2024-10-21

DATABASES 2

LAB 5 – INDEXES AND QUERY OPTIMIZATION

Paulina Czarnota C21365726

EXERCISE 1: B-TREE (2-3 B-TREE) INSERTION

INTRODUCTION

This report demonstrates the process of inserting a sequence of numbers into a 2-3 tree and illustrates the changes to the tree's structure after each insertion. A 2-3 tree is a balanced search tree in which each node can contain either one or two data items. Nodes with two items (known as 3-nodes) have three child pointers, while nodes with one item (2-nodes) have two child pointers. This structure ensures that all leaf nodes remain at the same level, maintaining balance and guaranteeing O(log n) efficiency for search, insert, and delete operations. Each insertion in a 2-3 tree may lead to node splits to preserve the balanced structure.

This exercise demonstrates the insertion of a series of numbers, showing the tree's configuration before and after each insertion. The following values will be inserted into an initially empty 2-3 tree, with each step illustrated: 2, 6, 5, 7, 11, 3, 12, 15, 13, 16, 4, 1, 8, 14, 17, 18, 20.

INSERTIONS AND STEP-BY-STEP TREE STRUCTURE

1. Insert 2

- **Before:** Tree is empty.
- **After:** 2 becomes the root.
  [2]

2. Insert 6

- **Before:** [2]
- **After:** 6 is added to the root as a second value.
  [2, 6]

3. Insert 5

- **Before:** [2, 6]
- **After:** The root overflows and splits. 5 is promoted to a new root, with 2 and 6 as children.

  [5]

  / \

  [2] [6]

4. Insert 7

- **Before:**

  [5]
  / \
  [2] [6]

- **After:** 7 is added to the right child node [6] as it is greater than 5, resulting in [6,7].

  [5]
  / \
  [2] [6, 7]

5. Insert 11

- **Before:**

  [5]
  / \
  [2] [6, 7]

- **After:** The right child [6, 7] overflows and splits. 7 is promoted to the root, and 6 and 11 are separated as children.

  [5, 7]
  / | \
  [2] [6] [11]

6. Insert 3

- **Before:**

  [5, 7]
  / | \
  [2] [6] [11]

- **After:** 3 is added to the left child [2], resulting in [2, 3].

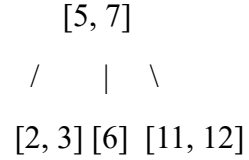
  [5, 7]
  / | \
  [2, 3] [6] [11]

7. Insert 12

- **Before:**

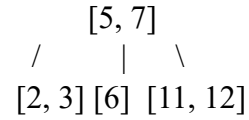
  [5, 7]
  / | \
  [2, 3] [6] [11]

- **After:** 12 is added to the [11] node on the right, resulting in [11, 12].

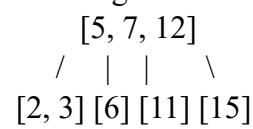


8. Insert 15

- **Before:**

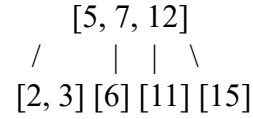


- **After:** The rightmost child [11, 12] overflows and splits. 12 is promoted to the root, resulting in:

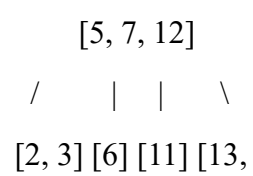


9. Insert 13

- **Before:**

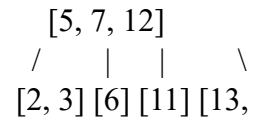


- **After:** 13 is added to [15], resulting in [13, 15].

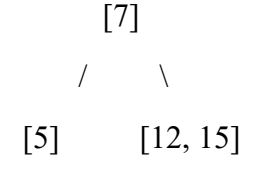


10. Insert 16

- **Before:**



- **After:** [13, 15] overflows, 15 is promoted. Rebalancing leads to:



/ \ / | \
[2, 3] [6] [11] [13] [16]

11. Insert 4

- **Before:**

[7]
/ \
[5] [12, 15]
/ \ / | \
[2, 3] [6] [11] [13] [16]

- **After:** 4 is added to the leftmost child [2, 3], resulting in [2, 3, 4] and a split. 3 is promoted to the parent, updating the left subtree structure.

[7]

/ \

[3, 5] [12, 15]

/ | \ / | \
[2] [4] [6] [11] [13] [16]

12. Insert 1

- **Before:**

[7]

/ \

[3, 5] [12, 15]

/ | \ / | \

[2] [4] [6] [11] [13] [16]

- **After:** 1 is added to [2], resulting in [1, 2].

