Hash-Based Indexing

• Records in a file are grouped into **buckets**

• Each bucket consists of a **primary page** and zero or more **overflow pages**

• A **hash function** $h$ takes a search-key value $k$ and returns the address of a bucket

• To find records matching a search key value $k$, calculate $h(k)$, then look through bucketed pages sequentially to find matching data entries

Tree-Based Indexing

• Search key values are organized in a **tree**

• The highest level is the **root**

• The lowest level (the **leaf level**) contains data entries

• Each node in the tree is a page on disk
  – Retrieving nodes involves disk I/O
  – And therefore the number of disk reads in a search is equal to the length of the path from root to leaf

• A **B+ tree** is an index structure that ensures all paths from root to leaf are the same length
Comparing File Organizations

- We are interested in the total cost of accessing and modifying data with a given file organization scheme.

- Specifically, what is the cost of:
  - Scan (fetch all records in a file)
  - Search with equality selection (fetch records that match an equality condition)
  - Search with range selection (fetch records that match a range condition)
  - Insert (insert a new record into a file)
  - Delete (delete a record from a file)

Cost Model

- To estimate cost, we need a model of total execution time.

- Our model is a simplified one:
  - B is the number of pages (assuming 100% capacity)
  - R is the number of records per page (100% capacity)
  - D is the average time to read/write a page from/to disk
  - C is average time to process a record

- Consider the average case.

- This is good enough to indicate trends.
Heap Files

- Heap file = randomly ordered records

- Costs:
  - Scan: $B(D+RC)$
  - Search with equality selection: $0.5B(D+RC)$
    - (if equality field is a key; same as scan if not)
  - Search with range selection: $B(D+RC)$
  - Insert: $2D+C$
  - Delete: search cost + $C+D$

Sorted Files

- Records stored directly, sorted on one or more fields

- Costs:
  - Scan: $B(D + RC)$
  - Search with equality selection: $D \log_2 B + C \log_2 R$
    - (assuming selection field is the sort field)
  - Search with range selection: $D \log_2 B + C \log_2 R$
  - Insert: search + $B(D + RC)$
  - Delete: search + $B(D + RC)$
Clustered File/Tree Index

- Sorted file with B+-tree index on sort field
  - $F = \text{fanout}$, max number of pointers in a node

- Assumption: pages at 67% capacity

- Costs:
  - Space overhead: $0.5B$
  - Scan: $1.5B(D + RC)$
  - Search with equality selection:
    - $D \log_b(1.5B) + C \log_2 R$
  - Search with range selection: same
  - Insert: search + D
  - Delete: same

Heap File/Unclustered Tree Index

- Unsorted file with B+-tree on search key

- Assumption: index data entry = 10% record size

- Costs:
  - Space overhead: $0.15B(F - 1)$
  - Scan: $BD(R + 0.15) + 2BRC$
  - Search with equality selection:
    - $D \log_b(0.15B) + C \log_2(6.7R) + D$
  - Search with range selection: “same” (plus another $D$ for each matching record)
  - Insert: $3D + C + D \log_b(0.15B) + C \log_2(6.7R)$
    - (heap insert + tree search + leaf node page write)
  - Delete: heap delete + tree search + leaf node page write
### Heap File/Unclustered Hash Index

- **Unsorted file with hash function on search key**
  - \( H = \text{time to compute hash function} \)

- **Assumption:** index pages are 80% full

- **Costs:**
  - **Space overhead:** \( 0.125B \)
  - **Scan:** \( BD(R + 0.125) + 2BRC \)
  - **Search with equality selection:** \( H + 2D + 4RC \)
  - **Search with range selection:** \( B(D + RC) \)
  - **Insert:** heap insert + \( H + 2D + C \)
  - **Delete:** search + \( 2D \)

