Data System : Project Phase 2

Vidhi Pareek (2020101102)
Hardik Gupta (2020101045)

Implementation of External Sorting -

External sorting was implemented using the merge-sort algorithm. It works as follows :-

- Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

The algorithm essentially consists of two phases -

- Sorting Phase - runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, and written back to disk as temporary sorted subfiles (or runs).
- Merging Phase - Sorted runs are combined in multiple merge passes. In each pass, there can be several merge steps. The degree of merging, or "d_M," tells us how many sorted subfiles can be merged in each step. During a merge step, we need one buffer block to hold a disk block from each of the sorted subfiles being merged. Additionally, we need one more buffer to store a disk block of the resulting merged file, which is larger and made by combining smaller sorted subfiles.

For example, To sort 900 megabytes of data using only 100 megabytes of RAM, the algorithm will work in following manner :

- Read 100 MB of data into main memory and sort it using a method like quicksort.
- Save the sorted data to disk.

- Repeat steps 1 and 2 until all the data is divided into 100 MB sorted chunks (in this case, there are 9 chunks, as 900MB / 100MB = 9).
- Read the first 10 MB (which is 100MB divided by 9 chunks + 1) from each sorted chunk into input buffers in main memory, and allocate the remaining 10 MB for an output buffer.
- Perform a 9-way merge by combining the data from the input buffers and storing the result in the output buffer. When the output buffer is full, write its contents to the final sorted file and clear it. As any of the 9 input buffers become empty, refill them with the next 10 MB from their associated 100 MB sorted chunk until no more data is available.

How did we implement it ?

Sort Phase :-

- For the sort phase, we iterated through all the pages of the table, sorted them internally and wrote them back. Std::sort was used for sorting the data of pages with customComparator.

```
  // Sort Phase
 for(int page_num=0; page_num < this->blockCount; ++page_num)
 {
      Page page = bufferManager.getPage(this->tableName, page_num);
      vector<vector<int>> data = page.getAllRows();

      std::sort(data.begin(), data.begin() + page.getRowCount(),
CustomComp);
      page.setAllRows(data);

      // write page back
      page.writePage();
 }
```

Merge Phase :-

Pseudo code for merge phase can be found in the following code snippet -

```
/*
    Merge Phase
*/
Initialize some variables:
- Set mergePhaseNum to 0
- Set k to BUFFER_SIZE - 1
- Set numElem to 1 (number of pages in each list)
- Calculate numList as the number of lists needed (round up to the nearest integer) to merge the data

While there's more than one list to merge:
    Create an array called 'arr' to hold the starting and ending page numbers for each list

    Divide the pages into lists for this merge phase:
    - For each list in the range from 0 to numList:
        - Add the start and end page numbers to the 'arr' array

    Create a result cursor to store the merged data, named like "MP{mergePhaseNum}"

    Perform a k-way merge in passes:
    - For each pass in the range from 0 to numList, with a step size of 'k' /*merging k list at a time */:
        - If it's the first merge phase (mergePhaseNum is 0):
            - Use kWayMerge to merge the lists, considering 'arr', 'passNum', 'k', 'this->tableName',
              'CustomComp', and store the result in the 'resultCursor'
        - Otherwise:
            - Create a name for the previous merge result cursor like "MP{mergePhaseNum-1}"
            - Use kWayMerge to merge the lists, considering 'arr', 'passNum', 'k',
              the previous result cursor's name, 'CustomComp', and store the result in the 'resultCursor'

    Increment the mergePhaseNum by 1
    Update numElem as k raised to the power of mergePhaseNum
    Calculate the new numList as the number of lists needed for the next phase

// Continue this process until there's only one list left to merge
```

The Merge Phases consist of a series of runs, with each run being referred to as a phase. In every phase, we merge k lists simultaneously to create larger sorted lists. To begin, we initialize our process with a number of lists equal to the number of blocks in the file. We continue merging these lists until we achieve a single, comprehensive sorted list.

During a particular phase run, we make use of an auxiliary array to keep track of the pages associated with a specific list. This array contains the starting and ending page numbers of the blocks belonging to that particular list.

Following this, we execute a merging run, where we take k lists at a time and carry out k-way merges on them. This involves combining these lists into a single, sorted output.

To facilitate the storage of the merged results in pages, we utilize an object of the Cursor class. This object helps us manage the data effectively during the merging process. We also adhere to a naming convention for the result pages. Each result page is named according to the pattern "MP{mergePhaseNum}_Page{PageNum}." This naming convention allows us to clearly identify and organize the pages generated at each step of the merging process.

Changes in Cursor :

A new constructor has been introduced for the Cursor class to facilitate efficient page iteration.

New Constructor Definition:

```
Cursor::Cursor(string &tableName, int x, int y, bool retrieveFlag)
```

The behavior of this constructor depends on the value of retrieveFlag, which can be either true or false.

- When retrieveFlag is set to true, the values of x and y represent the startPage and endPage indices of the list. This configuration is used for reading through pages and enables row-by-row iteration between the specified pages using the getNext method.

