Today’s Lecture

• B+ Trees. Section 10.3-10.6, 10.8
• Static and Extendible Hashing. Section 11.1 and 11.2

Problem with ISAM

• Inserts and deletes affect only the leaf pages. As a consequence, long overflow chains at the leaf pages may result
• Long overflow chains may significantly slow down the time to retrieve a record (all overflow pages have to be searched)
• Overflow pages are not sorted. Therefore, lookup is slow. Possible to keep overflow pages sorted also but inserts will be slow
• Overflow pages do not go away unless all records in the overflow pages are deleted, or a complete reorganization is performed
B+ Tree: The Most Widely Used Index

• A dynamic index structure that adjusts gracefully to inserts and deletes
• A balanced tree
• Leaf pages are not allocated sequentially. They are linked together through pointers (a doubly linked list).

Main characteristics:
– Insert/delete at $\log_F N$ cost; keep tree height-balanced. ($F = \text{fanout}, N = \# \text{leaf pages}$)
– Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The parameter $d$ is called the order of the tree.
– Supports equality and range-searches efficiently.
Format of a node

- Same as that of ISAM
- Non-leaf nodes with m index entries contain m+1 pointers to children
- Pointer $P_i$ points to a child with key values $k$ such that $k_i \leq k < k_{i+1}$
- $P_0$ points to a child whose key values are less than $k_1$

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf. At each node, a binary search or linear search can be performed
- Search for 5*, 15*, all data entries $\geq 24^*$ ...

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 $L$.
- Put data entry onto $L$.
  - If $L$ has enough space, done!
  - Else, must split $L$ (into $L$ and a new node $L_2$)
    - Distribute entries evenly, copy up middle key.
    - Insert index entry pointing to $L_2$ into parent of $L$.
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases 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.

Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)
• Notice that root was split, leading to increase in height.
• In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

• Notice that the value 5 occurs redundantly, once in a leaf page and once in a non-leaf page. This is because values in the leaf page cannot be pushed up, unlike the value 17.
Redistribution with sibling nodes

- If a leaf node where insertion is to occur is full, fetch a neighbour node (left or right).
- If neighbour node has space and same parent as full node, redistribute entries and adjust parent nodes accordingly.
- Otherwise, if neighbour nodes are full or have a different parent (i.e., not a sibling), then split as before.
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf $L$ where entry belongs.
- Remove the entry.
  - If $L$ is at least half-full, done!
  - If $L$ has only $d-1$ entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as $L$).
    - If re-distribution fails, merge $L$ and sibling.
  - If merge occurred, must delete 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 re-distribution. 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).

Delete 24*

Root
Example of Non-leaf Re-distribution

• Tree is shown below during deletion of 24*. (What could be a possible initial tree?)
• In contrast to previous example, can re-distribute entry from left child of root to right child.

```
Root
5 13 17 20
2 3 5 7 8 14 16 17 18 20 21
2 2 2 2 2 2 2 2 2 2

Root
17
5 13
2 3 5 7 8 14 16
2 2 2 2 2 2 2
2 2 2 2 2 2 2
```

After Re-distribution

• Intuitively, entries are re-distributed by `pushing through' the splitting entry in the parent node.
• It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.
**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** for creating a B+ tree index on existing records is much more efficient.

---

**Bulk Loading**

- Sort all data entries
  - If data entries are (key, pointer) pairs, sort these pairs according to key values and not the actual data records
- Allocate an empty page to be the root. Insert pointer to first (leaf) page in root page
Bulk Loading

- Add entry into root page for each page of sorted data entries. Doubly linked data entry pages. Proceed until root page is filled or no more data entry pages

- If root page is filled and to insert one more page of data entries, split the root and create a new root page
Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

Much faster than repeated inserts, especially when one considers locking!

**Bulk Loading**

| Winter 2003 |

---

**Summary of Bulk Loading**

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.

- Option 2: Bulk Loading
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.
Summary

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an index is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to indexing technique used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.
Summary

• Tree-structured indexes are ideal for range-searches, also good for equality searches.
• ISAM is a static structure.
  – 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; \( \log_2 N \) cost.
  – High fanout (\( F \)) means depth rarely more than 3 or 4.
  – Almost always better than maintaining a sorted file.

Hash-Based Indexing

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

- \( h(k) \mod N \) returns the bucket to which data entry with key \( k \) belongs. \( (N = \# \text{ of buckets}) \)
- We refer to the above as the hash function which maps values into a range of buckets

```
\begin{align*}
\text{Primary bucket pages} & \quad \text{Overflow pages} \\
0 & \quad \ldots \\
2 & \quad \ldots \\
\text{h(key) mod N} & \quad \ldots \\
\text{key} & \\
\text{h} & \\
N-1 & \\
\ldots & \\
\end{align*}
```

- # primary pages fixed (which is N), allocated sequentially, never de-allocated; overflow pages if needed.
- Buckets contain data entries, which can be in any of the three alternatives discussed earlier
- Hash function works on search key field of record \( r \). Must distribute values over range 0 ... N-1.
  - Typically, \( h(key) = (a \cdot key + b) \) for some constants \( a \) and \( b \)
  - Lots known about how to tune \( h \).
Static Hashing

- **Search for data entry k**: Apply hash function $h$ on $k$ to obtain the bucket number. Then, search the bucket for $k$.
  - Data entries in each bucket are typically maintained in sorted order to speed up the search
- **Insert a data entry k**: Apply hash function $h$ on $k$ to obtain the bucket number. Place data entry in that bucket. If no space left, allocate a new overflow page and place data entry in the overflow page. Chain up the overflow page.
- **Delete a data entry k**: Search $k$ and delete

**Static Hashing - Example**

- Assume 2 data entries per bucket and we have 5 buckets
- Insert key values $a,b,c,d,e,f,g$ where $h(a)=1$, $h(b)=2$, $h(c)=3$, $h(d)=4$, $h(e)=5$, $h(f)=6$, $h(g)=7$
- Insert $z,x$ where $h(z)=1$ and $h(x)=5$
- Insert $p,q,r$ where $h(p)=1$, $h(q)=5$, and $h(r)=1$
Static Hashing

- Long overflow chains can develop and degrade performance.
- Number of buckets is fixed. What if file shrinks significantly through deletions?
  - Extendible and Linear Hashing: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by doubling number of buckets?
  - Reading and writing all pages is expensive!
  - Idea: Use directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. No overflow page!
  - Trick lies in how hash function is adjusted!
Example

- Directory is array of size 4.
- **Search**: To find bucket for r, take last "global depth" # bits of h(r);
- If h(r) = 5 = binary 101, it is in bucket pointed to by 01.

Insert h(r)=9 (10012)
(Causes Splitting but no doubling)
Points to Note

- **Global depth of directory** $p$: Maximum number of bits needed to tell which bucket an entry belongs to.
- **Local depth of a bucket** $q$: Number of bits used to determine if an entry belongs to this bucket.
- Each bucket has $2^{p-q}$ pointers from the directory.
- If a bucket has only 1 pointer to it from the directory, splitting the bucket causes doubling. Otherwise, there is more than one pointer to the bucket; when the bucket is split, simply redistribute the pointers.
- **Delete**: Search and delete. Merging with its split image occurs when a bucket becomes empty; if every directory element and its split image directory entry point to the same bucket, shrink directory by 1/2.

Comments on Extendible Hashing

- If directory fits in memory, equality search answered with two disk access
  - 100MB file, 100 bytes per data entry, 4K pages
  - There is about 1,000,000 data entries and 25,000 directory elements; chances are high that directory will fit in memory.
- Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large.
- Multiple entries with same hash value cause problems!