CS315 Assignment 1: Alumni Hunt Report

Cache Commanders
Abhinav Garg, Akkinepally Kruthi, Ashutosh Kumar, Prattipati Mokshagna, Vikas Yadav

Abstract

This assignment aims to develop an efficient indexing and query execution engine for alumni data, emphasizing indexing strategies, data layout, and query optimization to enhance performance. This report details the design and implementation, including the use of B+ trees for name indexing, year-sorted data storage for efficient temporal queries, and techniques to minimize disk seek times.

1 Indexing, Statistics and Disk Layout

1.1 Choosing Indices

We considered two data structures for indexing: *Trie* and *B+ Trees*. Initial implementations for name queries were developed for both structures, allowing for performance comparisons.

The following outlines our implementation of the Trie:

```
----------------------------------
def TrieNode():
    child = [None] * 26
    ids = []
    return [child, ids]
----------------------------------
```

And the B+ Tree implementation is structured as follows:

```
----------------------------------
class Node:
  def __init__(self, leaf=False):
    self.ind = 0
    self.leaf = leaf
    self.n_child = 0
    self.children = []
    self.keys = []
    self.next = None

class Btr:
  def __init__(self, n=10):
    self.n = n
    self.leaves = []
----------------------------------
```

Our evaluation indicated that the B+ Tree implementation outperformed the Trie, particularly in terms of construction time (*t_build*) and index size (*idx_size*), while both structures exhibited comparable performance in other metrics. Consequently, we selected B+ Trees for indexing alumni names due to their efficiency in handling range queries and exact matches. Additionally, we implemented a secondary array-based structure to index graduation years, offering constant-time access

for year-based queries. This dual indexing approach optimizes query performance across both name and year attributes.

1.2 Index Construction Algorithm

1. **B+ Tree Construction** : This B+ tree implementation consists of a 'Node' class for individual nodes and a 'BTree' class for managing the overall structure, including insertion, depth-first traversal, and pattern-based search capabilities.

- **Node Class**:
  - *Purpose*: Represents a single node in the B+ tree.
  - *Attributes*:
    - `ind`: An index used for leaf nodes.
    - `leaf`: A boolean indicating if the node is a leaf (i.e., has no children).
    - `n_child`: Number of children currently held in the node.
    - `children`: A list containing references to child nodes or data elements (for leaf nodes).
    - `keys`: A list of keys (used for ordering and searching).
    - `next`: Points to the next leaf node, enabling sequential traversal.
  - *Functionality*: Serves as a container for the keys and references to child nodes.
- **BTree Class**:
  - *Purpose*: Manages the B+ tree structure, providing methods for insertion, depth-first traversal, and pattern-based retrieval.
  - *Initialization*:
    - `n`: The maximum number of children each node can hold, controlling the tree's branching factor.
    - `leaves`: A list to keep track of leaf nodes, enabling efficient access to leaf operations.
  - **Methods**:
    - `dfs` (Depth-First Search): Traverses the tree from a given node, linking leaf nodes sequentially through `next` pointers and indexing each leaf in `leaves`.
    - `create` (Tree Creation): Constructs the tree from a list of key-value pairs, creating leaf nodes up to `n` keys per node. Excess keys prompt the creation of new leaf nodes, linked via `next`. Internal nodes are then built by grouping leaves, creating higher levels up to the root.
    - `search_level_name` (Single-Level Name Search): Searches for a specific key starting from a given node.
      - For leaf nodes, it returns the associated data if the key is found.
      - For non-leaf nodes, it navigates to the appropriate child node and continues searching.
    - `search_name` (Tree-Wide Name Search): Calls `search_level_name` starting from the root, providing a simple interface to search the entire tree for an exact match.
    - `lower_bound` (Find Lower Bound): Finds the first key that matches or exceeds a given pattern. If the node is a leaf, it iterates through keys or moves to the next leaf if needed. In non-leaf nodes, it navigates to the correct child node for further search.
    - `upper_bound` (Find Upper Bound): Similar to `lower_bound`, it finds the last key that matches or is just below the pattern, allowing for range-based searches.
    - `pattern_search` (Pattern-Based Search): Retrieves all records matching a pattern, bounded by calculated lower and upper bounds using `lower_bound` and `upper_bound`.

This B+ tree implementation enables efficient insertion, lookup, and range-based searches. By linking leaf nodes sequentially, it supports fast traversal for ordered access and range queries, making it suitable for applications requiring efficient indexing of large datasets.

2. **Year Index Construction**

This function constructs an index by grouping data entries based on the 'year' attribute, assuming the data is sorted by year. The resulting index allows for efficient access to records for specific years by storing the start and end indices for each year.