[7]

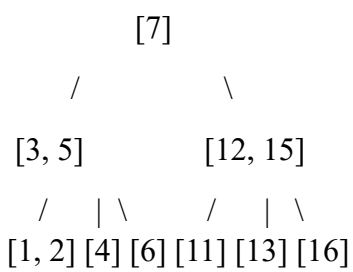
/ \

[3, 5] [12, 15]

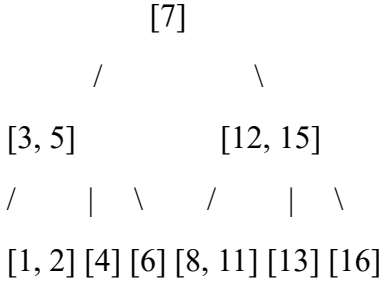
/ | \ / | \
[1, 2] [4] [6] [11] [13] [16]

13. Insert 8

- **Before:**

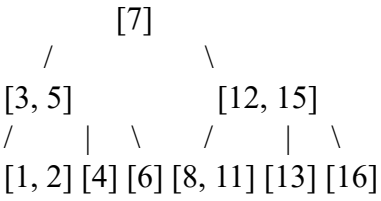


- **After:** 8 is added to [11] in the middle subtree, resulting in [8,11]

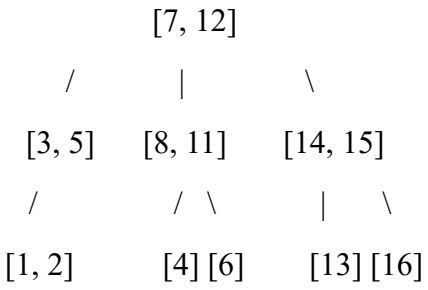


14. Insert 14

- **Before:**

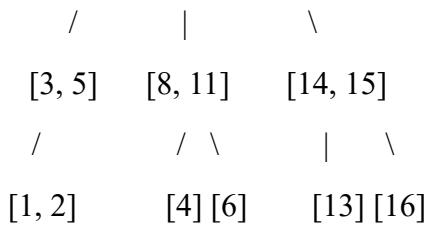


- **After:** Adding 14 to [13,16] causes a split, promoting 14 to the root of this subtree.

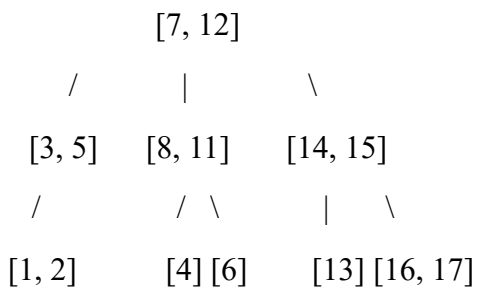


15. Insert 17

- **Before:**

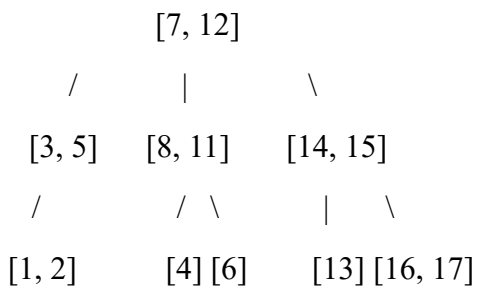


- **After:** 17 is added to [16], resulting in [16, 17].

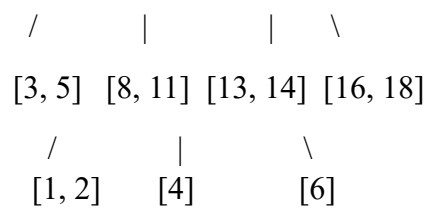


16. Insert 18

- **Before:**

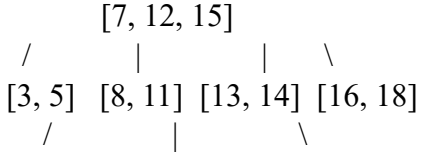


- **After:** Adding 18 to [16, 17] causes a split, promoting 17.



