**Part 1)**

**B+ Trees**

The code is a comprehensive project that explores the performance of B+ trees with different orders and densities under various operations. A B+ tree is a balanced tree data structure commonly used in databases and file systems for efficient search, insertion, and deletion operations.

The project consists of several functions that generate records, build B+ trees, perform operations on the trees, and conduct experiments to analyze their performance. The main steps in the project are as follows:

1. Generate a set of random records.

2. Build B+ trees with different orders (dense\_order and sparse\_order) and tree densities (dense and sparse).

3. Perform a series of operations (insertions, deletions, and searches) on the created B+ trees and print the tree structure after each operation.

Here's a detailed explanation of the primary functions:

1. `generate\_records(int num\_records, int min\_key, int max\_key)`: Generates a vector of random integers with `num\_records` size, and values ranging from `min\_key` to `max\_key`. It uses the Mersenne Twister pseudo-random number generator (`std::mt19937`) to generate the random keys.

2. `build\_dense\_tree(const std::vector<int>& records, int order)` and `build\_sparse\_tree(const std::vector<int>& records, int order)`: These functions create B+ trees using the given `records` and `order`. The `build\_dense\_tree` function inserts the records as they are, leading to a potentially unbalanced B+ tree. In contrast, the `build\_sparse\_tree` function inserts the records in a sorted manner, creating a more balanced B+ tree.

3. `perform\_operations(std::vector<BPlusTree\*>& trees, int min\_key, int max\_key, std::mt19937& gen)`: This function performs a series of operations on the given `trees`:

- Insertions: Inserts 2 random keys (values between `min\_key` and `max\_key`) and prints the tree structure after each insertion using the `print\_tree\_disp` function.

- Deletions: Removes 2 random keys and prints the tree structure after each deletion using the `print\_tree\_disp` function.

- Search and remove otherwise insert: Searches for 5 random keys, removes them if found, otherwise inserts them, and prints the tree structure after each operation using the `print\_tree\_disp` function.

- Search: Searches for 5 random keys and prints the tree structure after each search using the `print\_tree\_disp` function.

4. `perform\_experiments(int num\_records, int min\_key, int max\_key, int dense\_order, int sparse\_order)`: This function orchestrates the entire experiment by generating records, building B+ trees with different orders and densities, and performing operations on these trees. It first generates the random records and builds the dense and sparse trees with the given dense\_order and sparse\_order. Then, it performs operations on both sets of trees (dense and sparse) and prints the tree structures.

5. `main()`: The main function initializes the parameters for the experiment and calls `perform\_experiments()` to execute the experiment. It also demonstrates a simple example of a B+ tree with order 3 and a predefined set of records, printing the tree structure after inserting the records using the `print\_tree\_disp` function.

In summary, this code tests the functionality and efficiency of B+ trees with different orders and densities by performing a series of insertions, deletions, and searches while printing the tree structure after each operation. The experiment aims to help understand how B+ trees behave under different configurations and tree densities, which can be beneficial in optimizing the performance of databases and file systems that use B+ trees.

(a) Overview of the project:

The project aims to implement a B+ Tree data structure, which is an essential data structure used for searching, sequential access, insertions, and deletions in a sorted order. B+ Trees are particularly useful in database systems and file systems. The project involves generating random data, building dense and sparse B+ Trees, and testing the implemented B+ Tree operations.

(b) B+ Trees: Generating data, building dense and sparse B+ Trees, and implementing B+ Tree operations:

1. Generating data: The **generate\_records** function generates a specified number of records with random keys within a given range.
2. Building dense and sparse B+ Trees: The **build\_dense\_tree** and **build\_sparse\_tree** functions create two B+ Trees by inserting the generated records in different orders. The dense tree is created by inserting records in ascending order, while the sparse tree is created by inserting records in a random order.
3. Implementing B+ Tree operations: The code includes functions for various B+ Tree operations such as inserting, deleting, and searching keys in the tree. Additionally, the code manages the splitting and merging of nodes when necessary, ensuring that the tree remains balanced.

(c) Performance dependence on the dataset and organization:

The performance of the algorithms depends on the size and distribution of the dataset as well as the organization of the data in the tree. For instance, a dense B+ Tree may require more splits and merges during insertions and deletions compared to a sparse B+ Tree due to its structure. The distribution of keys in the dataset also affects the efficiency of search operations, as a more evenly distributed dataset will lead to better-balanced trees, resulting in faster search operations. Moreover, the order of the tree can also impact its performance, as larger orders result in wider nodes, reducing the height of the tree and potentially speeding up search operations.

(d) Discussion: Implementation experience, difficulties, and learning outcomes:

Implementing the B+ Tree data structure and its algorithms was a challenging task, especially in ensuring the tree remained balanced during insertions and deletions. Splitting and merging nodes while maintaining the correct pointers and key values required careful consideration and debugging. Overcoming these difficulties involved thoroughly understanding the underlying concepts of B+ Trees, reading documentation, and testing the code extensively.

The project provided a deeper understanding of the B+ Tree data structure and its algorithms. It highlighted the importance of maintaining a balanced tree for efficient search, insertion, and deletion operations. Implementing the algorithms and testing them on various datasets gave insights into the factors affecting their performance, such as tree organization, key distribution, and tree order. Overall, the project reinforced the value of B+ Trees in practical applications such as databases and file systems.

**Part 2)**

**Join based on Hashing**

1. Data Generation:

`generateRelationS(**int** size)` generates a relation S with a specified number of tuples where B is the key attribute, **and** C can be of any type. The values of attribute B are random integers between 10,000 **and** 50,000. The function takes the size of the relation as a parameter **and** returns the relation S in the form of a vector of tuples.

