A B+ tree is a self-balancing tree data structure that is commonly used in databases and file systems for efficient data storage and retrieval[1][5]. Here are some key characteristics of B+ trees:

1. Structure: B+ trees consist of a root, internal nodes, and leaf nodes[1].
2. Data storage: Unlike B-trees, B+ trees store all data records in the leaf nodes, while internal nodes contain only keys and pointers to child nodes[5].
3. Leaf node linkage: All leaf nodes in a B+ tree are linked together in a sequential order, forming a linked list[5].
4. Order: B+ trees have an order 'm', which determines the maximum number of children a node can have (m) and the maximum number of keys it can store (m-1)[3].
5. Balance: All leaf nodes are at the same level, ensuring the tree remains balanced[4].
6. Occupancy: Each node, except the root, must be at least half full, containing between d and 2d entries, where d is the minimum degree of the tree[2].
7. Efficient operations: B+ trees support efficient insertion, deletion, and search operations, typically with a time complexity of O(log n)[3].
8. Range queries: The linked leaf nodes enable fast sequential access and efficient range query processing[5].

These properties make B+ trees particularly well-suited for applications requiring fast data retrieval, especially in systems dealing with large datasets stored on disk[1][5].

To determine why B+ trees outperform AVL trees for indexing large datasets, you would need to compare their structural properties, performance characteristics, and use-case suitability:

Key Comparison Factors

Aspect	B+ Tree	AVL Tree
Node Structure	Stores data only in leaf nodes; internal nodes hold keys/pointers[1][2]	Stores data in all nodes[3]

Tree Height	Shorter due to high fanout (100+ children per node)[2]	Taller binary structure (max height = $2\log N$ $2\log N$)[3][4]
Disk I/O Efficiency	Optimized for block storage with fewer disk accesses[1][2]	Designed for in-memory operations, less disk-friendly[4]
Range Queries	Linked leaf nodes enable sequential scans without tree traversal[1][2]	Requires full tree traversal for range operations[3]
Insertion Cost	Bulk-loading optimizations for large datasets[2]	Frequent rotations increase overhead in write-heavy scenarios[3]
Memory Usage	Smaller internal nodes (keys only) allow more keys per disk block[1][2]	Larger per-node metadata (balance factors)[3]

Critical Information from Sources

1. B+ Tree Advantages
   - Internal nodes contain only keys/pointers, enabling higher fanout and shorter trees[1][2]
   - Leaf nodes form a linked list for
   - $O(1)$
   - $O(1)$ sequential access[1]
   - Better for systems with block-based storage (e.g., databases) due to reduced disk seeks[2]
2. AVL Tree Limitations

- Binary structure leads to taller trees (
- $\log N$
- $\log N$ vs B+ tree's
- $\log_m N$
- $\log$
- m
- N, where
- m
- $m \approx 100$)[3][4]
- No native support for efficient range scans[3]
- Rotation overhead becomes significant with large datasets[4]

3. Performance Tradeoffs
    - B+ trees require ~60% fewer disk accesses for billion-record datasets[1][3]
    - AVL trees become height-bound (
    - $\text{height} \propto \log N$
    - $height \propto \log N$) vs B+ trees' fanout-bound design[2][4]

This comparison shows B+ trees are superior for disk-based large datasets due to their block-storage optimization, while AVL trees remain effective for smaller, in-memory datasets requiring strict balance

B-Trees usually have larger number of keys in single node and hence reducing the depth of the search, in record indexing the link traversal time is longer if the depth is more, hence for cache locality and making the tree wider than deeper, multiple keys are stored in array of a node which improves cache performance and quick lookup comparatively.