Introduction to Data Management

Lecture 15
(More About Indexing)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- HW’s and quizzes:
  - HW #6 due **TOMORROW** (at 6 pm)
  - We’re now back in a “Friday HW groove”
  - Beware of (real!) late penalties again
- This week’s meetings
  - Discussion & quiz (were) as usual
- Today’s lecture plan:
  - Deeper dive into B+ tree indexes
  - Overview of (static) hash-based indexes
  - (More on indexes in CS122C if desired 😊)
**B+ Tree: Most Widely Used Index!**

- Insert/delete at log \( F \cdot N \) cost; keep tree *height-balanced*. (\( F = \) fanout, \( N = \) # leaf pages)
- Minimum 50% occupancy (except for root).
  - Each node contains \( d \leq m \leq 2d \) entries.
  - The (mythical) \( d \) is called the *order* of the B+ tree.
- Supports equality and range-searches efficiently.

**Example B+ Tree**

- Search begins at root, and key comparisons direct the search to a leaf (as in ISAM).
- *Ex:* Search for 5*, 15*, all data entries \( \geq 24* \), ...

*Based on the search for 15*, we know it is not in the tree!*
Example B+ Tree (a clarification)

- Search begins at root, and key comparisons direct the search to a leaf (as in ISAM).
- Ex: Search for 5*, 15*, all data entries >= 24*, ...

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

Inserting a Data Entry into a B+ Tree

- Find correct leaf L (by searching for the new k).
- Put new data entry (k*, a.k.a. (k, I(k)) in leaf L.
  - If L has enough space, done! (Most likely case!)
  - Else, must split L (into L and a new node L2)
    - Redistribute entries evenly and copy up middle key.
    - Insert new index entry pointing to L2 into parent of L.
- This can happen recursively.
  - To split an index node, redistribute entries evenly but push up the middle key. (Contrast with leaf splits!)
- Splits “grow” tree; root split increases its height.
  - Tree growth: gets wider or one level taller at top.
Inserting $8^*$ into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

- Note difference between copy-up and push-up; be sure you understand the reasons for this!

![Diagram of Example B+ Tree After Inserting $8^*$]

Example B+ Tree After Inserting $8^*$

- Notice that root was split, leading to increase in height.

- In this example, could avoid split by redistributing entries; however, not usually done in practice. (Q: Why is that?)
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is still at least half-full, done!
  - If $L$ has only $d-1$ entries,
    - Try to redistribute, borrowing from sibling (adjacent
      node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling.
- If merge occurred, must delete search-guiding entry (pointing to $L$ or sibling) from parent of $L$.
- Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

- Deleting 19* is easy.
- Deleting 20* is done with redistribution. Notice how middle key is copied up.
... And Then Deleting 24*

- Must merge.
- Observe "toss" of index entry (on right), and "pull down" of index entry (below).

Example of Non-leaf Redistribution

- New/different example B+ tree is shown below during deletion of 24*
- In contrast to previous example, can redistribute entry from left child of root to right child.
After Redistribution

- Intuitively, entries are redistributed by “pushing through” the splitting entry in the parent node.

Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- Bulk Loading can be done much more efficiently!
- Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.
Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (A split may go up the right-most path to the root.)

- Much faster than repeated inserts!

A Note on B+ Tree “Order”

- (Mythical) order \((d)\) concept replaced by physical space criterion in practice (“at least half-full”).
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean that different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries.
(Page Implementation Details)

Q: What if you were to “open up” a B+ Tree page?

- Control info (e.g., level, # children, free space offset)
- Search key array (with possible on-page indirection for variable-length data, using offsets), or key/data array – for non-leaf vs. leaf pages, respectively
- Child pointer array, where pointer = page id on disk!

A Brief Aside: Hash-Based Indexes

- As for any index, 3 alternatives for data entries \( k^* \):
  - Data record with key value \( k \)
  - \( <k, \text{rid of data record with search key value } k \rangle \)
  - \( <k, \text{list of rids of data records with search key } k \rangle \)
  - Choice is orthogonal to the indexing technique!
- **Hash-based** indexes are fast for equality selections. *Cannot* support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.
**Static Hashed Indexes**

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- \( h(k) \mod N \) = bucket (page) to which data entry with key \( k \) belongs. \( (N = \# \text{ of buckets}) \)

\[
\begin{array}{c|c|c}
\hline
\text{h}(k) \mod N & 0 & 25*, \ --, -- \\
\hline
1 & 0*, 24*, \ -- & \text{Ex: Find data with key=25} \rightarrow 25* \\\n2 & 9*, 17*, 81* & \text{--, --} \\\n\hline
\end{array}
\]

(Ex: Using \( N = 8 \) and \( h(k) = k \) (a bad \( h(k) \)...!)

\(* = \text{overflow page} \)

**Static Hashed Indexes (Cont’d.)**

- Buckets contain data entries (like for ISAM or B+ trees) – very similar to what we just looked at.
- Hash function works on search key field of record \( r \). Must distribute values over range 0 ... M-1.
  - \( h(key) = (a \ast key + b) \) usually works fairly well.
  - \( a \) and \( b \) are constants; lots known about how to tune \( h \).
- Long overflow chains can develop and degrade performance.
  - Extendible and Linear Hashing: Dynamic techniques to fix this problem. (Take CS122c!)

\[
\begin{array}{c|c|c}
\hline
\text{primary bucket pages} & 0*, 24*, \ -- & 25*, \ --, -- \\
\hline
\text{overflow pages} & \text{9*, 17*, 81*} & \text{--, --} \\
\hline
\end{array}
\]
Indexing Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure. (Prehistoric B+ Tree!)
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; log \( f \) \( N \) cost.
  - High fanout (\( F \)) means depth rarely more than 3 or 4.
  - If K*’s are data records, use keys (vs. RIDs) elsewhere!

Indexing Summary (Cont’d.)

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems, and outside DBMSs as well, because of its versatility. Also the most optimized (e.g., for loading, locking, recovery, incremental loads, ...).
- Other database indexes to be aware of:
  - Hash-based (for exact match queries).
  - R tree (for spatial indexing and queries).
  - Inverted keyword (for text indexing and queries).