

07-02

20:47 – 20:48

So think about this

So, 思考下这个

20.48–20.51

instead of having a table heap with my tuples

相对于我们所拥有的一张包含了一堆tuples的表

20.51--20.55

and then a B+tree that stores my primary key

我们使用B+ Tree来保存我们的主键

20:55 – 20:59

And so if I want to keep in sync, ~~what if just~~ they were to merge together

So, 如果我想保持同步, 将它们(主键和tuple) 合并在一起

20.59–21.05 !!!

and the leaf nodes what's actually the tables that the tuples, you know corresponding to a primary key

叶子节点, 实际上也就是表中的tuple, 它会对应一个主键

21:05 – 21:09

So now when I want to do traversal to find out a particular key or particular tuple

So, 现在当我想要通过遍历来找到某个特定的key或者tuple时

21:10 – 21:13

Instead of having to do in the first case, I traverse the index get a record ID

无须像第一个例子中所做的那样, 对索引进行遍历, 然后拿到一个record ID

21.13–21.15

then do a lookup in the page table and find that

然后在page表中根据record ID进行查找, 并找到我要的东西

21.15–21.18

and they go scan inside that block to find that tuple that I want

然后跑到这个block中进行扫描, 以此找到我想要的tuple

21:19 – 21:20

What if as I do the traversal

而在这里, 当我进行遍历时

21.20–21.22

when I land in the leaf node

当我落在这个叶子节点上时

21.22–21.24

there's already the data that I want

我就已经拿到了我们想要的数据库

21:24 – 21:27

So MySQL and SQLite probably most two famous ones that do this

So, MySQL和SQLite可能是这方面最著名的例子

21:28 – 21:32

In cases like Oracle and SQL server, I think by default you get the one at the top

比如Oracle和SQ server，默认情况下，应该使用的是方案1

21.32-21.34

but you can tell it to do this at the bottom

但你也可以让它们使用第二种方案（底部这种方案）

21.34-21.37

now you get the pass special flags

你只需传入特殊的标志符即可

B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys + values in all nodes in the tree.

→ More space efficient since each key only appears once in the tree.

A **B+Tree** only stores values in leaf nodes. Inner nodes only guide the search process.

21:37 – 21:41

So now I want to distinguish since we understand the basics about B+tree

因为我们已经对B+ Tree有所了解

21:42 – 21:44

Let's distinguish it from the original B-tree

So，让我们将它和原始的B-Tree（B树）区分开来

21:45 – 21:47

So the major difference is that

So，它的主要区别在于

21:47-21.48

in the original B-tree

在原始的B-Tree（B树）中

21.48-21.56

the values of the stored the index could be anywhere in the tree

value可以存放在树的任何位置

21:56 – 22:00

Meaning any inter node could also have a value like a record ID or tuple actually for themselves

这意味着，任何inner node也可以保存诸如record id或者tuple之类的value

22:01 – 22:05

In the B+tree,the values are only in the leaf nodes

在B+Tree中，value只能放在叶子节点中

22:06 – 22:07

So what are the implications of this

So，这其中的含义是什么呢？

22.07-22.09

Well one

Well，第一点

22:09 – 22:10

In the B-tree case

在B-Tree（B树）中

22.10-22.12

I don't have any duplicate keys

我不会有任何重复的key

22:12–22:19

because I can guarantee that each key will only appear once in my tree

因为我可以保证在我的树中，每个key只出现一次

22:19 – 22:19

In the B+tree

在B+Tree中

22:19–22:23

because I have all this guideposts up above in the inner nodes

因为我会将所有的路标都放在inner node中

22:23–22:24

I'm basically duplicating keys

也就是说，我会有重复的key

22:26 – 22:32

Furthermore, if I delete a key in a B+tree, I would remove it from the leaf node

此外，如果我删除了B+Tree上的一个key，我会将它从叶子节点上移除

22:32 – 22:34

But I may not actually remove from the inner nodes

但实际上我可能并不会将它从inner node上移除

22:34–22:37

depending whether our rebalance or not

这取决于我们是否要进行重新平衡

22:37–22:40

right there I may not have a path going down to it

我们可能也就没有通向它的一条路了

22:40 – 22:41

I am sorry

抱歉，我说错了

22:41–22:45

by deleting from the leaf node, I may keep it in the inner node

当我将它从叶子节点上删除时，我可能会将它保存在inner node中

22:45–22:47

because that's how I figure out what path to go down, if I'm looking for other keys

因为如果我要查找其他key，那么我还可以通过这条路线往下去查找

22:48 – 22:54

So A b-tree is gonna be more economical and how much storage space it occupies

So，相比之下，B-Tree更加经济，占用的空间也少

22:54–22:55

because it's not duplicating keys

因为它不会对key进行复制

22:56 – 23:01

But the downside is gonna be and this is why that nobody and end up actually using this

in a real system

但之所以最终没人在真实的系统中使用它的原因是

23:01–23:06

it's that it makes doing updates more expensive when you have multiple threads

当你使用多个线程来进行更新操作的时候，这样做的代价会更加昂贵

23:06 – 23:09

Because now you could be moving things up and down

因为你可以将东西上下移动

23:09 – 23:12

Right, the tree you know I have an inner node I modify something

比如说, 我有一个inner node, 我对它进行某些修改

23:12–23:16

and I made it propagate a change below me and above me

然后, 我将这个修改向上和向下进行传播(知秋注: 修改删除某一个inner node所带来的影响, 比如它内部数据结构中相关指针的指向, 这个在并发操作下是需要保护的)

23:16 – 23:18

And therefore I have to take a latches on both directions

因此, 我必须在这两个方向上加一个latch

23:18–23:23

~~and that causes~~ as we'll see you next class or next next week that caused a lot of issues

