Briefing Document: Foundations of Database Searching

Date: October 26, 2023**Source:** Excerpts from "02 - Foundations.pdf" by Mark Fontenot, PhD, Northeastern University

Subject: Fundamental Concepts in Database Searching and Data Structures

This briefing document summarizes the foundational concepts related to searching within database systems, as presented in the provided excerpts. The document covers the importance of searching, basic search algorithms (linear and binary), data storage considerations (arrays and linked lists), and the need for more efficient data structures for database indexing.

Main Themes and Important Ideas:

1. Searching as a Core Database Operation:

The document emphasizes that searching is the most frequently performed operation in a database system. This highlights the critical importance of efficient search mechanisms for overall database performance.

"# Searching is the most common operation performed by a database system"

Furthermore, the `SELECT` statement in SQL is identified as a versatile and complex tool, often involving search operations.

"# In SQL, the SELECT statement is arguably the most versatile / complex."

2. Baseline Search Algorithm: Linear Search:

Linear search is introduced as the baseline for search efficiency. It involves examining each record in a collection sequentially until the target is found or the end of the collection is reached.

"# Baseline for efficiency is Linear Search" "# Start at the beginning of a list and proceed element by element until: ■ You find what you're looking for ■ You get to the last element and haven't found it"

The time complexity of linear search is discussed:

"# Worst case: target is not in the array; n comparisons" "# Therefore, in the worst case, linear search is O(n) time complexity."

This highlights that in the worst-case scenario, the time taken for a linear search grows linearly with the number of records.

3. Data Storage Structures and Their Impact on Search:

The document discusses two fundamental ways to store collections of records: contiguous allocation (arrays) and linked lists.

•

Arrays (Contiguously Allocated Lists): Offer fast random access because elements are stored in a single block of memory. However, inserting elements in the middle of an array is slow as subsequent elements need to be shifted.

"# Arrays are faster for random access, but slow for inserting anywhere but the end"

Illustration: "5 records had to be moved to make space" (during insertion).

•

Linked Lists: Allow for fast insertion and deletion of elements at any position because only memory addresses (pointers) need to be updated. However, random access is slow as it requires traversing the list from the beginning.

"# Linked Lists are faster for inserting anywhere in the list, but slower for random access"

Illustration: "Extra storage for a memory address" (overhead of linked lists).

The document summarizes the trade-offs:

Observations:

- Arrays - fast for random access
- slow for random insertions
- Linked Lists - slow for random access
- fast for random insertions

4. Improved Search Algorithm: Binary Search:

Binary search is presented as a more efficient search algorithm, but it requires the data to be in sorted order and typically stored in an array (or a structure that allows efficient random access).

"# Input: array of values in sorted order, target value"

The algorithm works by repeatedly dividing the search interval in half.

Illustration of Binary Search: Shows how the `left` and `right` pointers are adjusted based on the comparison with the `mid` element.

The time complexity of binary search is significantly better than linear search:

"# Worst case: target is not in the array; log2 n comparisons" "# Therefore, in the worst case, binary search is O(log2n) time complexity."

This logarithmic time complexity means that the search time grows very slowly as the number of records increases, making it much more efficient for large datasets compared to the linear growth of linear search.

5. Challenges of Database Searching Beyond Simple Algorithms:

The document then transitions to the complexities of searching within a database stored on disk. It highlights that data is typically stored by column IDs and their values.

"# Assume data is stored on disk by column id’s value"

While searching for a specific ID is fast, searching for a specific value in another column (e.g., `specialVal`) necessitates a linear scan of that entire column.

"# Searching for a specific id = fast." "# But what if we want to search for a specific specialVal?" "# Only option is linear scan of that column"

A key challenge is that data cannot be physically sorted on disk based on multiple attributes simultaneously without data duplication, which is space-inefficient.

"# Can’t store data on disk sorted by both id and specialVal (at the same time)" "# data would have to be duplicated → space inefficient"

6. The Need for External Data Structures (Indexing):

To overcome the limitations of linear scans for non-key attributes, the document emphasizes the necessity of external data structures, which serve as indexes.

"# We need an external data structure to support faster searching by specialVal than a linear scan."

The document explores initial ideas for such structures:

- **Sorted Array of Tuples (specialVal, rowNumber):** Allows for efficient searching using binary search but suffers from slow insertions.

"# An array of tuples (specialVal, rowNumber) sorted by specialVal" "a) We could use Binary Search to quickly locate a particular specialVal and find its corresponding row in the table" "b) But, every insert into the table would be like inserting into a sorted array - slow…"

•

Sorted Linked List of Tuples (specialVal, rowNumber): Offers fast insertions but requires a linear scan for searching.

"# A linked list of tuples (specialVal, rowNumber) sorted by specialVal" "a) searching for a specialVal would be slow - linear scan required" "b) But inserting into the table would theoretically be quick to also add to the list."

7. Towards More Efficient Data Structures: Binary Search Trees:

The document concludes by posing the question of needing a data structure that offers both fast searching and fast insertion, hinting at more advanced data structures like Binary Search Trees.

"# Something with Fast Insert and Fast Search?" "# - Binary Search Tree - a binary tree where every node in the left subtree is less than its parent and every node in the right subtree is greater than its parent."

This sets the stage for further discussion on data structures optimized for database indexing.

Conclusion:

The provided excerpts lay a crucial foundation for understanding the challenges and fundamental approaches to searching in database systems. It highlights the trade-offs between different data storage methods and search algorithms, ultimately leading to the recognition of the need for specialized external data structures (indexes) to achieve efficient searching on non-primary key attributes. The introduction of the Binary Search Tree suggests a direction for exploring more advanced indexing techniques.