b-trees 2.0
prof. Stratos Idreos

HTTP://DASLAB.SEAS.HARVARD.EDU/CLASSES/CS165/
clustered/primary index on A
Btree on other columns: we keep the positions that map on the clustered index at the leaves of the tree.

Secondary index on any column(s)
Page size: 64K - holds 16K 4 byte ints

N elements, P pages

sorted array

info to navigate lower level
value-pointer

4+4 bytes for each page (value+pointer)
64K/8= index 8K pages

12,20
12,15,17
20,...

>=12

<12

1,2,3...

== 8K pages

35,50
35,...
50,...

can index 8K pages
of the next level
Writes

1,2,3…  12,15,17  20,…  35,…  50,…  35,50
writes

1, 2, 3, ...
12, 15, 17
20, ...
...
2, 5, 6, 3, 2
22, 25, 24
7, 8, 3, 5, 4
Workload skew may generate long linked lists… Queries arriving on different leaves will have different performance.

Not robust

No need to change the internal part of the tree (maybe only periodically)

No need to lock the internal part of the tree for concurrent access
b-tree: dynamic tree -> always balanced
b-tree: dynamic tree -> always balanced

all similar queries cost ~the same = robust
**b-tree:** *dynamic tree -> always balanced*

all similar queries cost ~the same = robust

we need to maintain the structure, not only the data
**b-tree:** dynamic tree -> always balanced

- all similar queries cost ~the same = robust
- we need to maintain the structure, not only the data
- every node/page is at least 50% full (except root)
initial state

leaf

new value

after insert
initial state

leaves

node capacity = 4

after insert

node capacity = 4
Redistribution: Utilize free space elsewhere -> Affects 3 nodes.

leaf nodes

- Initial state:
  - Node 1: 3 5 7 9
  - Node 2: 12 16

- After insert (node 6):
  - Node 1: 3 5 6 7
  - Node 2: 9 12 16

- Redistribution: Utilize free space elsewhere affects 3 nodes.

Heights:

- Node 1: 3
- Node 2: 3

Node capacities:

- Node 1: 4
- Node 2: 4

Node capacity = 4
initial state

leaves

node capacity = 4

after insert

initial state

after insert

node capacity = 4

leaves
Split: Node overflows and new node is generated
Affects three nodes

node capacity = 4
insert "6"
node capacity = 4
delete "5"
node capacity = 4
No merges when < 50% full
Just delete when empty

Extra: T. Johnson and D. Shasha
Utilization of B-trees with Inserts,
Deletes and Modifies.
In Proceedings of the ACM Symposium
on Principles of Database Systems
(PODS), 1989.
two kinds of updates: data and structure: locks vs latches
once data is “in”, maybe the structure does not matter temporarily
complete split in two separate steps
only one lock at a time
complete split in two separate steps only one lock at a time

Step 1

...
complete split in two separate steps
only one lock at a time

Step 1

blink
complete split in two separate steps
only one lock at a time

Step 1

step

blink

Step 2

blink
complete split in two separate steps
only one lock at a time

Step 1

Step 2

Extra: Efficient Locking for Concurrent Operations on B-Trees
Philip L Lehman and S Bing Yao
ACM Transactions on Database Systems (TODS), 1981
searching internal node

(v1,p1) (v2,p2) (v3,p3) (v4,p4) (v5,p5) (v6,p6)…
It can be anything; any layout

searching internal node

(v1,p1) (v2,p2) (v3,p3) (v4,p4) (v5,p5) (v6,p6)…
how big should nodes be
the more data we can fit in a node (without changing its size) the faster our index becomes

any ideas how to achieve this
what does a leaf contain
pointers or positions/ids
buffers for loading

any problems
how to load a b-tree

for(i=0; i<totalValues;i++)
    insertToBtree(tree,value[i])

?
bulk loading
1. sort all
2. build tree

1, 2
3, 4
5, 6
7, 8
9, 10
bulk loading
1. sort all
2. build tree
bulk loading
1. sort all
2. build tree

1,2
3,4
5,6

3.5

7,8
9,10
bulk loading
1. sort all
2. build tree
no perfect tree…

it depends on what we are trying to do
read/write ratio, data distribution,
response time guarantees
no perfect tree…
it depends on what we are trying to do
read/write ratio, data distribution,
response time guarantees

If you are thinking about research:
How can we automate data structure design?
(and coding!)
project milestone 3

reasonably cache conscious b-tree
no redistribution, only splits
no merges, only merge on empty
no parent pointers,
you can keep a simple access queue

no prev pointers at leaves
no prev/next pointers at internal nodes

test variations: e.g., how to scan a leaf,
or leaf layout, node size

both primary and secondary index
Employee(\text{Id}, \text{name}, \text{address}, \text{office}, \text{salary}, \text{year hired}, \ldots)

We have a B-tree on table Employee which uses salary as the key and also contains attributes “name” and “year hired”.

We want to give a 5% raise to all employees that work for more than 10 years in the company and have a salary lower than 100K.

1) Write the SQL query  
2) How to update the B-tree?  
3) What is the query plan?
We want to give a 5% raise to all employees that work for more than 10 years in the company and have a salary lower than 100K.

update employee
set salary=salary*1.05
where salary<100K and year_hired<2007
b-tree on employee.salary

X 100K
b-tree on employee.salary

for each tuple if it qualifies (check years hired) update

$X$  $100K$
b-tree on employee.salary

for each tuple if it qualifies (check years hired) update

100K
b-tree on employee.salary

For each tuple if it qualifies (check years hired) update we are going to keep updating until everyone is at least at 100K
b-tree on `employee.salary`

**just go backwards?**

for each tuple
if it qualifies (check years hired)
update

100K

we are going to keep updating until everyone is at least at 100K
b-tree on
employee.salary

for each tuple
if it qualifies (check years hired)
update

we are going to keep
updating until everyone
is at least at 100K
b-tree on employee.salary

get all qualifying IDs first
then update in one go
or
maintain an extra structure
e.g., a bit vector
or hash on tuple ID
to remember
the updated tuples

for each tuple
if it qualifies (check years hired)
update

we are going to keep updating until everyone
is at least at 100K
Read: **Modern B-Tree Techniques**
by Goetz Graefe
Foundations and Trends in Databases, 2011
b-trees 2.0

DATA SYSTEMS

prof. Stratos Idreos

HARVARD
School of Engineering
and Applied Sciences