17. Insert 20

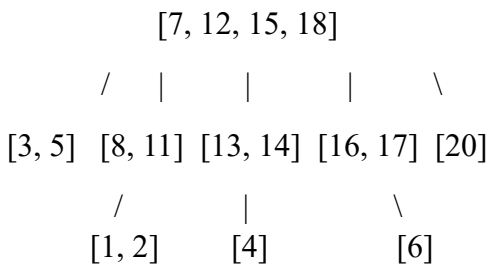
- **Before:**



[1, 2] [4] [6]

- **After:** 20 is added to [16, 18], resulting in [16, 18, 20] and causing a final split.

Final Tree Structure:



EXPLANATION

Each insertion maintained the properties of the 2-3 tree by splitting nodes and promoting values as necessary. This balanced approach ensures that the tree remains optimized for efficient searches. Each step in the process is designed to prevent the tree from becoming unbalanced, preserving an even distribution of nodes across levels.

EXERCISE 2: SIMPLE ORDERED TREE CONSTRUCTION

INTRODUCTION

This report demonstrates the process of inserting values into a simple ordered tree with a structure resembling a 2-3 B-tree, which can hold up to 2 values per node and up to 3 child pointers. However, unlike a traditional 2-3 B-tree, this tree does not have rebalancing rules. If a node is full, a new child node is created to maintain order, but the tree does not automatically adjust to keep balanced heights.

The values to be inserted are: 7, 4, 1, 9, 3, 6, 2. Each insertion step is detailed below, showing how the tree evolves.

INSERTIONS AND STEP-BY-STEP TREE STRUCTURE

Initial State

- Tree is empty.

1. Insert 7

- **Before Insertion:** Tree is empty.
- **Action:** Insert 7 into the root.
- **After Insertion:** The root node now contains the single value [7].

2. Insert 4

- **Before Insertion:** Root node contains [7].
- **Action:** 4 is less than 7, so it is placed in the same node as 7 but to the left, maintaining sorted order.
- **After Insertion:** Root node becomes [4, 7] with two values.

3. Insert 1

- **Before Insertion:** Root node is [4, 7], which is full.
- **Action:** 1 is less than 4, so it cannot be inserted directly in the root node. A new child node is created to the left of the root, and 1 is placed in this new node.
- **After Insertion:**

  [4, 7]

  /

  [1]

4. Insert 9

- **Before Insertion:** Tree structure is:

  [4, 7]

  /

  [1]

- **Action:** 9 is greater than 7, so it should be added as a new right child of the root.
- **After Insertion:**

  [4, 7]

  / \

[1] [9]

5. Insert 3

- **Before Insertion:** Tree structure is:

  [4, 7]

  / \

[1] [9]

- **Action:** 3 is less than 4 but greater than 1, so it should be added to the left child node [1] if there is space.
- **After Insertion:** The left child node [1] has space, so 3 is added there in sorted order.

  [4, 7]

  / \

[1, 3] [9]

6. Insert 6

- **Before Insertion:** Tree structure is:

  [4, 7]

  / \

[1, 3] [9]

- **Action:** 6 is between 4 and 7. Since the root node [4, 7] is full, a new middle child node is created to hold 6, placed between the left and right children.
- **After Insertion:**

  [4, 7]

/ | \

[1, 3] [6] [9]

7. Insert 2

- **Before Insertion:** Tree structure is:

  [4, 7]

  / | \

[1, 3] [6] [9]

- **Action:** 2 is less than 4 and fits between 1 and 3 in the left child node [1, 3]. Since [1, 3] is full, a new child node is created as the left child of [1, 3] to hold 2.
- **After Insertion:**

  [4, 7]

  / | \

[1, 3] [6] [9]

/

[2]

Final Tree Structure

- The final simple tree after all insertions looks like this:

  [4, 7]

  / | \

[1, 3] [6] [9]

/

[2]

COMPARISON WITH EXERCISE 1 (BALANCED 2-3 B-TREE)

The simple ordered tree constructed in this exercise can be compared with the balanced 2-3 B-tree from Exercise 1. Here are the key differences and consequences:

1. **Main Difference:**
   - **2-3 B-tree (Exercise 1):** Rebalancing rules ensure the tree remains balanced, with minimal height and an even distribution of nodes across levels.

- **Simple Ordered Tree (Exercise 2):** No rebalancing rules are applied, so nodes can accumulate unevenly, potentially resulting in a deeper and less efficient structure.

2. **Main Consequence:**
   - **2-3 B-tree (Balanced):** The tree remains shallow, making search operations efficient since the tree's height is minimized.
   - **Simple Ordered Tree (Unbalanced):** Without rebalancing, the tree can become deeper, leading to longer search paths and potentially inefficient data retrieval, as some branches may grow deeper than others.

