
CC in B-trees

Christos Faloutsos

www.cs.cmu.edu/~christos

Roadmap

1) Roots: System R and Ingres
2) Implementation: buffering, indexing, q-opt
3) Transactions: locking, recovery
   granularity of locks
   optimistic CC
   recovery

B-trees
4) Distributed DBMSs
5) Parallel DBMSs: Gamma, Alphasort
   --

References


(for B-trees: e.g., see Knuth, Volume III)

Detailed overview

• B-trees - reminders
  – definition
  – search
  – insertion
  – (deletion)
  – variations: B+trees, B*trees
• CC on B-trees - B-link-trees

B-trees

• the most successful family of index schemes (B-trees, B+trees, B*-trees)
• Can be used for primary/secondary, clustering/non-clustering index.
• balanced “n-way” search trees
B-tree properties:

- each node, in a B-tree of order n:
  - Key order
  - at most $n$ pointers
  - at least $n/2$ pointers (except root)
  - all leaves at the same level
  - if number of pointers is $k$, then node has exactly $k-1$ keys
  - (leaves are empty)

Properties

- “block aware” nodes: each node -> disk page
- $O(\log (N))$ for everything! (ins/del/search)
- typically, if $m = 50 - 100$, then 2 - 3 levels
- utilization $\geq 50\%$, guaranteed; on average $69\%$

Queries

- Algo for exact match query? (eg., ssn=8?)

---

Queries

- Algo for exact match query? (eg., ssn=8?)

---

Queries

- Algo for exact match query? (eg., ssn=8?)

---

Queries

- Algo for exact match query? (eg., ssn=8?)
Queries

• Algo for exact match query? (eg., ssn=8?)

> Queries

• what about range queries? (eg., 5<salary<8)
• Proximity/ nearest neighbor searches? (eg., salary ~ 8 )

B-trees: Insertion

• Insert in leaf; on overflow, push middle up (recursively)
• split: preserves B - tree properties

• what about range queries? (eg., 5<salary<8)
• Proximity/ nearest neighbor searches? (eg., salary ~ 8 )

B-trees

Easy case: Tree T0; insert ‘8’
Hardest case: Tree T0; insert ‘8’

```
<6  6 9  >6
  1 3 7 8 13

push middle up
```

Hardest case: Tree T0; insert ‘2’

```
<6  6 9  >6
  1 3 7 8 13

2  push middle up
```

```
6 9
1 2 3 7 13

Final state
```

```
6 9 2
1 3 7 13

2
```

Hardest case: Tree T0; insert ‘2’

```
<6  6 9  >6
  1 3 7 8 13

Ovf: push middle
```

```
2 6 9
1 3 7 13
```

```
6 9
1 3 7 13

Final state
```

```
6
2 9
1 3 7 13
```

B-trees - insertion

- Q: What if there are two middles? (eg, order 4)
- A: either one is fine
**Pseudo-code**

```plaintext
INSERTION OF KEY 'K'
find the correct leaf node 'L';
if ('L' overflows)
  split 'L', by pushing the middle key upstairs to parent node 'P';
  if ('P' overflows)
    repeat the split recursively;
  else{
    add the key 'K' in node 'L'; /* maintaining the key order in 'L' */
  }
else {

```

**Detailed overview**

- B-trees - reminders
  - definition
  - search
  - insertion
  - (deletion)
  - variations: B+trees, B*trees
- CC on B-trees - B-link-trees

**Deletion**

Rough outline of algo:
- Delete key;
- on underflow, may need to merge

In practice, some implementors just allow underflows to happen…

**B-trees: Insertion**

- Insert in leaf; on overflow, push middle up (recursively – 'propagate split')
- split: preserves all B-tree properties (!!!)
- notice how it grows: height increases when root overflows & splits
- Automatic, incremental re-organization (contrast with ISAM!)

