COURSE PROJECT 2: Project Report

Luis Martinez Morales
GitHub Repo
Video

A. Project Overview

B+ Trees

The first part of this project is to design and implement a B+ Tree system capable of efficiently managing large datasets through structured storage and optimized operations. The system emphasizes generating unique records, constructing dense and sparse B+ Trees of varying orders, and rigorously testing core tree operations. Key features include:

- **Data generation:** Specifically, producing 10,000 unique integer keys within a specified range
- **Tree Construction:** building two types of B+ Trees
    - *Dense Tree:* B+ Tree whose non-root nodes are as full as possible
    - *Sparse Tree:* B+ Tree whose nodes are as sparse as possible
- **Core Operations:** Implementing search, range search, insertion, and deletion algorithms, with detailed logging to track node-level changes.

Unique Aspects

- **Dense vs. Sparse Construction**
    - Dense trees prioritize node fullness to reduce operational costs, while sparse trees simulate worst-case scenarios for underflow handling.
- **Order Flexibility:** Trees are built for orders 13 and 24, demonstrating how node capacity impacts performance and structure.
- **Operation logging:** Every insertion or deletion prints affected nodes before and after modifications, providing transparency into tree restructuring.

Data Generation

- A Python script (`generator.py`) generates 10,000 unique keys between 100,000 and 200,000. The dataset is sorted and stored in `btree_keys.txt`, ensuring reproducibility through a fixed random seed.

Challenges

- **Tree Integrity:** Making sure every node follows the rules for how many keys it must hold (minimum number of keys in sparse trees)
- **Split/Merge Handling:** Managing node splits during insertion. Similarly, deleting nodes is also an issue because the restructuring must be handled correctly.
- **Range Search Optimization:** Making the range search work efficiently involved moving through leaf nodes in order.
- **Performance Variability:** Testing showed different variables of trees allows there to be an analysis on the performance of B+ Trees.

Join based on Hashing

The goal of the second part of the project is to implement a two-pass hash join algorithm for natural joins between relations $R(A, B)$ and $S(B, C)$, while keeping strict memory constraints and simulating block-based disk I/O operations. The system focuses on:

1) **Data Generation:** Creating datasets for $R(A, B)$ and $S(B, C)$.
2) **Virtual Disk Simulation:** Modeling disk I/O operations (read/writes) to reflect limitations in the real world, where data must be loaded into a 15-block memory before processing.
3) **Hash-based Partitioning:** Using a modular hash function to distribute tuples into partitions for efficient join operations.
4) **Algorithm Validation:** Testing the implementation through two separate experiments with distinct data to analyze join results and disk I/O efficiency.

Key Components

- **Memory Constraints:** only 15 blocks (120 tuples) are available in the virtual main memory.
- **Block Size:** Each disk block holds 8 tuples.
- **Hash Function:** $hash_b(b) = b\ mod 14$

Unique Aspects

- **Virtual Disk Management:**
  - A Python dictionary (`disk`) simulates disk storage, tracking blocks for relations R, S, and their partitions.
  - Disk I/O counters (`disk_io_reads`, `disk_io_writes`) quantify algorithm performance.
- **Partitioning Strategy:**
  - Relations are split into 14 partitions during the first pass, ensuring each partition fits into memory during the join phase.
- **Experiment Design:**
  - **Experiment 5.1:** Test joins where R's B-values are guaranteed to match S's unique keys.
  - **Experiment 5.2:** Tests joins with R's B-values partially overlapping S's range, highlighting mismatches.

Data Generation

- Data is generated using Python's random module
  - $S(B, C)$: 5,000 unique B-values (10,000-50,000) with sequential C-values.
  - $R(A, B)$: 1,000-1,200 tuples with B-values either sampled from S (Experiment 5.1) or randomly generated (Experiment 5.2)
  - Similarly, a fixed seed is used to ensure reproducibility.

Challenges

- **Memory Efficiency:** Ensuring partitions fit within 14 blocks during the join phase required careful hash function tuning.
- **I/O Accounting:** Accurately tracking disk reads/writes to validate the theoretical $3 \times \left(B(R)\right) + B(S)$ I/O cost model.

