1. Overview of the Project

This project explores two key algorithms that have been commonly used in the database management systems: the B+ Tree and the join based hashing for relational algebraic operations.

The primary goals are to:

1. Generate appropriate data for each algorithm’s requirements.
2. For B+ Trees, implement dense and sparse structures with a configurable order, and core operations: search, insertion, deletion, and range search for both structures.
3. For join based hashing, simulate virtual disk and main-memory constraints for joining relations R(A,B) and S(B,C). Design and implement the two-pass natural join operation based on hashing, tracking I/O costs.
4. Test the implementation by setting up experiments, generating relations and call implementation functions, visualize or print the results.

The report is organized into four main sections. Section 2 details the B+ Tree implementation. Section 3 describes the join-by-hashing implementation. Section 4 analyzes experimental results. Section 5 reflects on implementation challenges and insights gained.

2. B+ Trees

2.1 Data Generation

* Generate 10,000 unique integer keys in a specified range (e.g., 100 000–200 000).
* A dense tree stores one record per key; a sparse tree groups keys into fixed-size blocks before insertion.
* Functions used:
  + generate\_records(min\_val, max\_val, total): returns a total number list of unique random keys between min\_val and max\_val.

2.2 Building Dense and Sparse B+ Trees

* Tree order defines maximum capacity: each node holds up to the number of order keys.
* Each “BPlusTreeNode” holds a boolean “is\_leaf” flag, a sorted list of keys, and a children list that in internal pointers to child nodes, for leaf nodes those pointers points to records (dense) or blocks (sparse), plus a next pointer linking leaf nodes.
* Dense mode: insert each key individually.
* Sparse mode: insert each block as a single unit by using the block’s first key as the representative.
  1. create\_blocks(records, block\_size): for sparse mode, sorts the record list and partitions it into blocks of size block\_size.
* Construction: instantiate four trees:
  1. Two dense trees, one with order of 13 and one with order of 24
  2. Two spare trees, one with order of 13 and one with order of 24
  3. For each tree, loop over the generated records and call tree.insert(...).

2.3 B+ Tree Operations

2.3.1 Search

* Uses tree.search(target)
* Start at the root, choose a child pointer by comparing keys, and go down until reaching a leaf.
* At the leaf, linearly scan keys; return the record (dense) or block (sparse) if found, else return None.

2.3.2 Range Search

* Uses tree.range\_search(low, high).
* Just like point search, go down to the leaf containing low.
* Traverse leaf-level linked list (node.next) and collect all keys (or block contents) until exceeding high.

2.3.3 Insertion

* insert(key) dispatches ro either \_insert\_dense(key) or \_insert\_spare(key) base on the tree mode, both insertions follow the similar strategy
* If the root insert into is leaf and the length of keys is less than order, just insert by calling \_insert\_into\_leaf.
* Else when length of keys is greater or equal to order, which indicates overflows. Create a new internal root and append the old “self.root” as its only child, then call \_split\_child
* \_split\_child(parent, idx, child) will allocate a new leaf node (new\_node), compute mid = order // 2, and split the child into two parts.
  + For a leaf node, it moves “child.keys[mid:]” and “child.children[mid:]” into new\_node, updates the next pointer (new\_node.next = child.next; child.next = new\_node), and promotes “new\_node.keys[0]” into “parent.keys[idx]”.
  + For an internal node, it promotes “child.keys[mid]” into “parent.keys[idx]”, moves “child.keys[mid+1:]” and “child.children[mid+1:]” into new\_node.  
    Finally, new\_node is inserted into “parent.children[idx+1].”
  + In one sentence, \_split\_child splits a full child at mid, promotes a key to the parent, and inserts a new sibling node.
* After the split is done, call \_insert\_non\_full, which recursively goes down to the correct child, splits full nodes along the path if necessary, and inserts the key into a non-full leaf.

2.3.4 Deletion

* delete(key) recursively locates and removes the key (or the in-block item for sparse trees).
* After removal, if a node has fewer than ⌈order/2⌉ – 1 keys, \_rebalance is called to restore balance:
  + Redistribution:  
    Borrow a key from the left or right sibling if the sibling has more than the minimum number of keys.
    - In a leaf node, keys and child pointers are moved, and parent keys are updated accordingly.
    - In an internal node, parent keys are also moved down into the child node when redistributing.
  + Merging:  
    If redistribution is not possible, merge the deficient node with a sibling and pull down a separator key from the parent (for internal nodes). In a leaf merge, next pointers are updated to maintain the leaf chain.
* After rebalancing, if the root becomes empty (i.e., has no keys), the tree shrinks by replacing the root with its only child.

2.4 Visualization

* visualize\_full\_bplus\_tree walks every node in breadth-first order, generates a high-resolution Graphviz PDF of the entire tree.
* trace\_path\_to\_leaf finds and returns the path from the root to the leaf node where a given key would be located.
* visualize\_bplus\_tree renders only the nodes on a specified root‑to‑leaf path, highlighting the node affected by an insert or delete.

3. Join by Hashing

3.1 Relation Generation

* Relation S: generate 5 000 B-values in [10 000, 50 000], each paired with a random C-value.
* Store S in a virtual disk as a list of blocks (size 8 tuples each).
* Relation R1: generate 1 000 (A,B) tuples by sampling B-values from S (allows matches).
* Relation R2: generate 1 200 (A,B) tuples with B randomly in [20 000, 30 000], which may have fewer matches in S.

3.2 Virtual Disk & Main Memory

* Relations are stored as virtual disks (lists of blocks), with a block size of 8 tuples and a maximum main memory of 15 blocks.
* disk\_read and disk\_write simulate disk I/O operations and maintain counters for performance tracking, raising a MemoryError if memory overflows.