我会在下堂课或者下下周的时候向你们展示, 这样做所引起的许多问题

23:23 – 23:23

In a B+tree

在B+ Tree中

23:23–23:26

I only make changes to the leaf nodes

我只对叶子节点进行修改

23:26–23:29

I may have the propagate changes up above, but I only go in one direction

我可能需要将修改结果向上传播, 但我只需要一个方向就可以

23:29 – 23:33

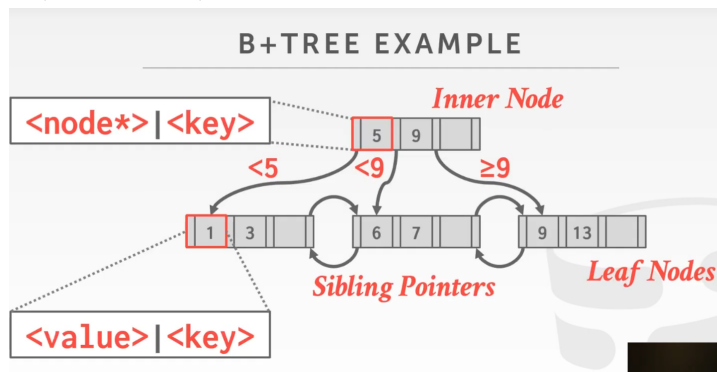
Yes,..... yeah

请问

23:35 – 23:38

So the question is can I repeat what I said about duplicates in a B+tree

So, 她的问题是, 我能不能再讲下B+ Tree中的重复项这种情况



23:38 – 23:44

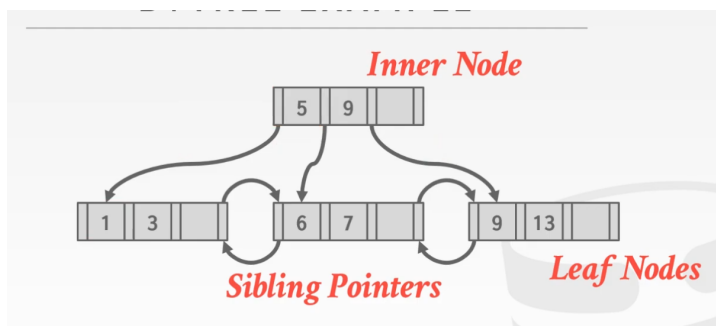
So going back to, to this guy here

So, 我们来看这张幻灯片

23:45 – 23:47

So this is the B+tree

So, 这是一个B+ Tree



23:47 – 23:53

So the keys that I have that I'm trying to index are 1 3 6 7 9 13

So, 我所拥有并试着去索引的key是1, 3, 6, 7, 9, 13

23:53 – 23:55

But if you look in the root node

但如果你看下根节点

23:55–23:56

I have a 5

我有一个5

23:56–23:59

5 does not appear anywhere in the leaf node

5并没有出现在任何叶子节点上

23:59–24:02

meaning ~~hey probably got~~ in this case here

意味着, 在这个例子中

24:02–24:05

but had gotten inserted and then it got deleted

但当我将它插入, 接着又将它删除后

24:05 –24:08

But I didn't reshuffle or reorganize my my tree

但我并没有将我的树进行重新整理或者是重组

24:08 – 24:10

So I left it in the inner node

So, 我将它留在了inner node中

24:11 – 24:12

In a b-tree

在B-Tree (B树) 中

24:12–24:14

that'll ~~never~~ happen each key only appears once

一个key只出现一次

24:14–24:16

and any if it appears in the tree

如果它在树中出现的话

24:16–24:17

then it appears in our key set

那它就会出现在我们的key的集合中

24:18 – 24:18

So make sense

So, 懂了吗?

24:24 – 24:30 ! ! ! ! ! !

The question statement is we leave in here for searching purposes ,and it's still stored physically in our nodes

她的问题是，出于搜索的目的，我们将这个5放在了树里面，它依然以物理的方式保存在我们的节点中

24:30 – 24:34

But if I asked this tree do you have key 5

但如果我去问这棵树，它里面有没有值为5的key

24:34–24:35

I would say no

我会说No

24:35*–24:37

because I always have to go to the leaf node

因为我始终都得跑到叶子节点上来找想要的

24:37–24:39

then I try to find 5, and I'm not gonna find it

然后，我试着找到5，但是我找不到它

24:39 – 24:43

So it still be there ,but it's it's not actually a real key

So，它虽然依然在那里，但实际上它并不是一个真实的key

24:44 – 24:44

Yes, yes

请问

24:49 – 24:54

Okay, so question is how do we do with inserts when we felt on the leaves we'll get that in a second, yes

Ok, So，他的问题是，当我们在叶子节点上时，我们该如何进行插入，我们会在稍后看到这个

24:52–24:54

that's the next topic

这是下一个话题

24:58 – 25:01

This question is will there not be any duplicates in the leaf nodes

他的问题是，在叶子节点中是否存在任何重复项

25:03 – 25:04

Yes and no

可以说有，也可以说没有

25:04 – 25:05

So we'll see in a second

So，我们会在稍后看到

25:05–25:11

So this this would be considered a unique index, a unique tree or they're unique keys

So，这可以被认为是一个唯一索引，唯一键的树或者是唯一键

25:11 – 25:13

You can't have keys that have non unique values

我们的key的值必须是唯一的

25:13–25:17

we have to handle that.we'll get to that in a second as well

不然，我们必须对其进行处理，我们会在之后看到

25:17 – 25:17

Okay

25.17–25.24

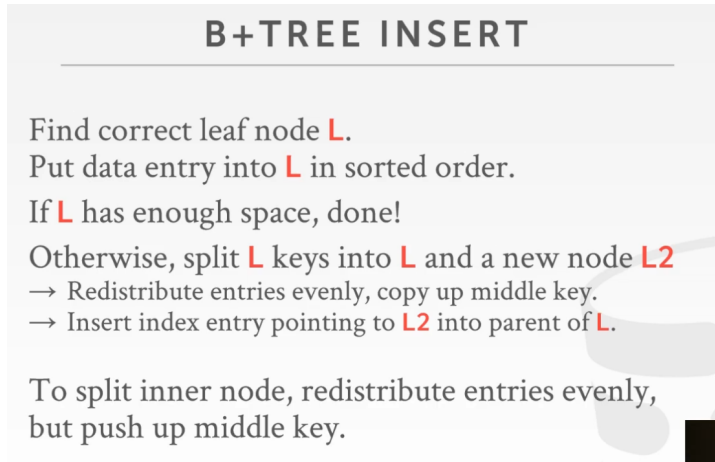
so I think the next topic is what he was yes is how do we actually how to actually modify this

So, 我觉得下一个课题就是该如何修改这个

25.24–25.25

absolutely yes, inserts, okay

确实没错, 我们要来讲insert (插入) 操作



B+ TREE INSERT

Find correct leaf node **L**.
Put data entry into **L** in sorted order.
If **L** has enough space, done!
Otherwise, split **L** keys into **L** and a new node **L2**
→ Redistribute entries evenly, copy up middle key.
→ Insert index entry pointing to **L2** into parent of **L**.