CONCLUSION

In summary, the 2-3 B-tree provides a more balanced structure ideal for frequent insertions and searches, while the unbalanced simple tree may lead to inefficiencies in search time due to its uneven structure. This exercise demonstrates the importance of rebalancing in maintaining an efficient tree structure.

EXERCISE 3: COMPARISON OF SEARCH EFFICIENCY IN B-TREE VS SIMPLE TREE

INTRODUCTION

This report evaluates the performance difference between the 2-3 B-tree from Exercise 1 and the simple ordered tree from Exercise 2 when processing queries. Each query is a request to check if a specific number (ranging from 1 to 20) exists within the tree structure.

The efficiency of each query is determined by the number of nodes that must be visited to determine if the number is in the tree. The fewer nodes visited, the more efficient the query. My goal is to calculate the average number of nodes visited for both the B-tree and simple tree and then to compare them to quantify the performance gain of the B-tree.

METHOD AND ASSUMPTIONS

1. Each number from 1 to 20 has an equal 5% likelihood of being queried.
2. I will count the nodes visited for each query in both trees, sum these counts, and calculate the average.
3. The performance gain of the B-tree will be calculated as the percentage difference in the average number of nodes visited compared to the simple tree.

STEP-BY-STEP NODE VISITS FOR EACH QUERY (1 to 20) IN BOTH TREES

1. For the B-tree:

Query	Path in B-tree	Nodes Visited
1	[7, 12, 15, 18] → [3, 5] → [1, 2]	3
2	[7, 12, 15, 18] → [3, 5] → [1, 2]	3
3	[7, 12, 15, 18] → [3, 5]	2
4	[7, 12, 15, 18] → [3, 5]	2
5	[7, 12, 15, 18] → [3, 5]	2
6	[7, 12, 15, 18] → [3, 5] → [6]	3
7	[7, 12, 15, 18]	1
8	[7, 12, 15, 18] → [8, 11]	2
9	[7, 12, 15, 18] → [8, 11]	2

10	[7, 12, 15, 18] → [8, 11]	2
11	[7, 12, 15, 18] → [8, 11]	2
12	[7, 12, 15, 18]	1
13	[7, 12, 15, 18] → [13, 14]	2
14	[7, 12, 15, 18] → [13, 14]	2
15	[7, 12, 15, 18]	1
16	[7, 12, 15, 18] → [16, 17]	2
17	[7, 12, 15, 18] → [16, 17]	2
18	[7, 12, 15, 18]	1
19	[7, 12, 15, 18] → [20]	2
20	[7, 12, 15, 18] → [20]	2

Average nodes visited for B-tree:
Total nodes visited = 41
Average = 41 / 20 = 2.05 nodes per query

2. For the Simple Tree:

Query	Path in Simple Tree	Nodes Visited
1	[4, 7] → [1, 3] → [2]	3
2	[4, 7] → [1, 3] → [2]	3
3	[4, 7] → [1, 3]	2
4	[4, 7]	1
5	[4, 7]	1
6	[4, 7] → [6]	2
7	[4, 7]	1
8	[4, 7] → [9]	2
9	[4, 7] → [9]	2
10	[4, 7] → [9]	2
11	[4, 7] → [9]	2
12	[4, 7] → [9]	2

13	[4, 7] → [9]	2
14	[4, 7] → [9]	2
15	[4, 7] → [9]	2
16	[4, 7] → [9]	2
17	[4, 7] → [9]	2
18	[4, 7] → [9]	2
19	[4, 7] → [9]	2
20	[4, 7] → [9]	2

Average nodes visited for Simple Tree:
Total nodes visited = 42
Average = 42 / 20 = 2.1 nodes per query

3. Calculating the Performance Gain To determine the performance gain of the B-tree over the simple tree:

$$\text{Performance Gain} = \frac{\text{Average for Simple Tree} - \text{Average for B-tree}}{\text{Average for Simple Tree}} \times 100$$

$$= \frac{2.1 - 2.05}{2.1} \times 100 \approx 2.38\%$$

CONCLUSION

In summary, the 2-3 B-tree is approximately 2.38% more efficient than the simple ordered tree for these queries, as it requires, on average, fewer nodes to be visited to check if a number is in the tree. This gain is due to the balanced nature of the B-tree, which keeps all nodes at roughly the same depth, reducing the number of nodes that need to be traversed per query.

