?
01:17:18 –
Hit it
散会！