**Briefing Document: Fundamentals of Database Searching**

**1. Searching as a Fundamental Database Function**

Searching is the most frequently executed operation within a database system, making efficient search techniques crucial for database performance.

- "Searching is the most common operation performed by a database system."
- SQL's **SELECT** statement is one of the most complex and versatile commands, often involving searches.
- "In SQL, the SELECT statement is arguably the most versatile / complex."

**2. Linear Search: The Baseline Algorithm**

Linear search serves as the foundational method for evaluating search efficiency. It operates by sequentially examining each element in a dataset until the target value is found or the dataset is exhausted.

- "Baseline for efficiency is Linear Search."
- The process involves:
    - Starting at the first element and checking each sequentially.
    - Stopping when the target value is found or the end of the dataset is reached.

The worst-case scenario involves examining every element, leading to a time complexity of **O(n)**.

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

**3. Impact of Data Storage Structures on Search Performance**

Databases rely on two primary data storage methods, each with distinct advantages and drawbacks:

**Arrays (Contiguously Allocated Lists):**

- Enable fast random access since elements are stored sequentially in memory.
- Slow insertion performance due to the need to shift elements when inserting in the middle.
- "Arrays are faster for random access, but slow for inserting anywhere but the end."
- Example: Inserting an element may require shifting multiple records.

**Linked Lists:**

- Facilitate quick insertion and deletion at any position, as they only require pointer updates.

- Slower random access because traversal must begin at the head of the list.
- "Linked Lists are faster for inserting anywhere in the list, but slower for random access."
- Example: Extra memory is needed to store pointers.

**Comparison Summary:**

- Arrays: Fast for random access but slow for insertions.
- Linked Lists: Efficient for insertions but slow for random access.

### 4. Binary Search: A More Efficient Searching Method

Binary search provides a significant improvement over linear search but requires a sorted dataset, typically stored in an array or a structure supporting rapid access.

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

The algorithm reduces the search space by half in each step, adjusting **left** and **right** pointers accordingly.

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

Since binary search exhibits logarithmic time complexity, search performance improves drastically for large datasets compared to linear search.

### 5. Complexities of Searching in Database Systems

Searching in databases stored on disk presents additional challenges. Data is often stored by column IDs and values.

- "Assume data is stored on disk by column id's value."

Efficient ID-based searches are possible, but searching by non-key attributes (e.g., **specialVal**) requires a full column scan.

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

A key limitation is that physical sorting on disk cannot accommodate multiple attributes without duplicating data, 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 Role of Indexing in Enhancing Search Performance

To address the inefficiencies of linear scans for non-key attributes, external data structures—indexes—are necessary.

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

Two initial indexing approaches are explored:

**Sorted Array of Tuples:**

- Enables binary search for efficient lookups.
- However, insertions are slow due to the need to maintain order.
- "An array of tuples (specialVal, rowNumber) sorted by specialVal."
- "Binary Search allows quick lookup, but insertions are slow."

**Sorted Linked List of Tuples:**

- Provides quick insertions but requires a full scan for searches.
- "A linked list of tuples (specialVal, rowNumber) sorted by specialVal."
- "Search is slow (linear scan), but insertions are fast."

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

A well-designed indexing structure should optimize both search and insertion performance. This leads to the consideration of **Binary Search Trees (BSTs)**.

- "Something with Fast Insert and Fast Search?"
- "Binary Search Tree: A binary tree where every node in the left subtree is smaller than its parent, and every node in the right subtree is greater."