- `build_by_year` **Function:**
  - *Input*: A list of tuples, where each tuple contains data with the year as its third element (i.e., `tuples[i][2]`).
  - *Output*: A list called `group_by_year`, which stores start and end indices in `tuples` for each year from 1900 to 2100.
- **Algorithm Overview**:
  - **Initialize Variables**:
    - `group_by_year`: An empty list that will store start and end indices for each year.
    - `current_index`: Tracks the current position in the sorted list `tuples`.
    - `total`: Holds the total number of tuples for iteration bounds.
  - **Iterate Through Each Year (1900-2100)**:
    - For each `year` in the range, initialize `start` to the current value of `current_index`.
    - Use a `while` loop to move `current_index` forward until the year of the current tuple no longer matches the `year`.
    - Define `end` as the last index where the year matched.
    - Append the range `[start, end]` to `group_by_year`, with each entry representing the range of indices corresponding to each year.
  - **Result**: The `group_by_year` list contains 201 entries (one for each year from 1900 to 2100), with each entry indicating the start and end index in `tuples` for records of that year. This enables direct access to all records from a specific year without additional searches.

This index structure is particularly useful for fast year-based queries, as each range provides immediate access to all records for a given year.

1.3 Statistics Calculation

To determine the appropriate index for a given query, we rely on the query type, as explained in the **Query Processing** section of this report. For year-based queries, the only statistic calculated and maintained is the **length of the original data**, which facilitates efficient range determination without additional computational overhead. This streamlined approach optimizes performance by minimizing unnecessary calculations, ensuring that only relevant data length information is retained for rapid query processing.

1.4 Disk Layout Design

The data layout on disk has been strategically designed to minimize seek time and optimize query performance:

1. **Name-based Order**: Alumni IDs are stored in ascending order based on names. This organization allows for contiguous access patterns, which significantly enhances the performance of name-based queries.

2. **Year-based Order**: The second storage order arranges alumni IDs sorted by their graduation year. This layout enables efficient data retrieval for queries that focus on the year of graduation.

The first half of the disk is allocated for Name-based order, while the second half is dedicated to Year-based order. This dual-order storage strategy results in additional disk usage but effectively reduces seek times for different types of queries. By organizing the data sequentially according to the relevant criteria for each query type, we ensure swift access and improved overall performance.

2 Query Processing and Optimization

2.1 Query Types and Handling

1. **Single Predicate Queries**:

- **Name-based Queries**: For queries that involve names, the *B+ tree index* is utilized.
  - For exact match queries (e.g., 'name ='), the B+ tree's `search_name` method is employed to efficiently locate the required entries.
  - For pattern matching queries (e.g., 'name LIKE'), the `pattern_search` method traverses the linked leaves within the specified pattern range, ensuring accurate retrieval of all relevant records.
- **Year-based Queries**: *The year index provides the start and end indices for each year*, enabling direct access to records based on the specified comparison operators (i.e., '=', '>=', and '<='). This structure allows for efficient querying based on graduation year, facilitating swift data access.

2. **Conjunctive Queries**:

For queries that combine name and year conditions, the handling process involves two steps:

- **Step 1: Name Condition Evaluation**: The B+ tree index is utilized to identify alumni that match the specified name condition.
- **Step 2: Year Condition Verification**: For each alumnus retrieved from Step 1, the year condition is checked. This approach minimizes the amount of data accessed, improving the efficiency of the query processing. This verification is managed by the `update_locations` function.

It is important to note that for conjunctive queries, indices corresponding to the first half of the disk (which follows the *Name-based Order*) are returned. If the second half of the disk (corresponding to the *Year-based Order*) is used, the seek time (*t_seek*) increases significantly.

2.2 Optimizations

1. **Linked B+ Tree Leaves** : Linking leaves enables efficient range scanning for pattern-matching queries without re-traversing the tree.
2. **Year Index Lookup** : Array-based year indexing allows constant-time lookup, improving performance for date-based queries.
3. **Disk Layout Strategy** : Storing data in two orders (name and year) reduces the need for random disk accesses, minimizing seek time.

3 Evaluation and Results

3.1 Seek Time and Read Time Calculation

Seek time is minimized by sequentially accessing data blocks in name or year order. For example, queries involving alumni names use the linked B+ tree leaves to access contiguous blocks. Similarly, year-based queries use the contiguous year index to reduce seeks. This layout significantly improves performance by reducing non-sequential disk reads.

4 Conclusion

The indexing engine efficiently handles various query types by using a combination of B+ trees and year-based sorted data structures. The dual-disk layout, while increasing space usage, optimizes seek times and read performance. Further improvements could include dynamic indexing strategies to handle varying data distributions.

References

Abraham Silberschatz, Henry Korth and S. Sudarshan - Database System Concepts. 7-McGraw-Hill Education (2020)