- Conversely, when retrieveFlag is set to false, x and y represent maxNumRows and the number of columns for the page. This setup is employed for writing data into pages, row by row, using the setNext method. If a page becomes full, it is written, and the process continues with the next page.

K-way Merge Function :-

```
function kWayMerge(lists, k, resultCursor):
Initialize individual cursors for each list:
for i from 0 to k:
    cursor[i] = Cursor(lists[i].startPage, lists[i].endPage, retrieveFlag=true)

while at least one list is not empty:
    minRow = INFINITY // Initialize a minimum row value

    for i from 0 to k:
        if cursor[i].hasMoreRows(): // Check if the current list is not empty
            currentRow = cursor[i].getNextRow() // Get the top row of the list

            if currentRow < minRow: // Compare the current row with the current minimum
                minRow = currentRow // Update the minimum if a smaller row is found

    resultCursor.setNext(minRow) // Store the minimum row in the result pages

resultCursor.writePages() // Write the merged sorted list into pages
```

In the kWayMerge function, the objective is to merge k lists simultaneously, creating a single sorted list. This merged sorted list is then written into pages using the resultCursor.

To manage the iteration through each list, individual cursors are generated for them, with designated start and end page indices for each page.

During the merging process, as long as at least one of the lists still contains data, the algorithm systematically retrieves the minimum row from the top of each list. The minimum among these rows is then stored in the resultPages using the setNext function of the Cursor. This way, the kWayMerge function efficiently combines multiple lists into a single, sorted result.

Implementation of Order By -

The implementation of "Order By" in our system is a direct application of External Sort designed for a specific column. To achieve this, we introduced a modification to the table's sorting function by adding a new parameter, resultTableName.

The significance of this modification is that, if the resultTableName differs from the name of the table being sorted, it ensures that the original table remains unaffected after the sorting operation. Instead, the sorted data is stored in a new table named resultTableName.csv.

As a result, when performing an "Order By" operation, we simply call the sorting function on the target table (tableName) and specify the name of the resulting sorted table (resultTableName.csv). This approach allows us to efficiently organize and sort data according to a specific column without altering the original dataset.

Implementation of Join -

The "Join" operation in our system leverages external sorting for its execution. To illustrate this, let's consider an example where we need to perform a join operation on two tables, namely `Table1` and `Table2`. Both of these tables are first sorted and stored in temporary tables, aptly named `sorted_Table1` and `sorted_Table2`. The order in which the tables are sorted depends on the type of binary operator used for the joining attributes.

When the binary operator is one of `==`, `>`, or `>=`, both tables are sorted in ascending order. Conversely, when the operator is either `<` or `<=`, we sort both tables in descending order. This choice of sorting order is crucial in ensuring a correct join operation.

To effectively handle this process, we create Cursors that facilitate the iteration through the sorted tables (`sorted_Table1` and `sorted_Table2`) and the result

table where the output of the join operation is stored. These Cursors help manage the data flow during the join operation, enabling the extraction of sorted data and the efficient storage of join results.

```
// Create cursors to iterate through these tables
   Cursor cursor1(table1->tableName, 0, table1->blockCount, true);
   Cursor cursor2(table2->tableName, 0, table2->blockCount, true);
   Cursor resultCursor(resultTable->tableName,
resultTable->maxRowsPerBlock, resultTable->columnCount, false);
```

Here's the algorithm for the "Join" operation, considering different binary operators:

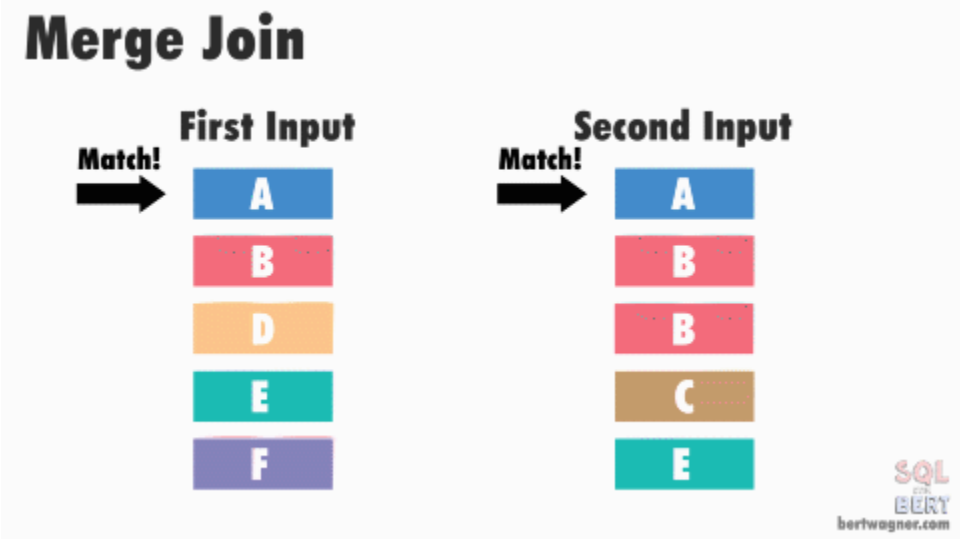
If the binary operator is `>`, `>=`, `<`, or `<=`:

1. Initialize record pointers `ptr1` and `ptr2`, pointing to the first record of `sortedTable1` and `sortedTable2`, respectively.
2. Increment `ptr2` until the records pointed to by `ptr1` and `ptr2` satisfy the given condition.
3. If the record pointed to by `ptr2` no longer satisfies the binary operation with the record pointed to by `ptr1`, all records in `sortedTable2` before `ptr2` are combined with the record pointed to by `ptr1` and stored in the `resultTable`.
4. Increment `ptr1` and repeat from step 2 until `ptr1` reaches the end of `table1`.