**B-trees – Deletion**

Easiest case: Tree T0; delete ‘3’

```
<6  
   
| 1 | 3 |
---|---|
| 1 | 7 |
| 7 | 13 |
```

```
>6  
  
| 6 | 9 |
---|---|
| 6 | 9 |
| 9 |
```

Easiest case: Tree T0; delete ‘3’

```
<6  
   
| 1 | 3 |
---|---|
| 1 | 7 |
| 7 | 13 |
```

```
>6  
  
| 6 | 9 |
---|---|
| 6 | 9 |
| 9 |
```
B-trees – Deletion

- Case 1: delete a key at a leaf – no underflow
- Case 2: delete non-leaf key – no underflow
- Case 3: delete leaf-key; underflow, and ‘rich sibling’
- Case 4: delete leaf-key; underflow, and ‘poor sibling’

B-trees – Deletion

- Case 2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

B-trees – Deletion

- Case 2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

B-trees – Deletion

- Case 2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

B-trees – Deletion

- Case 2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

B-trees – Deletion

- Case 2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)
**B-trees – Deletion**

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)
- Q: How to promote?
  - A: pick the largest key from the left sub-tree (or the smallest from the right sub-tree)
- Observation: every deletion eventually becomes a deletion of a leaf key

**B-trees – Deletion**

- Case2: delete a key at a non-leaf – no underflow (eg., delete 6 from T0)

**B-trees – Deletion**

- Case1: delete a key at a leaf – no underflow
- Case2: delete non-leaf key – no underflow
  - Case3: delete leaf-key; underflow, and ‘rich sibling’
  - Case4: delete leaf-key; underflow, and ‘poor sibling’

**B-trees – Deletion**

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)

**B-trees – Deletion**

- Case3: underflow & ‘rich sibling’ (eg., delete 7 from T0)
  - ‘rich’ = can give a key, without underflowing
  - ‘borrowing’ a key: THROUGH the PARENT!

**B-trees – Deletion**

- Case3: underflow & ‘rich sibling’
### B-trees – Deletion

- **Case 3:** underflow & ‘rich sibling’ (eg., delete 7 from T0)
  
  ![Diagram of Case 3](image1)

- **Case 2:** delete non-leaf key – no underflow
  
  ![Diagram of Case 2](image2)

- **Case 1:** delete a key at a leaf – no underflow
  
  ![Diagram of Case 1](image3)

- **Case 4:** underflow & ‘poor sibling’ (eg., delete 13 from T0)
  
  ![Diagram of Case 4](image4)

- **Case 4 (continued):** underflow & ‘poor sibling’ (eg., delete 13 from T0)
  
  ![Diagram of Case 4 continued](image5)
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```

B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
  - Merge, by pulling a key from the parent
  - Exact reversal from insertion: ‘split and push up’, vs. ‘merge and pull down’
  - I.e.:

```
+---<6---+
|   1   |
|   3   |
+---6-9---+
|   1   |
|   3   |
+---<6---+
```
B-tree deletion - pseudocode

DELETION OF KEY ‘K’
locate key ‘K’, in node ‘N’
if (‘N’ is a non-leaf node) {
  delete ‘K’ from ‘N’;
  find the immediately largest key ‘K1’;
  /* which is guaranteed to be on a leaf node ‘L’ */
  copy ‘K1’ in the old position of ‘K’;
  invoke this DELETION routine on ‘K1’ from the leaf node ‘L’;
} else {
  /* ‘N’ is a leaf node */
  ... (next slide...) 
}

In practice:

- no empty leaves;
- no gaps in the tree;
- no gaps in the tree.

B-trees in practice

In practice:

- no empty leaves;
- no gaps in the tree.

### Theory vs. Practice

**Theory**

- B-trees maintain a balanced structure.
- Keys are stored in sorted order.
- Each node has a minimum and maximum number of children.

**Practice**

- B-trees are not always perfectly balanced.
- Keys may not always be in sorted order.
- Nodes may have fewer or more children than specified by the minimum and maximum values.

### Format Differences

- **Leaf Nodes**
  - B-trees in theory: Store data and pointers to children.
  - B-trees in practice: Store data and pointers to children.

