Access Paths: B+ Trees, Hash Tables and their variants

Review

- Record/Tuple: fundamental unit
- Key: Unique subfield of a record
- Blocks: fixed region of disk storage
- Pages: in-memory representation of a block
- Database File:
  - collection of records with similar structure.
  - storage allocation done on basis of primary key
  - Clustered/Non-clustered

Are access paths important?

- Advantages:
  - Avoids sequential scans – Fast lookups
  - Eases database modification
- Disadvantages:
  - Increase in database modification time

Types of Access Paths

- Ordered
  - Sorted
  - Key comparison
  - E.g B+ tree, k-d tree
- Hashed
  - Key transformation
  - E.g Hash tables, extensible hashes, grid files, partitioned hash functions

How to choose?

- Depends on application semantics
- Compare on following properties
  - Access Types
    - Support various query types for e.g range queries, nearest neighbor queries
  - Access Time
    - Search Time
  - Cost of Updates (Inserts/Deletes)
    - Time to insert/delete a record
  - Space Overhead
    - Amount of disc space used by a method

Types of Ordered Indices
**B+ Tree - Definition**

- Typical node of a B+ tree of order n:
  \[ \begin{array}{cccc}
  k_1 & k_2 & \ldots & k_n \\
  P_1 & P_2 & \ldots & P_n \\
  \end{array} \]

- Key Sequence: \( k_1 < k_2 \leq k_3 \leq \ldots \leq k_{n-1} < k_n \)

- Pointer Reference: Each \( P_i \) points to a subtree such that:
  - \( k_j \) in the subtree the following hold true:
    \( k_i \leq k_j < k_{i+1} \)

- 50% Rule:
  - All nodes, except root, should have at least \( \lceil n-1/2 \rceil \) non-empty keys.
  - The root should have at least one key.

- Balance Property: All leaves are on the same level.

**B+ Tree - Example**

- \( N = 5 \)
- Min nodes = 2
- \( H = 2 \)

**B+ Tree - Range Search**

- Search for all values between \( G \) and \( K \)

**B+ Tree - Insert**

- Find node (recursive)
- Key exists?
  - Yes
    - Stop
    - No
    - Add Entry
    - Overflow?
      - No
        - Insert (key) into the leaf node
        - Done
      - Yes
        - Split the node into two nodes on the same level
        - Distribute keys evenly
        - Leaf?
          - Yes
            - Push Middle Key
            - No
              - Copy Middle Key

**B+ Trees - Before Insert**

- Insert \( J \)
**B+ Tree - After Insert**

**B+ Tree - Insert W,X,Y**

**B+ Tree - Insert**

**Insert - Implications**
- Splits grow a tree
- Growth may make the tree wider or longer
- The 50% rule insures structural properties of a B tree
- Balancing progresses from the leaf node up the tree toward the root

**B+ Tree - Delete**

**B+ Trees - Delete**
**B⁺ Trees – Delete**

- Delete L

```
M
```

**B⁺ Trees – Delete**

- Delete F

```
M
```

**Delete - Implications**

- Balance Property is maintained in case of underflow
- Avoid borrow and merge
- Lazy deletion

**How to find height of B⁺ tree**

- \( N \) = number of tuples to index
- \( C \) = average number of entries in leaf node
- \( F \) = average number of entries in index node

\[
H = 1 + \left\lfloor \log_h \left( \frac{\left\lceil \sqrt[3]{N/C} \right\rceil}{2} \right) \right\rfloor
\]

- A large value of \( F \) and \( C \) will reduce \( H \).
- Storage \( S = S_i(F)^{i-1} \quad 1 \leq i \leq h \)

**How H influences B⁺ tree efficiency**

- For varying \( H, F = 300, C = 300 \)

<table>
<thead>
<tr>
<th>( H )</th>
<th>( N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>90,000</td>
</tr>
<tr>
<td>3</td>
<td>7,000,000</td>
</tr>
<tr>
<td>4</td>
<td>3,100,000,000</td>
</tr>
</tbody>
</table>
Hash File Organization

- Avoids access to an index structure for key comparison.
- Is a key transformation to locate a record.
- Minimal access and storage costs
- Not efficient for range queries
- Primary and Secondary Hashing
- Static Vs Dynamic Hashing

Hashing

- Say, \( h(k) \mod N \) returns the bucket to which data entry with key \( k \) belongs (\( N = \# \) of buckets)

Disadvantages of Static Hashing

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

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

Extendible Hashing

- \( i \) grows and shrinks with the size of database
- \( j_k \) = number of bits in common hash prefix
- \( j_k < i \)

Insert

- \( i = j_k \)
  - One entry in bucket address table points to bucket \( i \)
  - Increase in size of bucket address table
  - Rehash remaining values in bucket \( j \)
  - \( i < j_k \)
  - Multiple entries in bucket address
  - Just split the bucket and increase \( j \) by 1

Delete

- Same tradeoffs as in B+ trees
Performance

• Access Time
  - An extendible hashing index has two levels
    - Directory + Bucket

• Insert Time
  - Minimum
    - Read the directory and one bucket
    - Write the bucket
  - Maximum
    - Read the directory and one bucket
    - Split and write two buckets and the directory

• Space Overhead
  - Does not guarantee minimum occupancy of buckets