# Database Algorithms Project Report

*Cherukuri Venkata Surya Krishna - 136004790*

## **Overview**

In this project, I tackled two foundational problems in database systems building efficient indexes with B+ trees and performing scalable joins with hash‑based algorithms under realistic external‑memory constraints. For the B+‑tree portion, I wrote a generic Java implementation parametrized by tree order, then provided two construction strategies:

- **Bulk‑loading (dense)** packs leaf nodes to near capacity before building internal levels, minimizing tree height and maximizing node utilization.

- **Incremental insertion (sparse)** inserts one key at a time into an initially empty tree, yielding a taller structure with more room for future growth.

Beyond construction, I implemented full search, range search (leveraging leaf‑level sibling pointers), insertion (with node splits and root promotion), and deletion (with careful borrowing or merging to rebalance underflowing nodes). Rigorous testing—first on small datasets (orders 3–5) to debug corner cases, then on 10000 randomly generated keys for orders 13 and 24 helped validate correctness and illustrate how density and sparsity affect tree shape, height, and operation cost.

In the join component, I simulated a virtual disk (unlimited blocks, each holding eight tuples) and a fixed 15‑block main memory. I implemented:

1. **Two‑pass hash join**, where both R(A,B) and S(B,C) are first partitioned into 14 buckets on disk using a simple modulo hash, then each corresponding bucket pair is joined in memory via a hash table on the smaller side. I track every disk read/write to measure I/O cost end‑to‑end.

2. **One‑pass hash join**, loads the smaller relation entirely into memory, builds an in‑RAM hash table, and then streams through the larger relation, performing on‑demand lookups against that hash table..

Through experiments joining 1000 R‑tuples drawn from S's B‑values and 1200 with random B's I observed how duplicate keys in S inflate join result sizes and how partitioning reduces memory pressure at the cost of extra I/Os. Overall, this project deepened my understanding of how physical data layout, buffering strategies, and algorithmic invariants interact in real‑world DBMS implementations.

## B+ Trees

### Data Generation (Part 1)

We first generate 10000 unique integer keys in the range [100000, 200000]. Using Java's `Random` and a `HashSet<Integer>`, we sample without replacement until we collect 10000 distinct values, then convert them into a list for tree construction.

### Building B+ Trees (Part 2)

Two construction strategies live in `TreeBuilder`:

1. **Dense (bulk-load) builder**
    - **Sort** the 10000 keys ascending.
    - **Pack leaves**: group up to (*order* − 1) keys per leaf node.
    - **Link** the leaves into a doubly-linked list so that range scans simply follow leaf pointers.
    - **Promote separators**: for each leaf except the first, take its first key as a separator and group child pointers into internal nodes of capacity *order*.
    - **Repeat** promotion up levels until a single root remains
    - This produces a tree where every non-root node is as full as possible, minimizing height.

2. **Sparse (incremental insert) builder**
    - **Sort** the keys.
    - **Insert** one key at a time into an initially empty B+ Tree:
    - Descend via internal separators, insert into the appropriate leaf, split if full, and propagate splits upward.

    - Nodes tend to hover near the minimum fill threshold (≈ ⌈(*order*−1)/2⌉ entries), often creating a taller tree.

### Operations on B+ Trees (Part 3)

We provide:

- **Exact Search**: recursively choose child pointers in `InternalNode` until reaching a `LeafNode`, then scan for the key.
- **Range Search**: find the leaf for the lower bound, collect keys in-range, then follow the `next` pointers until exceeding the upper bound.
- **Insertion**: identical to the sparse builder's split logic, but applied dynamically on any tree.
- **Deletion**: recursively remove the key, and if a node underflows, attempt to borrow from a sibling or merge and propagate changes upward. Internal separators are updated as leaves change.

**Experiments (Part 4)**

Using **`BTreeTester`**, we can do:

1. **Generation** of 10000 keys.
2. **Construction** of four trees:
   1. Dense order 13
   2. Sparse order 13
   3. Dense order 24
   4. Sparse order 24
3. **Operation tests** for each tree:
   1. 2 random insertions on each dense tree
   2. 2 random deletions on each sparse tree
   3. 5 additional mixed insert/delete ops on all four
   4. 5 search + range queries on all four

**Key observations**:

- **Dense, order 13** had height 3, high fan-out, shallow lookups, but bulk-load cost.
- **Sparse, order 13** reached height 4–5, with more frequent splits during insert.
- **Dense, order 24** shrank to height 2–3—excellent for read-heavy workloads.
- **Sparse, order 24** sat at height 3–4, balancing build time against search depth.

Printouts of node states before/after each split, borrow, or merge confirmed that our implementation correctly maintains B+ Tree invariants.

## Join by Hashing

**Data Generation (Part 1)**

- **Relation S(B,C)**: 5000 tuples, B ∈ [10000, 50000], C random [0, 999].
- Stored on **`VirtualDisk`** in blocks of 8 tuples; block indices are recorded for later access.