To split inner node, redistribute entries evenly,
but push up middle key.

25:27 – 25:30

So the way we're gonna do an insert is that,

So, 我们进行insert 操作的方式是

25.30–25.35

we want to find that we want to traverse down,

我们先往下遍历

25.35–25.39

and figure out what leaf node we want to insert our new key into

并弄清楚我们想在哪个叶子节点上插入我们新的key

25:39 – 25:42

So again we use those guideposts on the inner nodes decide whether we go left or right

So, 我们会使用inner node中的这些路标来决定我们是往左走还是往右走

25.42–25.48 ! ! ! !

depending on whether a key is less than or greater than what's stored in those key arrays

这取决于我们要插入的key是比保存在这些key数组中的值大还是小了

25:48 – 25:51

And then as we traverse down eventually we'll get to a leaf node

接着, 当我们往下遍历时, 我们最终会到达一个叶子节点

25:52 – 25:55

And then the leaf node is where we want to insert a key

这个叶子节点所在的位置, 也就是我们想插入该key的位置

25.55–25.56

and so if the leaf node has space

如果该叶子节点上还有空间

25.56–25.58

then we just inserted in

那么我们就直接插入即可

25.58–26.02

for keeping the keys in sorted order, maybe we should we sort them

为了让这些key保持有序性, 我们可能需要对它们进行排序

26:02 – 26:04

But there's not enough space we just insert it

但这里没有足够的空间来让我们插入这个key

26:04 – 26:06

If there's not enough space

如果该节点并没有足够的空间来让我们插入这个key

26:06–26:08

then we have to split the node

那么，我们就必须拆分这个节点

26:08–26:11 ! ! ! ! !

let's split the leaf node we would we just inserted into

我们将叶子节点拆分开来后，我们将key插进去

26:11 – 26:12

And so the way we're gonna do this

So，我们做到这点的的方法是

26:12–26:15

we're just gonna take a halfway point in our key space

我们找到叶子节点的中间位置

26:15–26:19

put all the keys that are less than the halfway point in one node

我们将中间位置左边的所有key放入一个节点

26:19–26:22

all the keys that were above that in another node

将右边所有的key放入另一个节点

26:22 – 26:28

And then we update our parent node to now include that that middle key

然后更新我们的父节点，让它包含这个中间key（知秋注：这个中间key其实就可以看作是一个叶子节点的边界key了）

26:29 – 26:32

And then we have an additional pointer to our the new node we just added

接着，我们会有一个额外指针来指向我们刚添加的新节点

26:33 – 26:36

And that may be happy to say,all right, well this is actually a recursive thing

这个可能会是一个递归的过程（知秋注：逐层往上更新父节点）

26:36–26:40

because if now my parent as I try to insert the new key in to the parent

因为如果我现在要将新的key插入到父节点中

26:40 – 26:42

If it doesn't have no space

如果它并没有空间来让我们插入

26:42–26:42

then we have to split it

那我们就得对它进行拆分

26:42–26:45

and then propagate the changes up above

然后将修改后的结果往上传播

26:45 – 26:49 ! ! ! ! !

So for one insert we may have to reorganize the in the entire tree

So，对于一次插入来说，我们可能就得要重新整理下整个树了

26:49 – 26:52

And this is what I was saying before like like just like in the hash table

这就有些像我之前在hash table中所说的一样

26:52 – 26:55

If I insert into it index or through the hash table and nothing's there

如果我将一个索引插入到hash table上，并且该位置没有任何东西

26:55–26:56

it's really fast

那速度就非常快

26:56–27:01

But if I have to scan a long long time to find the slot I can go into, that can be more expensive

但如果我需要花很长时间才扫描并找到一个我可以插入的slot，那么插入的代价就很高了

27:01 – 27:03

So sometimes we would insert into our tree

So，有时，当我们往我们的树中插入元素

27:03–27:04

and it's gonna be an expensive operation

这会是一个要付出昂贵代价的操作

27:04–27:07

because we're reorganizing the entire data structure

因为我们要重组整个数据结构

27:07 – 27:10

And other times it'll be super fast and we don't have to worry about it

其他时候，我们的插入速度会非常快，并且我们无须为此担心

B+ TREE VISUALIZATION

<https://cmudb.io/btree>

Source: [David Gales \(Univ. of San Francisco\)](#)

27:12 – 27:16

All right, so let's do a let's view a demo of this

So，我们来看个demo

27:16 – 27:21

So this is using this is a you know rather than me doing animations in PowerPoint

So，这里我不会在幻灯片上进行动画演示

27:21 – 27:37

This is from a professor at university San Francisco that has a nice you know a little web-based visualization we can use to, carry, yes okay

这是旧金山大学一个教授所做的基于Web的可视化demo

27:44 – 27:46

No out of type remotely

B⁺ Trees

Insert

Delete

Find

Print

Clear

☒ Max. Degree = 3

☐ Max. Degree = 4

☐ Max. Degree = 5

☐ Max. Degree = 6

☐ Max. Degree = 7

27:47 – 27:50

Alright, so we'll do a max degree of three

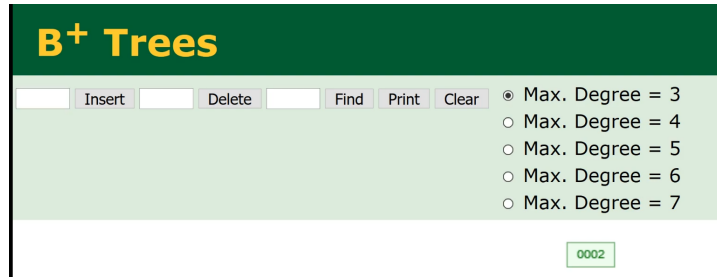
这里我们选Max Degree = 3

27:50–28:02

so that means that the the max number of nodes, we can have is two or sorry, keys in our each node is two and can have at most three paths going down