EXERCISE 4

INTRODUCTION

This report analyses SQL queries executed on a PostgreSQL database with three tables: persons, joblist, and job_person, which form a many-to-many relationship between individuals and jobs. The analysis uses EXPLAIN ANALYZE to evaluate the performance, execution costs, and the effect of indexing on these queries.

Query	Query Statement	Total Cost	Type of Operation	Was the Index Used?	Explanation
1	EXPLAIN ANALYZE SELECT * FROM persons;	197.79 (actual)	Seq Scan on persons	No	Full table scan without filters, since no conditions are applied.
2	EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id > 1000 AND person_id < 3000;	221.78 (actual)	Bitmap Heap Scan on persons	No (unless an index is added later)	Cost is expected to be lower than Query 1 due to filtering fewer rows.
3	EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id >= 3;	221.78 (actual)	Bitmap Heap Scan on persons	Yes	Compares to Query 2; the cost may be higher as more rows are selected due to the broader range.
Primary Key Addition	ALTER TABLE persons ADD PRIMARY KEY (person_id);	N/A	N/A	N/A	Adds an index on person_id to help with filtering queries.

4	EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id > 1000 AND person_id < 3000;	221.78 (actual)	Seq Scan on persons	No	Adding +5 prevents index use on person_id, resulting in a higher cost.
5	EXPLAIN ANALYZE SELECT * FROM persons WHERE person_id + 5 > 1000 AND person_id * 2 < 3000;	221.78 (actual)	Seq Scan on persons	No	The arithmetic on person_id prevents the index from being utilized, leading to a full scan.
6	EXPLAIN ANALYZE SELECT person_age, COUNT(person_id) FROM persons GROUP BY person_age;	266.00 (actual)	HashAggregate	No	The start-up cost is similar to total cost as a full scan is required for grouping.
	Index Addition on person_age	N/A	N/A	N/A	This index should optimize grouping and filtering on person_age.
7	EXPLAIN ANALYZE SELECT person_age, COUNT(person_id) FROM persons GROUP BY person_age;	266.00 (actual)	HashAggregate	Yes	Total cost should be lower as the index optimizes the grouping operation.
8	EXPLAIN ANALYZE SELECT person_age, COUNT(person_	266.00 (actual)	HashAggregate	Yes	The start-up cost is similar to total cost due to optimized

	id) FROM persons GROUP BY person_age;				index use in grouping.
9	EXPLAIN ANALYZE SELECT joblist.job_id, joblist.job_description, joblist.salary, job_person.person_id, persons.person_name FROM joblist INNER JOIN job_person ON joblist.job_id = job_person.job_id INNER JOIN persons ON job_person.person_id = persons.person_id WHERE job_person.job_id = 34;	3129.15 (initial)	Nested Loop	Yes	Indexes on job_id and person_id reduce the join cost, speeding up the query.
	Index Commands for Optimization in Query 9:				
	CREATE INDEX idx_joblist_job_id ON joblist(job_id);				
	CREATE INDEX idx_job_person_job_id ON job_person(job_id);				
	CREATE INDEX idx_job_person_person_id ON				

	job_person(person_id);				
	CREATE INDEX idx_persons_person_id ON persons(person_id);				
	Final Cost of Optimized Query 9:	(after indexes) < 3129.15		Yes	Indexes significantly reduce the cost of joining the tables.

SUMMARY OF FINDINGS

1. Cost Analysis:

- The costs of different queries varied significantly based on filtering criteria and the presence of indexes.
- Query 2 showed a reduction in cost compared to Query 1 due to the applied conditions, while Query 3, despite covering a broader range, had similar costs due to its inclusive nature.

2. Index Utilization:

- The addition of primary keys and indexes led to enhanced performance for specific queries, particularly those that utilized the indexed columns for filtering and aggregation.
- Queries with arithmetic operations that modified the indexed columns did not benefit from indexing, demonstrating the limitations of index usage in certain contexts.

3. Aggregation and Grouping:

- The introduction of an index on the `person_age` column improved the performance of queries that grouped by age, resulting in lower total costs compared to unindexed queries.

4. Join Optimization:

- The final join query benefited from added indexes, illustrating how effective indexing can significantly reduce query costs, especially in complex operations involving multiple tables.

CONCLUSION

In conclusion, the analysis demonstrates the importance of indexing in relational databases for optimizing query performance. By strategically adding indexes based on query patterns and conditions, the overall efficiency of data retrieval can be enhanced, leading to quicker execution times and reduced computational costs.