Database Algorithms Project Report

Cherukuri Venkata Surya Krishna - 136004790

Overview

In this project, we implement two core database‑management algorithms: B+ Trees for indexing and a hash‑based natural join. The B+ Tree component supports dynamic search, range queries, insertions, and deletions, and can be constructed either by bulk‑loading (dense) or by incremental insertion (sparse). The join component simulates an external‑memory environment—unlimited virtual disk, 15‑block main memory, 8 tuples per disk block—and realizes both the classic two‑pass hash join and an optimized one‑pass variant. Our goal was to explore how data organization and memory constraints influence algorithmic performance and behavior.

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 (≈ $\lceil(order-1)/2\rceil$ 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□ or S□ is empty.
       - **Build**: read each R□ block into memory slot 0, insert tuples into a `Map<B,List<Tuple>>`.
       - **Probe**: read each S□ 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.

<u>Reflections</u>

Tackling the B+ Tree implementation was challenging—deletion logic, in particular, required me to carefully choose between borrowing keys from a sibling or merging nodes, and a single misplaced pointer could corrupt the entire tree. I found that exhaustively testing tiny trees (order 3–5 with fewer than 50 keys) helped me iron out edge cases before confidently scaling up to 10,000 records. Once the core insertion and deletion routines were accurate, search and range queries fell into place seamlessly. On the join based on hashing, simulating an external disk with explicit I/O counters made me aware of how block‑level operations dominate performance, and implementing the two‑pass partitioning algorithm cut I/O by the hundreds. Finally, adding well‑placed exceptions throughout the code proved invaluable for debugging, pinpointing exactly where invariants were violated. Overall, I now better understand how physical data layout, buffer management, and algorithmic invariants drive real‑world DBMS performance.