- **Edge cases:** Handling scenarios where partitions are empty or tuples have no matches (e.g., Experiment 5.2)

B. B+ Trees Implementation

Part 1: Data Generation

The `generator.py` file has a function that generates 10,000 unique integer keys between 100,000 and 200,000. Specifically, the `generate_unique_records` function uses a set to ensure uniqueness and then writes sorted. Keys to `data/btree_keys.txt`. Then the `load_records` function reads the file into a list for tree construction. This directly tackles the requirement for Part 1 by creating a unique and sorted dataset.

Part 2: Building B+ Trees

The `builder.py` file has two different methods of creating b+ trees.

1. **Dense Trees** (`build_dense_tree`):
   a) Inserts all sorted keys into the tree
   b) Then nodes fill to the maximum capacity (order-1), triggering splits until all the nodes (Except the root) are full.
2. **Sparse Trees** (`build_sparse_tree`):
   a) Uses a `step` to skip keys, reducing insertion frequency.

Part 3: B+ Tree Operations

First, it is important to note that the tree is initialized (`BPlusTree` class) with a specified `order`. This dictates the maximum number of keys per node.

Key Properties:

- `min_keys_leaf = order // 2` : Minimum keys a leaf node must hold
- `max_keys_leaf = order - 1` : Maximum keys a leaf node can hold
- `min_keys_internal` and `max_keys_internal` : Similar rules for internal nodes

Root Node: Initialized as a leaf node (`self.root = BPlusTreeNode (is_leaf=True)`)

1. Search Operation

The `search` method locates a key in the tree and returns whether it exists, along with the number of steps taken.

- Traversal:
  (1) Start at the root node.
  (2) Traverse internal nodes by comparing the key with node keys to select the appropriate child pointer.
  (3) Terminate at a leaf node.
- Steps Counted: Each node traversal increments the step counter.
- Results: Returns `(found: bool, steps: int)`

2. **Range Search**

The `range_search` method retrieves all keys within a range `[start, end]` .

- Traversal:
    (1) Navigate to the leftmost leaf via `node.children[0]` .
    (2) Collect keys within the range while traversing leaves using `node.children[-1]` (assumes leaves are linked backward).

3. **Insertion**

The `insert` method adds a key to the tree while maintaining B+ Tree properties.

- Steps:
    1. Leaf insertion:
        a. Locate the target leaf node.
        b. Insert and sort the key.
        c. Split the leaf it if overflows (`max_keys_leaf` exceeded) .
    2. Leaf Split:
        a. Split point: `(order + 1) // 2` .
        b. The first key of the new leaf propagates to the parent.
    3. Internal Node Handling:
        a. If an internal node overflows, split at the middle key.
        b. Promote the middle key to the parent, updating child pointers.

Here is an example of what the insertion log should look like when inserting:

```
Inserting 123456
[Insert] Leaf node before insertion: [120000, 125000]
[Split] Leaf node overflow detected -> Split into [120000] and [123456, 125000]
[Insert] Created new root with key 123456
```

4. **Deletion**

The `delete` method removes a key and handles underflow.

- Steps:
    1. Leaf deletion:
        a. Remove the key from the leaf.
        b. Trigger underflow resolution if keys drop below `min_keys_leaf` .
    2. Underflow Resolution:
        a. Borrow: Take a key from a left/right sibling if possible.
        b. Merge: Combine with a sibling if borrowing is not feasible.
    3. Internal Node Updates:
        a. Adjust parent keys after borrowing or merging.

Here is an example of what the deletion log should look like when deleting:

```
Deleting 147106
[Delete] Leaf node underflow detected -> Borrow from right sibling
[Underflow] Parent node keys updated.
```

C. Join by Hashing Implementation

Part 1: Data Generation

The file hash_join.py generates datasets for relations $R(A, B)$ and $S(B, C)$ using three separate functions.