- **Non-Leaf Nodes**
  - B-trees in theory: Store data and pointers to children.
  - B-trees in practice: Store data and pointers to children.

### Conclusion

- B-trees are versatile data structures that can be adapted to different environments.
- In practice, B-trees are used in a variety of applications, including databases and file systems.
Detailed overview

- B-trees - reminders
  - definition
  - search
  - insertion
  - (deletion)
  - variations: B+trees, B*trees
- CC on B-trees - B-link-trees

B+ trees - Motivation

B-tree needs back-tracking – how to avoid it?

Solution: B+ - trees

- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

B+ trees

B+ tree insertion

INSERTION OF KEY 'K'
insert search-key value to 'L' such that the keys are in order;
if ( 'L' overflows) { 
  split 'L' ;
  insert (ie., COPY) smallest search-key value 
of new node to parent node 'P';
  if ( 'P' overflows ) { 
    repeat the B-tree split procedure recursively;
    /* Notice: the B-TREE split; NOT the B+ -tree */
  }
}
B+-tree insertion – cont’d

/* ATTENTION:
   a split at the LEAF level is handled by
   COPYING the middle key upstairs;
   A split at a higher level is handled by PUSHING
   the middle key upstairs
*/

B+ trees - insertion

Eg., insert ‘8’

COPY middle upstairs

B+ trees - insertion

Eg., insert ‘8’

COPY middle upstairs

B+ trees - insertion

Eg., insert ‘8’

COPY middle upstairs

B+ trees - insertion

Eg., insert ‘8’

Non-leaf overflow – just PUSH the middle

COPY middle upstairs
B+ trees - insertion

Eg., insert ‘8’

>6

>9

B*-tree

- In B-trees, worst case util. = 50%, if we have just split all the pages
- how to increase the utilization of B - trees?
- ..with B* - trees!

B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!

B* - trees: deferred split!

- Notice: shorter, more packed, faster tree
- It’s a rare case, where space utilization and speed improve together
- BUT: What if the sibling has no room for our ‘lending’?
B* - trees: deferred split!

• BUT: What if the sibling has no room for our ‘lending’?
• A: 2-to-3 split: get the keys from the sibling, pool them with ours (and a key from the parent), and split in 3.
• Details: too messy (and even worse for deletion)

B* - trees - Conclusions

• all B - tree variants can be used for any type of index: primary/secondary, sparse (clustering), or dense (non-clustering)
• All have excellent, \( O(\log N) \) worst-case performance for ins/del/search
• It’s the prevailing indexing method

Detailed overview

• B-trees - reminders
• CC on B-trees - B-link-trees
  - problem and past methods
  - main idea
  - algorithms and correctness proofs

Lehman and Yao – CC on B-trees

• “safe” node: node with <2k entries
• “unsafe” node: node with =2k entries
• Simple CC won’t do. Why?

Example

Transaction 1:
read x;
look for 15;
get ptr to y;

ERROR!!!

Transaction 2:
read x; read y;
insert 9
split y into y+y'

Example

Transaction 1:
read x;

Transaction 2:
read x;
look for 15;
get ptr to y;
read y;
insert 9
split y into y+y'
Previous B-tree CC algorithms

• What would you do?
• Can you find an algo, where “readers” need NO LOCKS? (!)

... unsafe node

Previous B-tree CC algorithms

• S-locks and X-locks? (~[Samadi, ‘76])
  – find highest affected node & X-lock it
  – but: writers need to (temporarily) X-lock root!

Previous B-tree CC algorithms

• ‘intent’ locks?