2. Virtual Disk I/O:

`readBlock(std::vector<Tuple>& memory, std::vector<Tuple>& disk, **int** blockNum)` reads a block from the **virtual** disk to the **virtual** main memory. It takes the main memory, **virtual** disk, **and** block number as arguments, **and** transfers the contents of the specified block from the disk to the memory.

`writeBlock(std::vector<Tuple>& memory, std::vector<Tuple>& disk)` writes a block from the **virtual** main memory to the **virtual** disk. It takes the main memory **and** **virtual** disk as arguments, **and** transfers the contents of the main memory to the disk, then clears the main memory.

3. Hash Function:

`hashFunction(**int** value)` maps the B-values of the relations to a proper range **for** the algorithm. It takes an integer value as an argument **and** returns its hash **using** the modulo **operator**.

4. Join Algorithm:

`twoPassJoin(std::vector<Tuple>& R, std::vector<Tuple>& S, **int**& diskIOs)` performs a natural join operation based on hashing. It takes relations R **and** S as input **and** counts the number of disk I/Os during the join operation. The function first checks **if** a one-pass join is possible, otherwise, it proceeds with a two-pass join.

- One-pass join: If the total number of tuples in R **and** S is less than **or** equal to the available tuples in the **virtual** main memory, a one-pass join is performed. The function builds hash tables **for** both relations R **and** S, **and** joins the tuples based on their B-values. The disk I/Os count **for** one-pass join is set to 0 since it does **not** require accessing the **virtual** disk.

- Two-pass join: If a one-pass join is **not** possible, the function proceeds with the two-pass join algorithm. The join operation is divided into two phases: Partitioning **and** Join.

- Partitioning: This phase divides both relations R **and** S into blocks **using** the hash function. It partitions the tuples based on their B-values **and** writes them to the **virtual** disk when the **virtual** main memory is full.

- Join: This phase reads the tuples from the **virtual** main memory **and** the **virtual** disk into hash tables. It then joins the corresponding tuples from R **and** S based on their B-values **and** stores the results in an output vector.

5. Experiment:

`generateRelationR(**int** size, **const** std::vector<Tuple>& S)` generates a relation R with a specified number of tuples, where the values of the attribute B are randomly picked from the relation S, **and** the attribute A can be of any type. It returns the generated relation R.

The main function performs three experiments:

- One-pass join example: Generates relations S\_small **and** R\_small with a total number of tuples that fit within the **virtual** main memory (120 tuples). This example will execute the one-pass join algorithm in the twoPassJoin function. It then prints the disk I/Os used **and** the resulting tuples in the join.

- 5.1: Generates a relation R **and** calculates its natural join with the relation S **using** the twoPassJoin function. It then prints the disk I/Os used **and** the tuples in the join with random B-values.

- 5.2: Generates another relation R with 1,200 tuples **and** calculates its natural join with the relation S **using** the twoPassJoin function. In **this** experiment, the values of the attribute B are randomly picked from integers between 20,000 **and** 30,000, but **not** necessarily from the B-values in the relation S. It then prints the disk I/Os used **and** all the tuples in the join R(A, B) ⋈ S(B, C).

In summary, the code generates relations R **and** S, simulates **virtual** disk I/Os, **and** performs one-pass **and** two-pass natural join operations **using** a hash-based approach. It also counts the disk I/Os used during the join operations **and** provides output **for** different experiments.

(a) An overview of the project:

This project is an implementation of a two-pass hash-based join algorithm in C++. The main goal of the project is to efficiently join two relations R(A, B) and S(B, C) using a hash-based join technique. The project includes functions to generate relations, simulate virtual disk I/O operations, select hash functions, and implement the join algorithm. Additionally, the project demonstrates a one-pass join when the input relations can fit in the available main memory.

(b) Join by Hashing:

1. Data Generation: The `generateRelationS` function generates relation S with a specified number of tuples, where each tuple has a random value for B and a random value for C. The `generateRelationR` function generates relation R with a specified number of tuples, where each tuple has a random value for A and a value for B that is taken from a randomly chosen tuple in relation S.

2. Virtual Disk I/O: The `readBlock` function reads a block of tuples from the disk to the main memory, and the `writeBlock` function writes a block of tuples from the main memory to the disk.

3. Hash Function: The `hashFunction` takes an integer value as input and returns the result of the modulo operation between the value and the number of memory blocks, which is used for partitioning and probing.

4. Join Algorithm: The `twoPassJoin` function takes relations R and S and a reference to an integer variable for counting disk I/O operations. The algorithm first checks if a one-pass join can be performed (i.e., if the total number of tuples in R and S can fit in main memory). If possible, a one-pass join is performed using hash tables. If not, a two-pass join is performed by partitioning the relations into memory blocks using the hash function and then joining the partitions in the second pass.

(c) Performance dependence on the given data set and its organization:

The performance of the join algorithm depends on the size of the relations, the distribution of the data in the relations, and the memory constraints. If the relations can fit in main memory, the join operation can be performed in a single pass, which is more efficient. In other cases, the two-pass join algorithm is performed, which requires more disk I/O operations, and the performance depends on the distribution of the data in the relations and the choice of the hash function.

(d) Discussion:

Implementing the two-pass join algorithm helped in understanding the importance of efficient data organization and the role of hashing in partitioning the data. One of the main difficulties encountered in the project was handling the virtual disk I/O operations and keeping track of disk I/O count, which required careful implementation. Another challenge was adapting the join algorithm to handle different data types for the C attribute in the tuples, which was resolved using C++ templates and type traits. Overall, the project provided a deeper understanding of hash-based join algorithms and the role of data organization in improving their performance.