So, 这意味着在我们的node中key的数量为2, 并且我们最多只有3条向下的路径

So, 这意味着在我们的每一个node中, key有2个, 并且最多只有3条向下的路径



28:02 – 28:09

So we insert can ever see that we insert 2, know if it's a physician mean

So, 这里我们插入一个2

28:09 – 28:09

Yes,

请问

28:12 – 28:14

So the degree says the number of paths coming out of it

So, 这里的degree指的是从一个节点处出来的路径数

28:15 – 28:19

So degree of three means I have most three paths coming out of me,

So, degree of 3的意思是, 从我这里最多能出来3条路线

28:19–28:20

if I'm an inner node

如果我是一个inner node

28:20–28:23

and therefore I had to store it I can store at most two keys

那么我最多能保存2个key

28:25 – 28:30

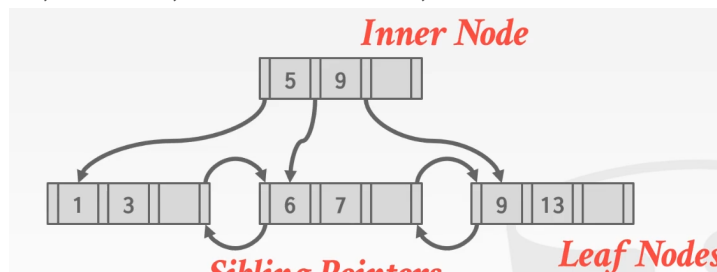
~~Because, again so good~~ I mean going back to what we showed in the very beginning

回到我们一开始讲的地方

28:33 – 28:36

Question, why do I set to three or why is it that way

So, 问题来了, 为什么我设置的是3, 我为什么这样做?



28:40 – 28:45

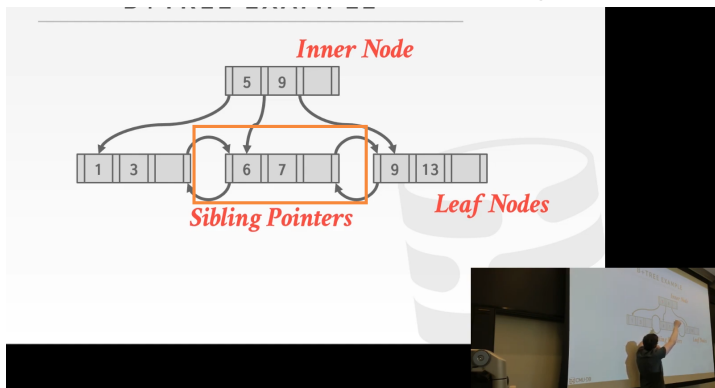
So again so this is say this is that a this has a degree of four

这张图上的degree其实是4

28:45 – 28:49

So it's always the number path is the number of keys plus one

SO, 路径的数量始终是一个节点所能保存的key的数量加1



28:49 – 28:51

So I be 1 2 3 keys

So, 这里三个key

28:51–28:56

and this guy has to have a right pointer and a left pointer

So, 这个节点必须要有一个左指针和一个右指针

28:56 – 29:00

Right, and he has have a right pointer ,but that's shared and there's the one at the end

这个右指针指向的是下面最后一个节点

29:00 – 29:03

So this there's four paths coming out for three key's

对于有3个key的node来说, 它可以分出四条路径来

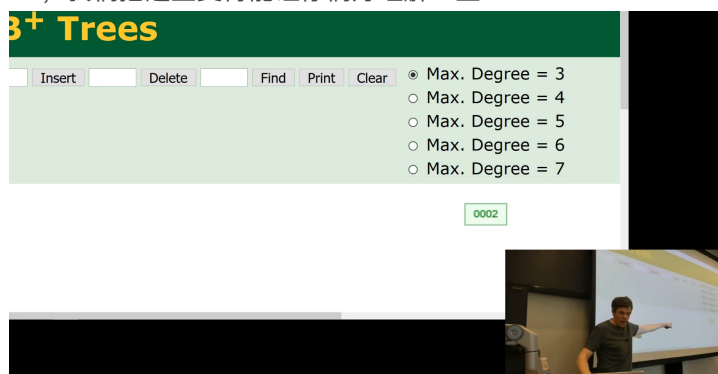
29:03 – 29:08

Okay, alright

29:08 – 29:10

So is there a way to make this looking better

So, 我们把这里变得能让你们好理解一些



29:12 – 29:14

Well, let's just keep going see how go so it's down over there

Well, 我们只需要一直关注图中的右下角图形展示

29:14 – 29:21

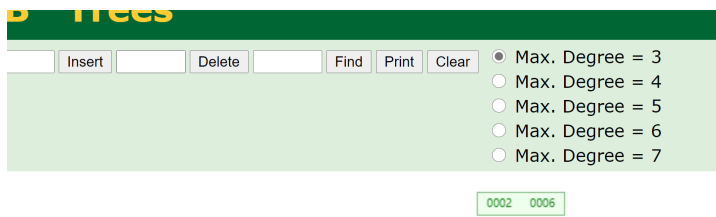
So I've only inserted ,and it's the demo ,I've only started two keys or one sorry one key

So, 我从一个key开始

29:24 – 29:26

So right now it only has one entry in it

So, 现在它里面只有一个条目 (即0002)



