CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #18: Indexes
Note

- Most material NOT in the textbook!
Indexes

- **Index** = data structure used to speed access to tuples of a relation, given values of one or more attributes.
Declaring Indexes

- No standard!
- Typical syntax:

CREATE INDEX StudentsInd ON Students(ID);

CREATE INDEX CoursesInd ON Courses(Number, DeptName);
Types of Indexes

- **Primary**: index on a key
  - Used to enforce constraints

- **Secondary**: index on non-key attribute

- **Clustering**: order of the rows in the data pages correspond to the order of the rows in the index
  - Only one clustered index can exist in a given table
  - Useful for range predicates

- **Non-clustering**: physical order not the same as index order
Using Indexes (1): Equality Searches

- Given a value \( v \), the index takes us to only those tuples that have \( v \) in the attribute(s) of the index.

- E.g. (use CourseInd index)

```sql
SELECT Enrollment FROM Courses
WHERE Number = "4604" and DeptName = "CS"
```
Using Indexes (1): Equality Searches

- Given a value $v$, the index takes us to only those tuples that have $v$ in the attribute(s) of the index.

- Can use Hashes, but see next
Using Indexes (2): Range Searches

- ``Find all students with gpa > 3.0’’
- may be slow, even on sorted file
- Hashes not a good idea!
- What to do?
Range Searches

- ``Find all students with gpa > 3.0' '``
- may be slow, even on sorted file
- Solution: Create an `index` file.
Range Searches

- More details:
- if index file is small, do binary search there
- Otherwise??

![Diagram of data and index files with page references](image)
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-trees

- Eg., B-tree of order $d=1$:
B - tree properties:

- each node, in a B-tree of order d:
  - Key order
  - at most n=2d keys
  - at least d keys (except root, which may have just 1 key)
  - 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 is a disk page
- $O(\log (N))$ for everything! (ins/del/search)
- Typically, if $d = 50 - 100$, then 2 - 3 levels
- Utilization $\geq 50\%$, guaranteed; on average 69%
Queries

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

- http://slady.net/java/bt/
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?)

*H steps (= disk accesses)*
Queries

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

- what about range queries? (e.g., $5 < \text{salary} < 8$)
- Proximity/ nearest neighbor searches? (e.g., salary $\sim 8$)
Queries

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

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

- Easy case: Tree T0; insert ‘8’
B-trees

- Tree $T_0$; insert ‘8’
B-trees

- Hardest case: Tree T0; insert ‘2’
B-trees

- Hardest case: Tree T0; insert ‘2’

push middle up
B-trees

- Hardest case: Tree T0; insert ‘2’

Ovf; push middle
B-trees

- Hardest case: Tree T0; insert ‘2’
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
Pseudo-code

INSERTION OF KEY ’ K’

find the correct leaf node ’ L’ ;

if ( ’ L’ overflows ) {
    split ’ L’ , and push middle key 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’ */
}
Deletion

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

- In practice, some implementors just allow underflows to happen...
B-trees – Deletion

- Easiest case: Tree T0; delete ‘3’
B-trees – Deletion

- Easiest case: Tree T0; delete ‘3’
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

- Case 1: delete a key at a leaf – no underflow (delete 3 from T0)
B-trees – Deletion

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

Delete & promote, ie:
B-trees – Deletion

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

![Diagram of B-tree deletion](image)

Delete & promote, ie:
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 (e.g., delete 6 from T0)

![Diagram of B-tree deletion](image)

**FINAL TREE**

- Node at non-leaf level `3` is deleted, leaving `1`, `7`, and `13` nodes with `3`, `9`, and `>9` values, respectively.