**Virtual Disk I/O (Part 2)**

- **`VirtualDisk`**: an unbounded `List<DiskBlock>`, each read/write via `readBlock`/`writeBlock` counts one I/O.
- **`MainMemory`**: an array of 15 in-RAM `DiskBlock` slots; `loadBlock(i, block)` brings one disk block into slot *i*.

**Hash Function (Part 3)**

- **Phase 1 (hash1): `floorMod(B, P)`** where *P* = M−1 = 14 partitions. Ensures uniform bucket distribution.
- **Phase 2 (in-memory join)**: we rely on Java's built-in **`HashMap`** (its **`hashCode`** and collision-resolution) to bucket R's tuples and probe with S's tuples..

**Join Algorithm (Part 4)**

1. **Two-Pass Hash Join**
    - **Partition Phase**
        - For each block of R (and separately S):
            - `readBlock` → `loadBlock` → for each tuple compute `p = hash1(B)` and buffer it in `buffers[p]`.
            - When a buffer reaches 8 tuples (block-full) or after the final tuple, flush it via `writeBlock(-1, …)` and record the new partition block index.
    - **Join Phase**
        - For each partition *p* in [0…13]:
            - Skip if either $R_p$ or $S_p$ is empty.
            - **Build**: read each $R_p$ block into memory slot 0, insert tuples into a `Map<B,List<Tuple>>`.
            - **Probe**: read each $S_p$ block, for each tuple look up matching key in the map, and emit `(r.A, B, s.C)`.
    - I/O is counted by resetting `disk.resetIoCount()` before and reading `disk.getIoCount()` after.

2. **One-Pass Hash Join**
    - If R (or S) fits in memory (≤ 14 blocks), build the hash table on that relation directly, then probe with the other—saving the entire partition phase I/O.

**Experiments (Part 5)**

- **Scenario 5.1 (R sampled from S)**
    - R: 1000 tuples → ~125 blocks
    - Join result: 1129 tuples, I/O = 2274 ( **Note: This result may vary as data generation is random**)
    - Printed 20 random join tuples.

```
Sample of 20 join tuples:
  (199,42027,102)
  (354,21697,376)
  (522,19392,517)
  (244,10963,694)
  (771,38681,447)
  (180,46231,290)
  (348,49151,418)
  (564,47197,829)
  (98,33091,313)
  (75,19615,75)
  (749,23410,909)
  (485,33765,918)
  (302,14778,89)
  (7,47422,63)
  (279,28381,649)
```

```
(984,13767,135)
(841,22244,181)
(895,44390,300)
(751,19139,624)
(790,24856,241)
```

- **Scenario 5.2 (R random B in [20000,30000])**
    - R: 1200 tuples → ~150 blocks
    - Join result: 153 tuples, I/O = 2349 (**Note: This result may vary as data generation is random)**
    - Printed all join tuples (fewer results due to limited overlap).

      ```
      All 153 join tuples:
        (813,25760,537)
        (524,25326,194)
        (222,21742,202)
        (65,29218,528)
        (89,24668,537)
        (719,24668,537)
        (295,26390,366)
        (564,21211,917)
        (14,25299,262)
        (13,29793,987)
        (64,25957,737)
        (95,21141,117)
         … so on
      ```

## (d) Performance Discussion

- **Tree Height vs. Order & Fill**
    - Bulk-loaded ("dense") trees achieve maximum fan-out and minimal height, accelerating lookups and range scans at the expense of a heavier initial construction pass.
    - Incrementally built ("sparse") trees incur more node splits and deeper levels but allow faster online insertion without full resorting.

- **Join I/O vs. Partitioning**
    - Two-pass partitioning reads + writes each relation once (≈ 2× blocks) and then reads each partition pair again (≈ blocks), for a total ≈ 3× the sum of blocks.
    - One-pass join reduces I/O by eliminating the partitioning step for the smaller relation.

- **Data Distribution Effects**
    - Uniform random B-values yielded balanced partitions. Real-world skew might require adaptive or median-based partitioning to avoid "hot" buckets.

## **Reflections**

Implementing the B+ Tree turned out to be a true exercise in patience and precision. Deletion was especially tricky, I had to decide, for each underflow, whether to borrow a key from a neighbor or to merge two nodes, and even a single off-by-one pointer would break the whole structure. Early on, I built dozens of tiny trees (order 3–5 with under 50 keys) to flush out every corner case; once those small experiments were rock solid, scaling up to 10000 records went surprisingly smoothly, and search and range queries “just worked.” On the hash-join side, modeling an external disk and counting every block read or write drove home how expensive I/O can be, and I saw I could shave hundreds of operations off by partitioning in two passes. Scattering well-placed exceptions through the code also paid dividends when something went wrong, the stack trace pointed me right to the violated invariant. By the end, I came away with a much deeper appreciation for how physical layout, buffer management, and strict invariants shape real-world DBMS performance.