29:26 – 29:32

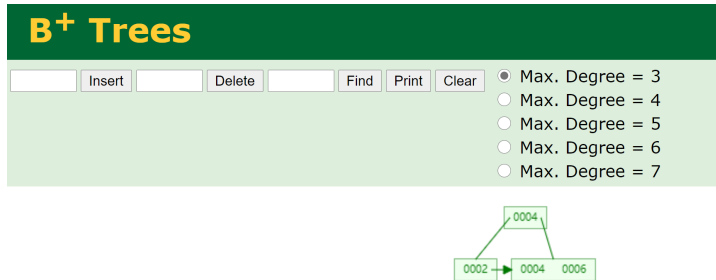
So now I'll insert, we insert six

So, 现在我们插入6

29:34 – 29:38

Right, so again it just it had space in that node, so I was able to insert it

So, 因为在该节点中还有空间, So, 我可以往里面插入数据



29:38–29:41

and now I insert four

现在, 我插入4

29:41 – 29:42

And at this point it has to split

此时, 我们必须对该节点进行拆分

29:42–29:48

because it can only you can only store, it can only store two keys

因为一个节点只能保存2个key

29:48 – 29:54

So it's split in half. put two over here four and six six in its own node

So, 我们把它一分为2, 我们将2放在一个节点, 4和6还放在它原来自己的节点内

29:54–29:58

and then they took the middle key for and moved it up as the new root

然后, 我们将中间key移到上面作为新的根节点

29:58 – 30:00

And again I have pointers going down to both of them

这里, 我有两个向下指向它们的指针



30:01 – 30:03

So now do it insert 1

So, 现在我们来插入下1

30:04 – 30:07

Right, back in fit over there accommodate just fine

可以看到这个5能够放在这里

如图所示, 可以看到插入结果

30:07 – 30:09

It's now insert 5, what should happen

现在我们来插入5，看看会发生什么

30:10 – 30:13

Right, it'll say well 5 is greater than 4

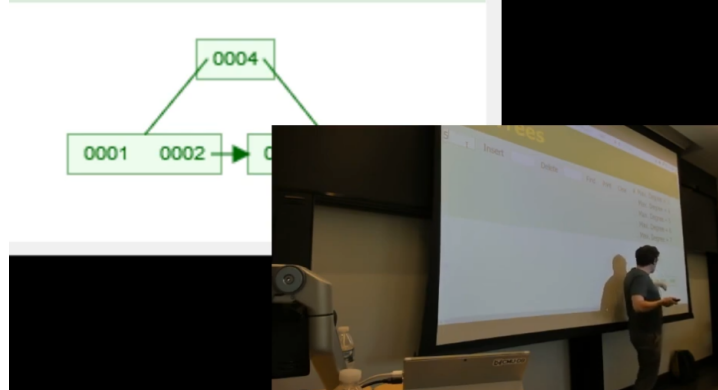
可以看到，5比4大

30:13 – 30:15

It's greater than equal to 4

它大于等于4

- lear
- Max. Degree = 3
 - Max. Degree = 4
 - Max. Degree = 5
 - Max. Degree = 6
 - Max. Degree = 7



30:15–30:16

so I no need to go down this direction

So，我无须沿着这个方向往下

30:16–30:24

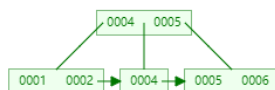
~~but I can only sort, I can always~~ I can only store two keys in this node

但在这个节点处，我只能保存两个key

30:24 – 30:27

So I'm gonna have to split this guy's and then rebalance everything

So，我必须将这个节点拆开，对树重新进行平衡



30:28 – 30:32

So ~~hit enter~~, right 4 goes on there ,puts 5 there

So，4跑这里来了，把5放到那里

30:32 – 30:34

Right, it's split then split the node

这里我们将节点拆开

30:34–30:36

put 4 in the middle over here

我们把4放到中间

30:36–30:36

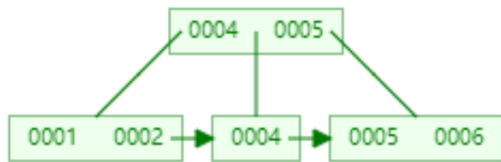
5 and 6 over here

5和6放在那边

30:36–30:40

and then put 5 up,because that was the middle key

然后把5放到上面，因为它是中间的key



30:40 – 30:44

And now we have pointers going to this node, the middle node here with 4 and that one 5

现在，我们就有一个指针指向中间节点上的4，右边则是5

现在，我们分别有三个指针指向下面这三个节点

30:46 – 30:52

Right, so again this is recursive as I keep inserting more stuff ,and I keep splitting, I keep splitting the changes up, yes

So，这是一个递归的过程，随着我插入更多的元素，我会对节点继续拆分，并将这些变化向上传递

30:54 – 30:56

So he says what if we have duplicate keys

So，他想问的是，如果有重复的key怎么办

30:56 – 31:00

So actually I don't know whether this will matter, so I just insert four

So，实际上我也不知道这是否重要，So，这里我插入个4看看情况

31:04 – 31:04

Yeah, it did that

这句不要

31:05 – 31:08

Um, so there's different way sorry how do i

不好意思

31:09 – 31:11

The resolution is rejected

我们的提交被拒绝了

31:11 – 31:15

Mmm, 11 now

试一下11看看

31:17 – 31:19

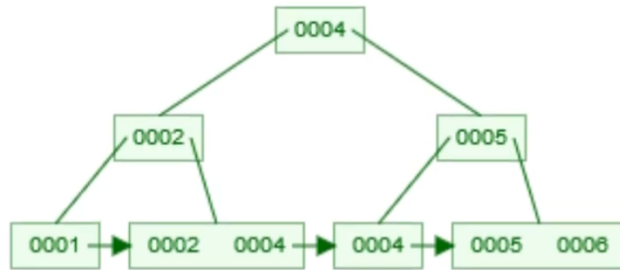
How did I do that ,sorry

我是怎么做到的？？？

31:25 – 31:26

There we go okay sorry

好了，我们来看一下



31:26 – 31:30

So this is just sort of a toy diagram

So, 这里所展示的只是一个示意图

31.30–31.34

in a real system you could store for together

在真实的系统中，你可以将它们存放在一起

31.34–31.39

and just maintain multiple entries for all the unique values of that you have the same key

只需要维护拥有同一个key的所有唯一值的多个条目即可

只需要维护拥有同一个key下多个条目所对应的值（每个条目对应一个唯一值）即可

31:54 – 32:01

So okay your statement is if all my keys are the same, it's 4,4,4,4,4,4

So, 你的问题是，如果我所有的key都一样，都是4

32.01–32.02

then if I'm looking for an exact key value pair

接着，如果我要寻找一个准确地key/value pair（键值对）

32.02–32.06

then it's log n, because I do its sequential scan, yes

那么它的复杂度就是 $O(\log n)$ ，因为我使用的是循序扫描

32:06 – 32:08

So yeah, we can pop up Postgres

So, 我们可以来看下PostgreSQL是怎么做的

32:09 – 32:11

We can make a table has a billion rows,

我们可以创建一个有10亿行的表

32.11–32.15

and for one column we set the value to one

我们将其中一列的值都设置为1

32:15 – 32:19

~~And we can call create,~~ you know every so every where every 1 billion row has the same value for that one column

So, 当这10亿行数据在那一列上的值都是一样的时候

32:20 – 32:24

And PostgreSQL will let us build an index on that column

PostgreSQL会让我们在这个列上构建出一个索引来

32:24 – 32:28

It's a stupid to Build, because as you said they're all the same

这样构建其实很蠢，因为我们说过，它们这一列的值都是一样的

32:29 – 32:32

so input it so and so how to say this

So, 我该怎么说呢

32:33 – 32:35

People will do stupid things

人们会做些愚蠢的事情

32.35–32.40

in general don't be stupid and don't build indexes on things that you shouldn't use

一般来讲，请不要犯蠢，不要在你不怎么使用的东西上构建索引。

32:40 – 32:42

Right, there's the other types of indexes we'll see

我们以后会看见其他类型的索引

32.42–32.45

so a hash table there's other things like inverted indexes we could use

我们还可以使用诸如反向索引（inverted index）这样的东西

32.45–32.47 ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !

that could be better if you had a lot of duplicate values

如果你有太多的重复值，使用这个（反向索引）可能会来得更好

32:47 – 32.54

But think of like email addresses or think of like phone numbers or things that work it's

gonna be vastly diverse

但像电子邮件地址，手机号之类的，它们是千差万别的

32.54–32.56

then we won't really have that problem

那么，我们就不会遇上这种问题

32.56–33.01

or primary key right, primary key has to be unique that would be great for this

还有主键，主键必须是唯一的，那么对于索引来说，这最好不过

33:01 – 33:03

Alright, so again so this is this clear

So, 你们听懂了吗

33:05 – 33:07

Okay, so let's go back

Ok, So, 让我们回到这里

33:10 – 33:13

So to do deletes now, we have the opposite problem

So, 如果我们进行删除操作, 我们就会遇上相反的问题

33:13 –33:18

Again insert so if we got to full, we run up space, we have to do that split

对于插入操作来说，如果我们的空间满了，耗尽了所有的空间，那我们就得进行拆分

33:19 – 33:20

If we delete

如果我们进行删除操作

33.20–33.24

then it may be the case we end up being less than half full

我们就会遇上节点上所保存的key的数量小于半满的情况

33.24–33.27

which would violate the guarantees we have to have in our B+tree

这就会违反我们在B+ Tree中所必须遵守的规则

33:27 – 33:29

And then therefore we have to do the opposite of a split which is a merge

因此，我们就得做拆分的逆向操作，也就是合并

33:30 – 33:37

So delete something, again I just do my traversal, I go down the tree try to find the key that I want to delete

So，如果我要删除某些东西，我先进行遍历，然后跑到树的底部试着找到那个我想删除的key

33:38 – 33:40

I'll and I'm always gonna land in a leaf node

我始终会落在一个叶子节点上

33:40–33:46

if my leaf node after deleting that key is still at least half full, then I'm done

当我删除了那个key后，如果我的叶子节点依然处于至少半满的情况，那么删除就完成了

33:46 – 33:50

I just remove it, maybe reorganize my my sort of key arrays, but then that's it

我只需把key移除就行，可能还需要整理下key数字，但基本就这些东西

33:51 – 33:52

But if I'm less than half full

但如果该节点并没有处于半满状态

33:52–33:56

then now I have to figure out how to get rebalanced

那么，现在我就得去弄清楚我该如何对树进行重新平衡

33:57 – 34:07

So the sort of one easy trick we could do is look at our siblings, in other leaf nodes and that's why we have those sibling pointers

So，我们能使用的一个简单技巧就是看下临近的其他叶子节点中的元素，这就是为什么我们有这些兄弟指针的原因了

34:07 – 34:11

We can look at them and try to steal one of their keys to make ourselves balanced

我们可以去看下这些节点，并试着从它们中抢一个key回来，以此让树变得平衡

34:11 – 34:17

Right, as long as that our sibling has the same parent as us, then it's okay for us to steal this

只要我们的兄弟节点和我们的节点有相同的父节点，那么我们就可以去打劫它们中的key

34:17 – 34:20

Because that doesn't require any rebalancing up above

因为这就不需要对上面这些节点做任何重新平衡的操作了

34:21 – 34:26

So if we're not able to steal from our sibling, then we have to merge

So，如果我们没法从我们的兄弟节点那里打劫到key，那么我们就得进行合并操作

34:27 – 34:32

I think we go take one of our siblings combine all our keys together

So，我们将其中一个兄弟节点中的key和我们当前节点中的key合并在一起

34:32–34:35

that may actually end up being too full as well

那么实际上，合并后的节点可能会太满了

34:35 – 34:40

But then we could split that as above as well that's the same thing as just copying this

但之后我们可以将它拆分开来

然后我们就可以像前面讲的一样来进行拆分

34:40 – 34:45

But we would merge, delete a key up above, and then now where everything's balanced
但如果我们删除了上面的某个key，我们会进行合并操作，然后B+ Tree中所有的东西都平衡了

34:45 – 34:47

Again, again just like in splits

比如，在拆分的时候

34.47–34.49

we're like I may have to go propagate the change everywhere

我们必须将我们所做的修改传播到B+ Tree的所有地方

34.49–34.53

when we merge and with deleting keys that our parent now may become less than half full

当我们进行合并和删除key时，我们的父节点可能会小于半满的状态

34:53 – 34:54

And it has to merge

那我们就必须要进行合并操作

34.54–34.57

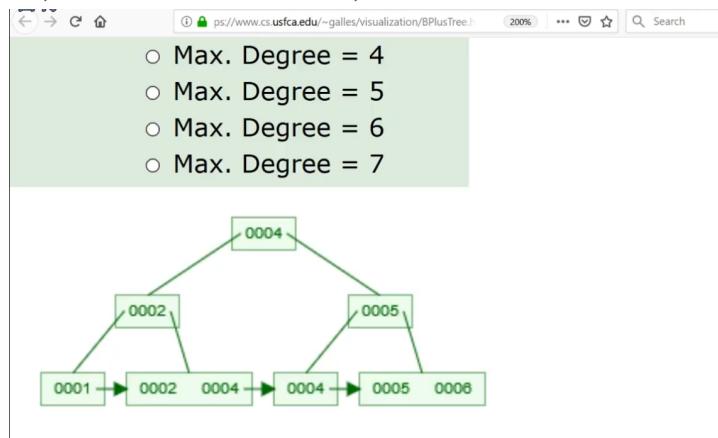
and that for we have to maybe restructure the entire tree

我们可能得重构整个树

35:00 – 35:04

All right, so let's go back and to our example here, and do our demo

So, 我们回到我们的例子中去，来看下我们的demo



35:05 – 35:09

Because now I got to figure out how to get to the top right corner

因为我总算知道怎么把屏幕移到右上角了

35:15 – 35:19 学生提问，后半段没听出来

So we just maintain the siblings in ? ? ?

