1. Hello! We are Group One, and we'll be presenting our project on B+ trees.
2. The agenda for this presentation will be to cover tasks one to three, specifically the implementation, design, and results. We'll end the presentation with a demonstration of our code and the expected results.
3. Our project is implemented in C++. The installation and usage details are in the report. The scope of this presentation is to provide a summary of our work and showcase the implementation details with graphical illustrations.
4. In Task one, we designed the storage for the B+ tree. Our preliminary investigation involved using Python code to analyze the "games.txt" file. We found 99 rows with missing data and 26,505 valid rows for our dataset. We assumed the incomplete rows could be discarded.
5. The optimal definition for our record storage is shown in the table, with fields sorted in optimal order and data types chosen for minimal space consumption. Our implementation ensures that records are unspanned, fitting in a single block without additional pointers. We enforce a fixed format and length for records, as the dataset does not have variable length text data. Records are stored contiguously within a block. This design ensures simple and quick record retrieval, reducing overhead processing.
6. Here is the C++ structure definition for the record. To illustrate field selection, we'll use "points\_home," "assists\_home," and "rebounds\_home" as examples. These are stored as unsigned 16-bit integers. While current values in the dataset range from 6 to 168, we chose this data type to accommodate potential future NBA match scores that may exceed 255.
7. We also optimized the order of storing the fields. [UG] If we stored them in the order they appear in the dataset file, it would result in a less efficient allocation of 32 bytes, with additional padding due to the largest data type taking up 4 bytes.
8. Instead, we ordered the blocks to minimize padding, reducing the overall storage space to 28 bytes per record.
9. The results for Task one are as shown on the slide
10. Moving on to Task two, We focused on designing the insertion and search operations for our B+ tree. This is the design of our insertion structure. In Case one, the trivial case, there is sufficient space in the leaf node, and we simply add the key to be inserted.
11. In the second case, the leaf node is full, and we split it into two nodes, add the middle key to the parent node, and update the pointers.
12. In the third case, if the parent node is full, we split it similarly. This splitting may propagate up to the root, potentially increasing the tree height. This is the more complex insertion case.
13. Next in our design is duplicate handling. We observed duplicate key values in our dataset. To illustrate how a B+ tree could fail without duplicate handling, consider the insertion order of 9, 10, 10, and 11 with an N value of 4. This would cause a violation of the B+ tree properties if no duplicate handling is enforced. As shown in the slide, the key 10 is on the left side of the root, violating the B+ tree property that lower bounds are on the right of the key.
14. To handle duplicate keys, we considered creating a composite key or using overflow blocks.
15. The composite key method would require choosing another column as a unique identifier. We could add a column for the order of insertion, creating a unique key using "FG\_percentage\_home" and the insertion order. While the composite key method is simpler, we opted for the overflow block method, which is more robust and doesn't require altering the primary key.
16. In the overflow block design, the leaf pointer points to an overflow block instead of a single record, allowing blocks to be chained like a linked list. This allows one key to point to multiple duplicate records.
17. Our implementation includes a parameter "K" in the leaf node, allowing for K duplicate records to be stored within the leaf node block. This reduces the I/O needed to load an overflow block, as the first K overflow records have their pointer within the leaf node. The value of K is a trade-off and could be further optimized to reduce I/O timing.
18. We designed two search algorithms: one for a single key and one for a range query. The single key search starts from the root node and traverses to the leaf node, comparing the target key value to the nodes' to traverse down the tree.
19. The range query search is a simple modification to find the lower bound and progressively move to the next right pointer until the upper bound is found.
20. Now, let's move on to the results of Tasks 2 and 3. In Task 2, we experimented with different N values for the B+ tree. [UG] We illustrate cases where N is too small, too large, and optimal. N is too small when the index block size is underutilized, and N is too large when it could house all unique keys, which is impractical. The optimal N value is the maximum value where the index node contents fit into a block size of 4,096 bytes, which we assume is the block size setting of the system.
21. As we progressively increase N, the height of the tree changes. We found that an N value of 69 allows the tree to maintain a steady height of 2, requiring a maximum of two index node hits before querying the data.
22. In Task 3, we explored the impacts of non-sequential and sequential files, comparing their timings to a linear scan. As we increase N in the non-sequential case, the number of index node accesses decreases, as expected, due to the decrease in tree height. The number of leaf nodes traversed also decreases as each leaf node can house more keys and records. The B+ tree scan time decreases with fewer index hits.
23. Comparing the brute force linear scan time to the B+ tree scan time, even with a non-sequential file, the B+ tree outperforms the linear scan by at least one second.
24. In the sequential file case, the B+ tree search time reduces by about nine times due to the decrease in data blocks accessed. This contrasts with the 122 data blocks accessed initially compared to the current two data blocks.
25. These are the results for Task three on a non-sequential file basis. As we increase N, the total querying time generally decreases.
26. The index node accesses also decrease, reducing the I/O time spent on index node loading.
27. Now, we will move on to the code demo.
    1. "To begin, the program requires compilation using a C++ compiler with the -O3 optimization flag to ensure optimal execution. We will execute the compiled binary from the command line, specifying two key parameters: the 'N' value, which defines the number of keys within the B+ tree, and the input data file. We will utilize two datasets: 'games.txt,' representing unsorted data, and 'games\_sorted.txt,' which has been pre-sorted for comparative analysis."
    2. Performance Analysis: "We will execute the program using both datasets to demonstrate the performance differences. Notably, the B+ tree's query speed with the pre-sorted dataset will be compared to the unsorted dataset, highlighting the impact of sequential data access. To provide robust performance metrics, we will conduct 1000 search queries and report the average execution time. Additionally, we will observe and document the creation of data blocks during the execution process."
    3. Code Overview: "The core functionality of the B+ tree is implemented within the 'B\_plus\_tree.cpp' file. This file encapsulates the insertion and search operations at the tree structure level and manages the loading of index data from disk storage. The search range operator utilizes an iterator to traverse leaf nodes and retrieve the requested records. The 'node.cpp' file defines the node structure and implements the 'insertLeaf' and 'insertInternal' functions, which handle insertion operations specifically for leaf and internal nodes, respectively. This file also contains the logic for managing node overflow through the 'split leaf child' and 'split internal child' functions."
    4. Conclusion: "This demonstration provides a clear illustration of the B+ tree's operational efficiency and its responsiveness to data organization. The observed performance metrics underscore the advantages of utilizing pre-sorted data for optimized search operations within a B+ tree structure."
28. Thank you!