1. Generating Relations $S(B, C)$:
   - The purpose of this function was to create 5,000 unique tuples where B is a key attribute.
   - Steps:
     - B-values: The values are randomly sampled using the range 10,000-50,000 using `random.sample` to ensure uniqueness.
     - C-values: Sequential strings `(C0, C1, …, C4999)` are generated using enumeration.
     - Blocking: Tuples are grouped into blocks of 8 which then results in $\lceil 5{,}000/8 \rceil = 625\ blocks$
   - Validation: There is an additional assertion check that checks for duplicate B-values so that the key constraint is enforced.
2. Generating Relation $R(A, B)$ specifically for Experiment 5.1:
   - The purpose of this function was to create 1,000 tuples where B-values are sampled from S and in which duplicates are allowed.
   - Steps:
     - B-values: The values are selected using `random.sample` from S's B-values.
     - A-values: Sequential strings `(A0, A1, …, A999)` are also generated using enumeration.
     - Blocking: Tuples are split: $\lceil 1{,}000/8 \rceil = 125\ blocks$
3. Generating Relation $R(A, B)$specifically for Experiment 5.2:
   - The purpose of this function was to create 1,200 tuples with B-values between 20,000 and 30,000, independent of S
   - Steps:
     - B-values: Random integers in the range 20,000-30,000 are generated with `random.randint` .
     - A-values: Sequential strings `(A0, A1, …, A1199)` are also generated using enumeration.
     - Blocking: Tuples are split: $\lceil 1{,}200/8 \rceil = 150\ blocks$

Part 2: Virtual Disk I/O

The virtual disk and main memory are simulated using Python structures. They enforce block based I/O operations.

1. Disk Structure:

- Storage: A global dictionary `disk` stores relations (for example: `disk['S']`) as lists of blocks.
- Block Size: Each block holds 8 tuples (e.g., `S_tuples[i:i+8]`).

2. I/O Operations:

- Reading:
    - `read_block(relation, block_num):`
        - First it increments `disk_io_reads` so that every read is properly tracked.
        - Then it returns the specified block from the disk.
        - This simulates reading from the virtual disk to the virtual main memory.
- Writing:
    - `write_block(relation, block):`
        - First it increments disk_io_writes so that every write is properly tracked.
        - Then it appends the block to the relation in `disk` .
        - This simulates writing from the virtual main memory to the virtual disk.

Part 3: Hash Function

The hash function is used to map the B-values of the relations to a proper range in the join algorithm.

- The formula I ended up picking was $hash_b(b) = b\ mod 14$. The reason being that with 15 total memory blocks, 1 block could be reserved for processing leaves and the other 14 blocks could be used to store partitions.
    - Additionally using the mod operations would distribute B-values somewhat evenly across 14 partitions.

Part 4: Join Algorithm

The two-pass hash join algorithm consists of partitioning and join phases:

Phase 1: Partitioning:

- The goal was to split R and S into 14 partitions using `hash_b` .
- `partition_relation(relation_name):`
    1. Initialize Buffers: 14 empty buffers (one per partition).
    2. Read Blocks: Load each block of the input relation (for example: `read_block('R', block_num)`)
    3. Hash Tuples: For each tuple, compute its partition using `hash_b`, and then add it to the corresponding buffer.
    4. Write Buffers: When a buffer reaches 8 tuples, write it to disk as a partition block (example use case: `write_block('R_part_0', buffer)`).
    5. Flush Remaining: Write partially filled buffers to disk after processing all blocks.

Phase 2: Join

- The goal is to join matching partitions of R and S.
- `Two_pass_join()`:
    1. Load R Partition: For partition i, read all blocks of `R_part_i` into memory and build a hash table keyed by B-values.
    2. Check the S Partition: Then I read blocks of `S_part_i` , and for each tuple, I check for matching B-values in R's hash table.
    3. Output Matches: Finally, for every match, a joined tuple is created (for example: `(A, B, C)`) and it is then added to `join_result` .

Memory management:

- When splitting the data into partitions, each partition is kept small enough to stay within 14 memory blocks. For instance, if relation R has 125 blocks total, dividing it into 14 partitions means each partition will hold about 9 blocks. This prevents memory overload.
- During the join phase, the algorithm processes the S partition by reading one block of it at a time into memory. At the same time, the hash table built from the corresponding R partition is kept in the remaining 14 memory blocks.

D. Algorithm Performance

B+ Trees

The performance of B+ Tree operations depends heavily on how the tree is structured (dense vs. sparse) and the order of the tree (13 vs. 24). For the upcoming experiments, I primarily looked at random search results, but I also created a test case and its own log file for testing random **range search** results.