35:19 – 35:22

Correct, he's question is to be maintained the siblings only in the leaf nodes, yes

没错，他的问题是，我们只在叶子节点处维护兄弟节点?说的没错

35:24 – 35:28

All right, so look to let's do delete 4, I want **sorry** delete 5

So, 我们来删除下5

35:28 – 35:30

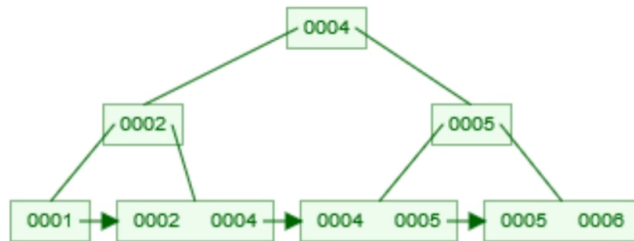
Let me scroll down and then hit enter

我把屏幕移下来点，再来看下效果

35:30 – 35:31

So we can see this

So, 我们可以看到这个



35:33 – 35:36

Alright, so just a traversal are those insert sorry

So, 我们来遍历下, 不好意思这是插入

35:40 – 35:41

Delete 5

删除5

35:46 – 35:50

Yeah, that's insert as Polly again sorry for the low resolution

我没错, 这个框是删除, 那个框是插入

35:51 – 35:52

All right so it's delete 5

So, 我们来删除5

35:53 – 35:55

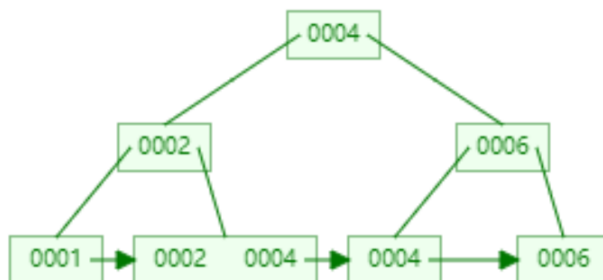
In this case here it should be fine

在这个例子中, 应该没啥问题

35:55–35:57

~~both him there's only found one of them ,so let's try the other one~~

So, 我们来试下另一个



35:59 – 36:01

Goes down that's fine

向下遍历, 看来没问题

36:01–36:03

again at this point here

在此处

36:03–36:05

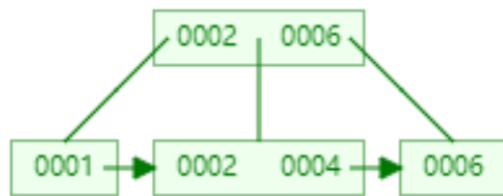
both these nodes are still more than half full ,so that's fine

这两个节点的状态都是半满以上, So这没问题

36:06 – 36:07

So now let's delete four

So, 现在我们把4删掉



36.07–36.13

and I suspect it will try to delete the one that's farther on that side

我怀疑它会试着删除这边这个东西

36:16 – 36:17

Go down found that deletes that

向下遍历，找到4，删除4

36.17–36.20

again that node is now half empty

再说一遍，这个节点现在是处于半空状态

36:22 – 36:24

I mean it has to have at least one

我的意思是它至少必须要有一个key在里面

26.24–36.30

and because it was empty, it merged everything and decrease the height of the tree

因为它是空的，它就得将所有东西进行合并，并减少树的高度

36:30 – 36:30

yes

请问

36:37 – 36:40

This question is if only the leaf nodes have sibling pointers

他的问题是，如果只有叶子节点上有兄弟指针

36.40–36.42

then how do you actually do this merge

那实际上，我们该怎么进行合并操作呢？

36:42 – 36:52

So the way it works basically think of the think of a thread going down, it can maintain a stack of what notes a visited as it goes down

它的工作方式是，这里有一条线程会往下遍历，它会维护一个stack，里面包含了它向下所遍历元素的记录

36:53 – 37:01

And we're actually gonna need to do this when we do what's called latch crabbing or coupling as we go down and we take latches

当我们往下遍历，实际上我们会去进行latch crabbing或者是coupling之类的事情

in case we need to reorganize everything

如果我们需要将所有东西进行重组

37:02 – 37:07

And so I have to know what I have to hold latches up, you know at when I go down somewhere I have to hold and latch my parent

我必须先拿到锁，即当我向下遍历某个地方的时候，我必须在我的父节点加上一个latch

37.07–37.10

in case I need reorganize whatever I'm doing down below

如果我需要重新整理下我下面所做的一切事情

以防我需要对我们下面的节点进行重新整理

37:11 – 37:12

So I don't release it until I know I'm safe

直到我觉得没问题了，我就会将锁进行释放

37:13 – 37:14

So I know how I got there

So，这样我就知道我是怎么做的了

37:15 – 37:15

Yes

请问

37:21 – 37:26

Yeah, his question is if there's two siblings to the left the right which one you choose, it depends

他的问题是，如果在左边和右边各有一个兄弟节点，我们该怎么选择？这得看情况

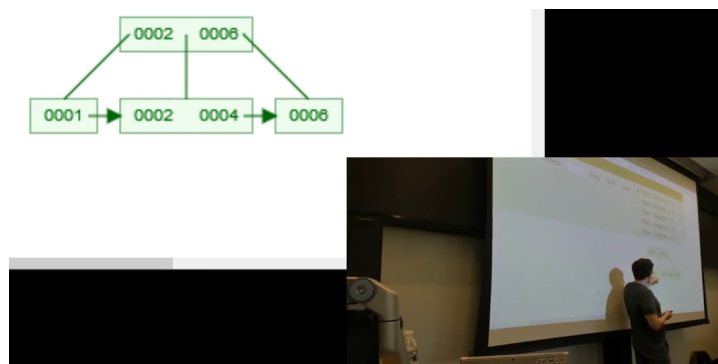
37:26 – 37:29

Right, typically you choose the one that has the same parent as you

通常情况下，我们选择拥有相同父节点的那个来做

37:30 – 37:35

Okay, I think you have two actually But if you're like if you were in the middle, but if you're in the middle, that means both are the same parent, so both are fine



37:35 – 37:36

Yeah, These guys have the same parent

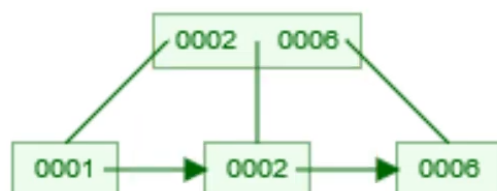
没错，这几个都有相同的父节点

37:36 – 37:40

So you say you want to reorganize this, you could choose to either left to right ,it doesn't matter

So，如果你想对它们进行重组，你可以选择使用左边的节点，也可以使用右边的节点，这都没关系

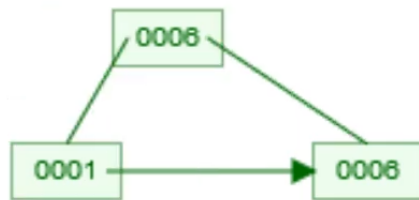
○ Max. Degree = 7



37:41 – 37:45

Let's see what this one does, so we so if we delete 4 that should take it out of the middle

我们来看下，这样做会发生什么，So，如果我们删除4，它应该把中间的移除掉



37:46 – 37:49

And then now I delete two and it's gonna pick people up 2 right

然后，选择我将2删除，上面的2也应该删除

37:51 – 38:00 ! ! ! !