Else if the binary operator is `==`:

1. Initialize record pointers `ptr1` and `ptr2`, pointing to the first record of `sortedTable1` and `sortedTable2`, respectively.
2. If the records pointed to by `ptr1` and `ptr2` are not equal, increment the record pointer of the smaller record.
3. If the records are equal, store the location of `ptr2` (page index and pointer position in the page), and continue to increment `ptr2` while storing the records until they are equal to `record1` in the `resultTable`.

4. Increment `ptr1`. If this new record is equal to the previous record in `sortedTable1`, move `ptr2` back to the location stored earlier. Otherwise, repeat from step 2.



This algorithm provides a structured approach to the "Join" operation, ensuring that records from the sorted tables are appropriately combined and stored based on the specified binary operator.

Block Access -

Since both tables have been pre-sorted by us, our approach ensures that we iterate through the sorted tables just once, instead of repeatedly searching for matching records. This significant optimization drastically reduces block access, effectively eliminating the need for repetitive scans through the tables. As a result, the total block access for the join operation is reduced to the sum of the block access required during the initial sorting of table1 and table2, and the blocks necessary for storing the result.

Implementation of Group By -

To execute the GroupBy operation, our approach follows these key steps:

1. Initial Sorting: We begin by sorting the original table in ascending order based on the grouping attribute. This sorted version of the table is stored in a temporary CSV file called "sortedTable.csv."

2. Cursor Creation: We create Cursors to facilitate a smooth, row-by-row iteration through the "sortedTable." These Cursors also aid in storing rows in the result table.

3. Utilizing the Check Class: A specialized class named "Check" plays a pivotal role in managing the GroupBy operation. It's responsible for storing important statistics, such as `maxValue`, `minValue`, `avgValue`, and `Count`, for records with the same grouping attribute. Additionally, the Check class enables the comparison of `attribute1` against `attribute_value` using the specified binary operation (`bin_op`).

Algorithm for Group By:

The GroupBy operation is performed through the following steps:

1. Iterative Processing: We iterate through the "sortedTable" row by row, sequentially.

2. Updating Statistics: If the grouping attribute of the current row matches that of the previous row, we update the relevant statistics for both `attribute1` and `attribute2`. These statistics serve as critical criteria for evaluating whether records meet the specified conditions.

3. Attribute Change Check: When the grouping attribute of the current row differs from that of the previous row, we perform the following actions:
   - First, we assess whether the statistics associated with the previous grouping attribute, particularly those related to `attribute1`, satisfy the prescribed conditions. If they do, we store the value of the grouping attribute and other relevant columns, including statistics for `attribute2`, in the result table.

- Subsequently, we reset the statistics for both `attribute1` and `attribute2`.
- This process is then repeated, starting from step 2, to continue examining records as they transition from one grouping attribute to the next.

Block Access :-

Total Block Access = Block Access for Initial Sorting + Block Count of the Table (Iteratively Going Row by Row) + Block Count for Storing Results

In mathematical notation:

Total Block Access (B_{total}) = $B_{sorting} + B_{table\_iteration} + B_{storing\_results}$

Where:

$B_{sorting}$ represents the block access required for the initial sorting of the table.
$B_{table\_iteration}$ represents the block count associated with iteratively processing the table row by row.
$B_{storing\_results}$ represents the block count required for storing the results of the GroupBy operation.

Learnings :-

We learned several important lessons through the implementation of External Sort, as well as Join, Order By, and Group By operations:

1. Effective Data Management: Our project deepened our ability to manage substantial datasets efficiently, with a particular focus on the External Sort implementation. We implemented external sorting techniques to handle data that exceeds available memory capacity, a skill with practical applications in data-intensive industries.

2. Algorithm Optimization: The project emphasized the significance of algorithm optimization in data processing. We recognized the importance of minimizing disk I/O operations, which can be time-consuming, especially when working with large datasets.

3. Join Operation Efficiency: Sorting tables before joining proved to be a game-changer in terms of query performance. By reducing the number of block accesses, it significantly improved the speed and efficiency of query execution.

4. Group By Logic: Implementing Group By operations deepened our understanding of grouping and aggregation. We developed a systematic approach for managing statistics and making conditional evaluations, which is crucial for summarizing and analyzing extensive datasets.

Contribution

Hardik and I collaborated equally throughout the project. We discussed implementation logic and divided our responsibilities.

Hardik implemented the External Sort and Order By operations, while I focused on the Join and GroupBy operations.

We both conducted extensive testing to ensure the project's robustness and functionality.