Briefing Document: B-Trees and B+ Trees
Date: October 26, 2023 Prepared For: [Intended Audience - e.g., Software Development Team, Database Architects] Prepared By: Gemini AI Subject: Review of B-Trees and B+ Trees for Disk-Based Data Structures

This briefing document summarizes the key concepts, characteristics, and importance of B-trees and their common variant, B+ trees, as described in the provided sources. These data structures are fundamental for efficient management of large-scale, disk-based systems requiring insertion, deletion, and key range searches.

1. Introduction and Historical Context
B-trees, attributed to R. Bayer and E. McCreight in 1972, rapidly became the dominant method for large file access by 1979, surpassing most alternatives except hashing.
B-trees and their variants are the "standard file organization for applications requiring insertion, deletion, and key range searches" and are used in "most modern file systems."
Their primary design goal is to effectively address the challenges of implementing search trees on disk.
2. Key Problems Addressed by B-Trees (According to "CS3 Data Structures & Algorithms.pdf")
The source highlights four major problems encountered with disk-based search trees that B-trees effectively address:

Minimizing Disk Accesses for Search: B-trees are "shallow" due to being "always height balanced (all leaf nodes are at the same level)" and having a "quite high" branching factor. This ensures that "only a small number of disk blocks are accessed to reach a given record."
Reducing Disk I/O During Updates and Searches: Operations in a B-tree "affect only those disk blocks on the path from the root to the leaf node containing the query record." Fewer affected blocks translate to less disk I/O.
Optimizing Range Searches: B-trees "keep related records (that is, records with similar key values) on the same disk block," which "helps to minimize disk I/O on range searches."
Improving Space Efficiency and Reducing Disk Fetches: B-trees "guarantee that every node in the tree will be full at least to a certain minimum percentage." This "improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation."
3. B-Tree Properties and Structure
A B-tree of order m has specific structural properties:
"The root is either a leaf or has at least two children."
"Each internal node, except for the root, has between ⌈m/2⌉ and m children."
"All leaves are at the same level in the tree, so the tree is always height balanced."
A B-tree is a "generalization of the 2-3 tree" (a 2-3 tree is a B-tree of order three).
Typically, "the size of a node in the B-tree is chosen to fill a disk block." A node can hold "100 or more children."
"A 'pointer' value stored in the tree is actually the number of the block containing the child node."
Disk access is often managed using a "buffer pool and a block-replacement scheme such as LRU."

4. B-Tree Operations

Search: It's an "alternating two-step process" starting from the root:

"Perform a binary search on the records in the current node." If the key is found, return the record.

If the key is not found and the current node is a leaf, report an unsuccessful search. Otherwise, follow the appropriate child pointer based on the binary search result.

Example: Searching for key 47 in a B-tree of order four involves traversing down the tree based on key comparisons at each internal node until the leaf node containing (or that should contain) the key is reached.

Insertion:Find the appropriate leaf node for the new key.

If there is space, insert the key.

If the leaf is full, split it into two and "promote the middle key to the parent."

If the parent also becomes full, the splitting and promotion process repeats up the tree, potentially leading to the root splitting and increasing the tree height.

"Note that this insertion process is guaranteed to keep all nodes at least half full."

Deletion: (Briefly mentioned as complex, with details provided for B+ trees)

5. B+ Trees: A Common Variant

The "B-tree as described in the previous section is almost never implemented." Instead, "a variant of the B-tree, called the B+ tree" is most commonly used. For even greater efficiency, the B* tree is mentioned as a more complex variant.

B+ trees are motivated by the idea of managing a sorted array-based list in chunks (disk blocks) with a tree-like structure to locate the appropriate chunk.

5.1. Key Differences Between B+ Trees and Standard B-Trees

Data Storage: "The most significant difference between the B+ tree and the BST or the standard B-tree is that the B+ tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search."

Node Structure: "Internal nodes are significantly different in structure from leaf nodes." Internal nodes store keys and pointers to child nodes, while leaf nodes store actual records (or keys and pointers to records in a separate file if used as an index).

Leaf Node Linking: "The leaf nodes of a B+ tree are normally linked together to form a doubly linked list." This enables efficient "traversal in sorted order by visiting all the leaf nodes on the linked list," making B+ trees "exceptionally good for range queries."

5.2. B+ Tree Operations

Search: Similar to a regular B-tree, but "the search must always continue to the proper leaf node." Even if the search key is found in an internal node, it's just a guide, and the actual record resides in a leaf.

Insertion:Find the leaf node where the record should be inserted.

If the leaf is not full, add the record.

If the leaf is full, split it into two, distribute the records evenly, and "promote a copy of the least-valued key in the newly formed right node" to the parent.

Parent node splits can also occur, potentially leading to a new root and increased tree height.

"B+ tree insertion keeps all leaf nodes at equal depth."

Deletion:Locate the leaf node containing the record.

If the leaf is more than half full, remove the record.

If deleting causes an "underflow" (node becomes less than half full), try to borrow a record from an adjacent sibling. This might require updating a key in the parent.
If borrowing is not possible, the under-full node "must give its records to a sibling and be removed from the tree" (merge). This merge can propagate up the tree, potentially reducing the tree height if the last two children of the root merge.
5.3. B* Trees: Enhanced Utilization
The B* tree is a variant that aims for higher space utilization.
Instead of splitting a full node into two (as in B+ trees), a B* tree tries to give some records to a neighboring sibling if possible. Only if both the node and its sibling are full do they split into three nodes.
Similarly, during underflow, a node is combined with its two siblings and reduced to two nodes, aiming for nodes to be at least two-thirds full.
While B* trees offer better performance due to higher fill rates, "the update routines become much more complicated."
6. B-Tree and B+ Tree Analysis and Performance
The asymptotic cost of search, insertion, and deletion in B-trees, B+ trees, and B* trees is $\Theta(\log n)$, where n is the number of records.
The "base of the log is the (average) branching factor of the tree," which is typically very high (e.g., 100 or more in database applications).
This high branching factor results in "extremely shallow" trees.
Example: A B+ tree of order 100 with height three can store at least 5,000 records and at most one million records. A height of four can accommodate hundreds of millions of records.
B+ trees guarantee that non-root nodes are at least half full, leading to an average fill rate of around 3/4.
While internal nodes in B+ trees are "purely overhead" as they only store keys for guiding search, the "high fan-out rate" means that the "vast majority of nodes are leaf nodes," so the overhead is low (e.g., approximately 1/75 of nodes might be internal in a B+ tree of order 100).
7. Performance Optimization Techniques
Caching Upper Levels: "The upper levels of the tree can be stored in main memory at all times." Due to the high branching factor, the top few levels require little space and can significantly reduce disk fetches.
Buffer Pool: Using a buffer pool to manage B-tree nodes in main memory can keep frequently accessed nodes readily available, reducing disk I/O. Standard replacement policies like LRU are often sufficient, especially if the buffer pool size is at least twice the tree depth.
8. Conclusion
B-trees and, more commonly, B+ trees are highly effective data structures for managing large datasets on disk. Their balanced structure, high branching factor, and optimized handling of disk blocks minimize disk I/O, leading to efficient search, insertion, and deletion operations. The B+ tree variant, with its separation of keys in internal nodes and data in linked leaf nodes, is particularly well-suited for range queries and is the prevalent choice for modern file systems and database implementations. While more complex variants like B* trees offer potential performance improvements through higher space utilization, the increased complexity of their update routines must be considered. Understanding the principles and trade-offs of these data

structures is crucial for designing and implementing efficient disk-based data management systems.