### Comparison of I/O Costs

<table>
<thead>
<tr>
<th>File type</th>
<th>Scan</th>
<th>Equal search</th>
<th>Range search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>BD</td>
<td>0.5BD</td>
<td>BD</td>
<td>2D</td>
<td>Search + D</td>
</tr>
<tr>
<td>Sorted</td>
<td>BD</td>
<td>Dlog, B</td>
<td>Dlog, B + m</td>
<td>Search BD</td>
<td>Search + BD</td>
</tr>
<tr>
<td>Clustered</td>
<td>1.5BD</td>
<td>Dlog, 1.5B</td>
<td>Dlog, 1.5B + m</td>
<td>Search D</td>
<td>Search + D</td>
</tr>
<tr>
<td>Tree Index</td>
<td>BD(R + 0.15)</td>
<td>D(1 + log, 0.15B)</td>
<td>D(log, 0.15B + m)</td>
<td>D(3 + log, 0.15B)</td>
<td>Search + 2D</td>
</tr>
<tr>
<td>Hash Index</td>
<td>BD(R + 0.125)</td>
<td>2D</td>
<td>BD</td>
<td>4D</td>
<td>Search + 2D</td>
</tr>
</tbody>
</table>

- **Best for scanning:** sorted
- **Best for equality search:** hash index
- **Best for range search:** clustered
- **Best for insertions:** heap
- **Best for deletions:** heap

- **Worst for scanning:** any index
- **Worst for equality search:** tree index
- **Worst for range search:** hash index
- **Worst for insertions:** sorted
- **Worst for deletions:** sorted
Creating Indexes in MySQL

CREATE INDEX indexName
  ON Relation (A1, ...)
  USING [HASH|BTREE]

- Creates an index of the specified type on the specified column(s)
- MySQL automatically maintains it with inserts/updates/deletes
- MySQL creates clustered indexes on primary keys by default
  - If no primary key, clustered index on first UNIQUE NOT NULL field
  - If no UNIQUE NOT NULL field, clustered index on internal row ID

Indexes and Performance Tuning

- A **workload** is a mix of queries and update operations
- We’d like to create indexes that will support the expected workload efficiently
- For each query in the workload:
  - What relations does it access?
  - What attributes are retrieved?
  - Which attributes are involved in select/join clauses?
  - How selective are those conditions?
- For each update in the workload:
  - What type of update (INSERT/UPDATE/DELETE)?
  - What attributes are affected?
Choosing Indexes

• What indexes should we create?
  – Which relations need indexes?
  – What fields should be used as search keys?
  – Do we need more than one index for a relation?

• What type of index?
  – Clustered? Hash? B+-tree?

Index Selection Guidelines

• Attributes in WHERE clauses are good candidates
  – Many exact matches ➔ hash
  – Many range queries ➔ tree

• If WHERE clauses often contain several conditions, consider multi-attr index
  – Order of attributes matters!

• Choose indexes that benefit as many queries as possible
Index Design Examples

• Range selection: SELECT E.dnum FROM Employees E WHERE E.age > 40
  – B+ tree index supports; hash index does not
  – Is B+ tree index worthwhile?
    • Consider selectivity as well as clustering

Index Design Examples

• Range selection with grouping: SELECT E.dnum, COUNT(*) FROM Employees E WHERE E.age > 10 GROUP BY E.dnum
  – B+ tree index on age again, but this time almost certain that age > 10
  – What about index on dnum?
Composite Search Keys

• A composite (or concatenated) search key is one that contains several fields
  – Equality queries specify a constant value for every field in the key
  – Range queries have either missing values or range tests

• Composite keys support more queries
  – But are larger and require more updates

Composite Key Design

• SELECT E.eid FROM Employees E WHERE E.age BETWEEN 20 AND 30 AND E.sal BETWEEN 3000 AND 5000
  – Hash index does not support, but B+ tree index does
  – Compare index on <sal, age> to index on <age, sal>
    • Assuming similar selectivity of clauses
Composite Key Design

• SELECT E.eid FROM Employees E WHERE E.age = 25 AND E.sal BETWEEN 3000 AND 5000
  – Compare index on <age, sal> to index on <sal, age>

• SELECT AVG(E.sal) FROM Employees E WHERE E.age = 25 AND E.sal BETWEEN 3000 AND 5000