The Impact of Tree Density

- Search Efficiency
    - Successful Searches:
        - Dense13: For all 5 attempts at searching for random keys, it took 5 steps to find the keys.
        - Dense24: Similarly, for Dense24 all 5 attempts at searching for a random key took 4 steps.
        - *The reason behind this result is most likely because dense trees have shorter heights (due to tightly packed nodes). For the higher order tree, it reduced height even more and probably is the cause for fewer step counts.*
    - Unsuccessful Searches:
        - Sparse13: For all 5 attempts at searching for random keys, it was unsuccessful, and it took 3 steps to come to this conclusion.
        - Sparse24: Similarly, for Sparse23 all 5 attempts at searching for a random key were unsuccessful but this time it only took 2 steps to come to this conclusion.
        - *The reason behind this result is because sparse trees terminate searches earlier when a key is not detected in underfilled nodes.*

While dense trees guarantee full data coverage and enable reliable search and range operations, the sparse trees terminated their search faster. The sparse tree search terminated faster and was unsuccessful because it was missing valid keys in range queries.

The Impact of Tree Order (13 vs. 24)

The tree's order directly affects the tree height and operation cost:

- Search steps
  - The Dense24 tree required 4 steps for successful searches when compared to the 5 steps it took for the Dense 13 tree.
  - The Sparse24 tree required 2 steps for unsuccessful searches when compared to the 3 steps it took for the Sparse13 tree.
  - *The reason behind this is that higher-order trees store more keys per node, reducing the tree heigh and traversal steps.*
- **Range Search Efficiency**
  - Both dense trees (13 and 14) found keys in the range, but Dense24 traversed fewer nodes because of its shorter height.
  - Sparse trees' inability to return range results stems from skipped keys during construction, not order.

Overall, higher-order trees optimize search performance by minimizing tree height. Order has no substantial impact on sparse trees' **range search** results if keys are excluded during their creation.

Search vs Range Search Trade-offs

This section will incorporate the results from random range search in `rangeSearch.log`.

- Search:
  - For dense trees it has consistent performance for successful searches but higher step counts.
  - For sparse trees it has faster termination for missing keys but ultimately it is unreliable for finding existing keys.
- Range Search:
  - Only dense trees are viable for range queries due to complete key coverage.

It is important to note that when I initially tested range search with hand selected ranges. These ranges were closer to the minimum (100,000) and had a small length (around 50 or 100). When I tried to search for these ranges in the dense trees it would always show up. But whenever I tried to search for this range in the sparse trees, they were always unsuccessful.

- Meaning that dense trees are great for range searching since they are more complete, and sparse trees can give a result if the range is large enough. If the search range is small, then sparse trees are useless at finding values. Like normal search, sparse trees are not good at finding values but at least terminate earlier.

Conclusion

These experiments highlight three critical dependencies:

1. **Tree density**

   a) Dense trees ensure reliable search/range search results but have higher insertion costs.
   b) Sparse trees optimize for unsuccessful searches but fail in range queries.
2. **Tree order**
   a) Higher orders reduce tree height and improve search efficiency.
3. **Data organization**
   a) Sparse trees exclude keys during construction, limiting their usefulness for exact searches and range searches.

Join based on Hashing

The performance of the two-pass hash join algorithm depends heavily on data distribution and overlap between relations, as demonstrated by the experiments. Below is a detailed analysis using the provided results.

Impact of Data Overlap

Experiment 5.1 (All B-values in R Exist in S):

- Joined Tuples: 1,000 (100% of R's tuples matched S).
- Reason: R's B-values were explicitly sampled from S, ensuring full overlap.
- Disk I/O:
  - Total I/O: 2,270 (this is close to the theoretical $3 \times (125 + 625) = 2{,}250$)
  - Efficiency: All disk reads/writes contributed to valid matches, minimizing wasted I/O.

Experiment 5.2 (Partial B-value Overlap):

- Joined Tuples: 161
- Reason: R's B-values (20,000-30,000) partially overlapped with S's broader range (10,000-50,000).
- Disk I/O:
  - Total I/O: 2347 (this is slightly higher than the theoretical $3 \times (150 + 625) = 2{,}325$).
  - Inefficiency: Non-matching partitions (e.g., R's B-values outside S's range) were still read/written, increasing I/O without contributing to results.

Impact of Data Distribution

Uniform vs. Clustered B-values:

Experiment 5.1 Observations:

- Uniform Distribution: S's B-values (10,000-50,000) and R's sampled values were evenly spread.
- Balanced Partitions: The hash function $b\ mod 14$ distributed tuples evenly across partitions.
- Efficient Join: Uniform partitions ensured memory could handle entire R-partitions during the build phase.