3.3 Hash Function

* Hash on B: hash\_b(b) = b % (MAX\_MEMORY\_BLOCKS - 1) ensures the number of partitions fits memory minus one for the join buffer.
* This simple modulo scheme uniformly distributes B-values across buckets if the data range is wide.

3.4 Two-Pass Hash-Join Algorithm

1. Partition Phase: scan the R and S relations block-by-block. Each tuple is assigned to a partition based on hash\_b(b).
   * Buffers bufR[i] and bufS[i] are used to accumulate tuples for each partition.
   * Once a buffer fills (8 tuples), it is flushed to disk using disk\_write, and the memory is freed (virtual\_memory.pop()) to avoid overflow.
   * After processing all blocks, any remaining tuples in partially filled buffers are flushed to disk.
2. Join Phase (two\_pass\_hash\_join): For each partition index i,
   * Reset memory and build an in-memory hash table from the S-partition S\_parts[i], reading blocks one by one.
   * Stream through the corresponding R-partition R\_parts[i], probing each R-tuple against the hash table.
   * For every matching (B,C) pair, output the joined (A,B,C) tuple.  
     Memory is reset before each partition’s join to ensure no overflow.
3. I/O costs (disk\_read\_count and disk\_write\_count) are recorded before and after each experiment to evaluate efficiency.

Experimental Results and Performance Analysis

4.1 Tree Construction and Structure

* Dataset: 10,000 unique integer keys generated randomly between 100,000 and 200,000.
* Trees Built:
  + Dense tree (one record per key) and sparse tree (grouped into blocks) for order 13 and order 24.
* Observation:
  + Dense trees had taller structures but more direct key access.
  + Sparse trees had fewer nodes (because of block grouping) but slightly slower searches because of the blocks.

4.2 Insertion and Deletion Operations

* Insertions:
  + 2 random inserts into each dense tree after initial construction.
  + Trees rebalanced correctly through node splits, with visualizations confirming correct structure updates.
* Deletions:
  + 2 random deletions applied to each sparse tree.
  + After deletion, \_rebalance was triggered when nodes fell below minimum occupancy, leading to either redistribution or merging with siblings, again with visualizations confirming correct structure updates.
* Mixed Updates:
  + 5 additional random insertions/deletions applied to all trees to simulate dynamic workloads.
  + All operations worked as expected (sorted keys, linked leaves, balanced height).

4.3 Search and Range Query Results

* In dense B+ trees, single-key searches are faster because each key is stored individually, allowing direct access without in-block scanning.
* In contrast, sparse B+ trees can offer faster range queries when multiple keys are stored within the same block, reducing the number of pointer traversals during sequential access.

4.4 Impact of Order (13 vs 24)

* Order 13:
  + Trees had greater height but smaller node sizes.
  + More frequent splits during insertions.
* Order 24:
  + Trees were shallower with larger nodes.
  + Insertions and deletions caused fewer splits or merges, but node operations were heavier due to more keys per node.

5.1 Experiment R1 with S with Filtering

* Join size: 1000 matching tuples.
* I/O counts: reads ~1 500 blocks, writes ~750 blocks.
* 3 matches after filtering
* Randomly picked 20 B-values from S and outputted only the tuples with matching B-values. Most randomly picked B-values had no matches in R1, as expected in a filtered join.

5.2 Experiment R2 with S

* Join size: significantly smaller ( ~150 matches) due to fewer overlapping B-values.
* I/O counts: reads ~1 550, writes ~775.
* Since R2's B-values were randomly selected from [20,000–30,000], only some B-values overlapped with S. Therefore the join size is relatively small, but many partitions had sparse data, increasing disk I/O slightly (more small flushes due to fewer tuples per partition).

5.3 Impact of Data Set and Organization

* The performance depends on how much overlap exists between R and S. When B-values are randomly chosen from a narrow range, fewer matches occur, leading to smaller join sizes but higher relative I/O overhead.
* Well-distributed data across partitions ensures efficient buffer usage and minimizes disk writes. In contrast, skewed or sparse data causes frequent small flushes, increasing disk I/O even when few join results are produced.

6. Discussion and Reflection

6.1 Implementation Challenges

* B+ Tree Deletion Complexity:  
  Handling underflow, redistributions, and merges across multiple levels required careful indexing and pointer updates.
* Split and Parent Updates During Inserting:  
  Implementing \_split\_child correctly was tricky — splitting a node required carefully moving keys and children to a new sibling, promoting the correct separator key to the parent, and updating both child links and parent pointers to maintain tree balance.
* Visualization Scalability:  
  Rendering a 10,000-key tree at once was too large and difficult to interpret. To better highlight changes before and after operations, trace\_path\_to\_leaf was used to find affected paths and visualize\_bplus\_tree was used to selectively render only parts of the tree instead of the full structure.
* Hash Join Memory Simulation:  
  Strictly enforcing a 15-block memory limit exposed corner cases where memory overflow occurred. It was resolved by immediately popping blocks from virtual\_memory after processing, ensuring buffers stayed within the allowed size.

6.2 Insights Gained

* Algorithmic Trade-offs:  
  Working with different tree orders deepened my understanding of how B+ Tree is structured: higher order reduces tree height but increases the complexity of node operations.
* Balancing I/O and CPU Costs:  
  The two-pass hash join reinforced the classic trade-off: minimizing I/O (fewer partitions) can shift the burden onto CPU and memory usage during join processing.
* Working on this project also made me reflect on how important careful schema design is. For example, in another project, I designed tables like students, courses, and enrollments, which similarly required efficient key lookups and data organization. Just like with B+ trees and hash joins, ensuring that foreign keys and indexes are properly set in the schema is essential for keeping both small queries and large joins fast and reliable.