Previous B-tree CC algorithms

• actually, S-locks, X-locks, and ‘warning-locks’ (‘writer-X-locks’)
Previous B-tree CC algorithms

- actually, S-locks, X-locks, and ‘warning-locks’

... and get w.l. here

leaf is safe - get X-lock for leaf and release w.l.
Detailed overview

- B-trees - reminders
- CC on B-trees - B-link-trees
  - problem and past methods
  - main idea
  - algorithms and correctness proofs

B\text{link}-tree

- Slightly MODIFY the B-tree structure
- Readers can ALWAYS proceed (!!)
- Use some redundancy, so that readers can always reach a valid B-tree
- Specifically: add right-link pointer on every node, pointing to the next node on the same level (rightmost nodes: null pointers)

B\text{link}-tree

- right-link pointer, so that the split for ‘a’ goes like:

\[
\begin{array}{c}
\text{a} \\
\text{f} \\
\text{b'} \\
\text{e}
\end{array}
\]

B\text{link}-tree

- right-link pointer, so that the split for ‘a’ goes like:

\[
\begin{array}{c}
\text{a} \\
\text{f'} \\
\text{b'} \\
\text{e}
\end{array}
\]

B\text{link}-tree

- right-link pointer, so that the split for ‘a’ goes like:

\[
\begin{array}{c}
\text{a'} \\
\text{f} \\
\text{b'} \\
\text{e}
\end{array}
\]

(\text{#})

B\text{link}-tree

- right-link pointer, so that the split for ‘a’ goes like:

\[
\begin{array}{c}
\text{a'} \\
\text{f} \\
\text{b'} \\
\text{e}
\end{array}
\]
**B^link-tree**

- **right-link** pointer, so that the split for ‘a’ goes like:

  ![Diagram](image)

  Notice: at ANY point, a reader sees a valid B(link) tree!

  (Even in state (*), if we are careful!)

**Details:**

- #1) **legal** to have “left twin” and no parent (when will this happen?)

**B^link-tree**

**Details:**

- #2) other change in the B+ tree structure?

**B^link-tree**

- **A**: highest Key value (in the sub-tree)

**Q**: Why?

**B^link-tree**

- **A**: highest Key value

**Q**: Why?

- **A**: so that, on search, we can detect case (*) and fetch the right twin

**A**: highest Key value

**Q**: Why?

- **A**: so that, on search, we can detect case (*) and fetch the right twin

  ![Diagram](image)
Advantages

- Allows for “temporary fix” until all pointers are added correctly
- Link pointers would be used infrequently – because splitting a node is a “special case”
- “Level traversal” comes for free as a side effect

Detailed overview

- B-trees - reminders
- CC on B-trees - B-link-trees
  - problem and past methods
  - main idea
    - algorithms and correctness proofs

Search

- No locks needed for reads
- Just move right as well as down

Search

- No locks needed for reads
- Just move right as well as down

Search

- case (*) - eg., look for ‘39’

Search

- case (*) - eg., look for ‘39’
Search

pseudo code:
– descend to leaf level (visiting right-twins, whenever case (*) appears)
– keep moving right, if necessary
more details: in book.
\[ x = \text{scannode}(v, A) : \text{scans page } A, \text{for value } v \]
and returns the correct pointer (link or not)

Insertion

• Well-ordered locks
• locks are requested bottom-up
• needs at most 3 locks at a time (and, usually, 1 or 2)
• Use stack to remember ancestors
• Split while preserving links
in more detail:

Insertion (cont’d)

else // ‘A’ is not safe - will split
get extra page B
re-distribute contents of ‘A’ into ‘newA’ and ‘B’
write(B); then write(newA)
// insert correct value to parent node ‘F’
// may need to scan right-links, if ‘F’ has split in the meanwhile!
lock(F); unlock(A)
split ‘F’ recursively, if needed

Deletions

• ignore underflows; no merging
• reorganize, if too many nodes underflow

Detailed overview

• B-trees - reminders
• CC on B-trees - B-link-trees
  – problem and past methods
  – main idea
  – algorithms
  – correctness proofs
Proofs - observations

- THM: deadlocks are impossible
  - because locks are ordered - top to bottom; left to right
- Lemma: readers are always handled correctly (thanks to the right-links!)

Conclusions

- B+ trees: **very** popular
- B-link trees allow high concurrency
  - 3 locks at most;
  - no locks needed for readers
  - no deadlocks!