**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.

**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.