Experiment 5.2 Observations:

- Clustered B-values: R's B-values (20,000-30,000) were concentrated in a narrower range.
- Skewed Partitions:
  - For example, B-values like 16130 (appeared twice in R, this is in the log for Experiment 5.1) are mapped to the same partition, increasing its size.
  - Uneven Workload: Some partitions (e.g., these covering 20,000-30,000) required more I/O due to higher tuple density.
  - The result being slightly higher I/O (2347 vs. 2325) despite fewer matches.

Hash Function Effectiveness

The choice of $hash_b(b) = b\ mod 14$ was critical to performance:

- Memory Alignment: With 15 memory blocks, reserving 1 for streaming S-partitions allows 14 blocks for R-partitions.
    - Example: R-partitions in Experiment 5.1 (125 blocks) around 9 blocks/partition, fitting comfortably into 14 memory blocks.
- Skew Mitigation:
    - **Experiment 5.1:** Modulo 14 distributed 5,000 unique B-values evenly (e.g., 357-358 tuples/partition).
    - **Experiment 5.2:** Clustered B-values led to uneven distribution but still avoided memory overflow.

Block Utilization

Full Blocks vs. Partial Blocks:

- **Experiment 5.1:**
    - R's 1,000 tuples formed 125 full blocks (8 tuples/block).
    - Minimal partial blocks → Efficient writes.
- **Experiment 5.2:**
    - R's 1,200 tuples 150 full blocks.
    - No partial blocks → Similar efficiency.

Both experiments achieved near-theoretical I/O counts, but partial blocks in partitioning (such as final buffer flushes) caused some minor deviations/errors.

Conclusion

In conclusion, the algorithm's performance depends a lot on the type of data being used. Some key takeaways are:

- Full overlap maximizes matches, and partial overlap wastes I/O.
- Uniform data ensure efficiency and in contrast skewed data increases variability.
- The hash function that is used is critical, proper partitioning aligns with memory limits and minimizes skew.

These experiments validate the two-pass hash join's I/O cost follows the $3 \times (B(R) + B(S))$ model, but real-world efficiency entirely hinges on the data characteristics.

D. Discussion

Implementing the B+ Tree and hash-based join algorithms were both challenging and enlightening. While I had prior exposure to database concepts such as hashing, diving deeper into the details revealed that there was still stuff I had to learn. It required both careful problem-solving and iterative testing.

The most time-consuming aspect of the B+ implementation was ensuring that node splits and merges behaved correctly during insertions and deletions. For instance, the `_split_leaf` and `_split_internal` helper functions initially caused confusion due to how split indices were calculated. The formula `split_index = (self.order + 1) //2` for leaves versus `split_index = len(node.keys) // 2` for internal nodes led to some errors and it was hard to identify it at first. At one point, lead nodes split unevenly, which left internal nodes with mismatches keys. Debugging this required extensive print statements to track node states before and after the respective operations.

Another hurdle I encountered with the B+ Tree implementation was the deletions. The _delete method's logic for underflow often left the tree in an inconsistent state. For example, when merging a leaf node with its sibling, the parent's keys would not always end up correctly updating. This would cause the validate function to flag the errors and required me to re-evaluate my algorithm and how nodes would behave during merges, especially for internal nodes.

Something I learned quickly with the B+ Tree implementation was the importance of incrementally testing. When I tried making the B+ Tree with 10,000 keys it immediately filled up the terminal with print statements. This made it hard to see what was going on and even with an output text file it was hard to traverse through the print statements. This led me to making separate log files for each part of the experiments (i.e., Part 4c1, part 4c2, etc.…) to properly address each mistake. I also ended up using a lot of print statements to properly address specific errors like node overflow.

For the two-pass join algorithm, it initially was very costly in terms of disk I/O operations. The `partition_relation` function wrote partially filled blocks to the disk unnecessarily and resulted in excessive `disk_io_writes`. Another issue I faced had to do with tuple alignment during the join phase. This led to mismatches when trying to access the correct indices. Once again, I fixed a lot of issues using print statements. I ultimately chose to also log the output onto separate log files for clarity.

Overall, this project deepened my understanding of B+ Trees and distributed join algorithms. I learned to have patience and use debugging techniques to fix small errors. More importantly, I learned about the importance of tree structure and data from implementing these algorithms. There are tradeoffs that occur from different order B+ Trees and hashing is important for distributing data evenly.