---

Prakash 2013

VT CS 4604
B-trees – Deletion

- Case 2: 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

- Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

Delete & borrow, ie:
B-trees – Deletion

- Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

Delete & borrow, ie:
B-trees – Deletion

- Case 3: underflow & ‘rich sibling’
  - ‘rich’ = can give a key, without underflowing
  - ‘borrowing’ a key: THROUGH the PARENT!
B-trees – Deletion

- **Case 3**: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

![Diagram showing deletion process in B-trees]

Delete & borrow, ie:
B-trees – Deletion

- Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)
B-trees – Deletion

- **Case 3:** underflow & ‘rich sibling’ (e.g., delete 7 from T0)

![Diagram showing deletion process in B-tree]

Delete & borrow, i.e.:
B-trees – Deletion

- Case 3: underflow & ‘rich sibling’ (e.g., delete 7 from T0)

![Diagram of B-tree deletion process](image)
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)
**B-trees – Deletion**

- **Case 4:** underflow & ‘poor sibling’ (e.g., delete 13 from T0)
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (e.g., delete 13 from T0)
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (e.g., 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.:
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (eg., delete 13 from T0)

Diagram:
- Node 6
- Node 7, 9
- Merge with ‘poor’ sibling
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’ (e.g., delete 13 from T0)
B-trees – Deletion

- Case 4: underflow & ‘poor sibling’
- -> ‘pull key from parent, and merge’
- Q: What if the parent underflows?
- A: repeat recursively
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..)
B-tree deletion - pseudocode

/* ’N’ is a leaf node */
if( ’N’ underflows ){
    let ’N1’ be the sibling of ’N’;
    if( ’N1’ is "rich"){
        /* ie., N1 can lend us a key */
        borrow a key from ’N1’ THROUGH the parent node;
    }else{
        /* N1 is 1 key away from underflowing */
        MERGE: pull the key from the parent ’P’,
            and merge it with the keys of ’N’ and ’N1’ into a
            new node;
        if( ’P’ underflows){ repeat recursively }
    }
}
Variations

- How could we do even better than the B-trees above?
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 and B*-trees

- Eg., Tree T0; insert ‘2’
B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!
  (through PARENT, of course!)
B*-trees: deferred split!

- Instead of splitting, LEND keys to sibling!
  (through PARENT, of course!)

![Diagram of B*-tree principles]
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!

- 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.
- Could we extend the idea to 3-to-4 split, 4-to-5 etc?
B*-trees: deferred split!

- **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.
- Could we extend the idea to 3-to-4 split, 4-to-5 etc?
- Yes, but: diminishing returns
B+ trees - Motivation

- B-tree – print keys in sorted order:
B+ trees - Motivation

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

- Stronger reason: for clustering index, data records are scattered:

```
<6
1 3
>6
6 9
<9
7
>9
13
```
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
- (vital, for clustering index!)
B+ trees
B+ trees
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

Based on the search for 15*, we know it is not in the tree!
B+ Trees in Practice

- Most popular index structure
- Excellent, $O(\log N)$ worst-case performance for ins/del/search; (~3-4 disk accesses in practice)
- guaranteed 50% space utilization; avg 69%
- Ins/Del algorithms a bit tricky---we won’t cover them here
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = $2 \times 100 \times 0.67 = 134$

- Typical capacities:
  - Height 4: $1334 = 312,900,721$ entries
  - Height 3: $1333 = 2,406,104$ entries
B+ Trees in Practice

- Can often keep top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB
A major problem in making a database run fast is deciding which indexes to create.

**Pro:** An index speeds up queries that can use it.

**Con:** An index slows down all modifications on its relation because the index must be modified too.
Example: Tuning

- Suppose the only things we did with our courses database was:
  1. Insert new facts into a relation (20%).
  2. Find the enrollment of a given course (80%).
- Then CourseInd on Courses(Num, DeptName) is good, but StudentInd on Student(Id) would be harmful.
**Tuning Advisors**

- A major research area for years
  - Because hand tuning is so hard.
- An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload.
Tuning Advisors --- (2)

- The advisor generates candidate indexes and evaluates each on the workload.
  - Feed each sample query to the query optimizer, which assumes only this one index is available.
  - Measure the improvement/degradation in the average running time of the queries.