Okay, actually it's can only have one or two, so it might empty, because of its degree of the tree

Okay,事实上，它（这个节点）只能有一个或两个元素，so 上面这个节点可能为空，因为它的degree（为3）

38:00 – 38:03

But like it doesn't matter, it still be correct

但这没关系，这依然是对的

38:05 –38:08

And so this diagram shows the the sibling pointer is going in one way

So，这张图显示，这里的兄弟指针指向的始终是右边

38:09 – 38:10

You can't have a go in both directions

它不可能有两个方向（知秋注：即按照我们的想法，可以有两个指针，一个指向左边，一个指向右边）

38.10–38.13

you have to do extra work to make that happen, but like you can do that

但你可以通过一些额外的工作来做到这点

38:13 – 38:16

A lot of times again for simplicity ,but you could just have a go in one direction,

很多时候，为了简单，我们可以只使用一个方向

38.16–38.24

but then you can't do you know order by in descending order and go the other direction, if you wanna do scans

但如果你想做扫描的时候，你就没法做降序扫描，或者以其他的方向进行扫描（知秋注：针对这里的情况来讲的，只能从小到大）

38:24 – 38:26

Right, pretty straightforward

很简单吧

38.26–38.31

of course getting the the details of the deletes and inserts doing that split merge is actually very difficult

当然，要明白删除和插入的细节以及拆分和合并之类的事情，实际上是非常难的

38:31 – 38:32

In practice

在实战中

38.32–38.39

and we'll see in next week how to actually make sure that when we're reorganizing the tree that we're thread safely

我们会在这周的课上向你们演示当我们在重新整理B+Tree的时候，该如何保证线程安全
38.39–38.41

and we don't have any integrity issues

以及避免那些完整性问题

以避免因为线程安全产生的异常（知秋注：多线程下对公有变量进行非线程安全的操作）

B+TREES IN PRACTICE

Typical Fill-Factor: 67%.

Typical Capacities:

- Height 4: 1334 = 312,900,721 entries
- Height 3: 1333 = 2,406,104 entries

Pages per level:

- Level 1 = 1 page = 8 KB
- Level 2 = 134 pages = 1 MB
- Level 3 = 17,956 pages = 140 MB

38:43 – 38.46

All right, so the in practice

So，在实战中

38.46–38.49

the research shows

据研究表明

38.49–38.55

the typical fill-factor for a real tree on real data is about 67 to 69 %

一个用来保存真实数据的真实二叉树所使用的的标准填充因子的范围在67%到69%

通常情况下，一个树的所使用的标准填充因子范围在67%到69%

38.55–39.04

meaning the amount of data are storing in your nodes that's actually real It's up to you
know 67% of his actually use useful data

也就是说，你节点中所保存数据里，实际上只有67%是有用的（知秋注：其他都是初始数据或未知数据）

39:05 – 39:07

So typical capacities you can have you know

So，所谓的标准容量指的是

39.07–39.12

when for the 8 kilobyte pages

对于一个8kb大小的page来说

Typical Capacities:

- Height 4: 1334 = 312,900,721 entries
- Height 3: 1333 = 2,406,104 entries

39.12–39.21

~~with a~~ this number of pages are at four levels, you can basically store 300,000 key value
pairs

如果树的高度是4层，基本上来讲你可以保存300000个key/value pairs（键值对）

39:21 – 39:32

Right, so you can index and get in log n time to any one of three 300 million keys, very very Quickly

So, 你可以对这些条目进行索引, 并且随机访问这三千万个条目中任意一个的时间复杂度是 $O(\log n)$, 速度真的非常非常快

39:33 – 39:37

And most of the data is going to be stored on the leaf pages as you would expect

正如你们所期望的那样, 大部分数据都是存在leaf page上

39:37 – 39:40

Right, because guys as you add more keys

因为随着你往树里面添加更多的key

38:40–39:41

you start to fan out

它就会开始开枝散叶

39:41–39:45

and most of the data is gonna be stored in those leaf nodes

我们会将大部分数据都保存在这些叶子节点上

#####

CLUSTERED INDEXES

The table is stored in the sort order specified by the primary key.

→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.

→ If a table doesn't contain a primary key, the DBMS will automatically make a hidden row id primary key.

Other DBMSs cannot use them at all.

39:47 – 39:52

All right, so let's talk about some other things you can do with these indexes

So, 我们来讨论下我们可以用这些索引所能干的其他事情

39:53 – 39:58

So this concept of this notion of what called clustered indexes

So, 这里所展示的这种概念叫做聚簇索引 (clustered index)

39:59 – 40:04

And so I said in the beginning that the table heap for a database is unordered

正如我在一开始所说, 数据库中的table heap是无序的 (知秋注: 一张表的数据堆在一起, 就是 table heap)

40:04–40:09

meaning we can insert tuples into any page in any order

这意味着, 我们能以任何顺序将tuple插入到任何page中去

40:09 – 40:11

We don't the follow the you know the temporal order will how things have inserted

我们并不是按照插入的时间顺序进行排序

40:12 – 40:16

But there may be some times where we actually want to have the data sorted in a certain way

但有时，我们实际上想让数据以某种方式进行排列

40.16–40.18

like for example like the primary key

比如，主键 (primary key)

40:18 – 40:20

So these would be called clustered indexes

So, 这被叫做聚簇索引 (clustered index)

40.20–40.23

so you can define an index when you create a table

So, 当你创建一张表的时候，你可以定义一个索引

40:23 – 40:26

You can define a what's called a clustered index

你可以定义一个聚簇索引 (clustered index)

40.26–40.34

and the database system will guarantee that the the physical layout of tuples on pages

will match the order that they're sorted in the index

数据库系统会保证，索引会对page中tuple的物理布局进行匹配排序（知秋注：对磁盘上实际数据重新组织以按指定的一个或多个列的值排序）