IIT DELHI

Assignment report for COL362(DBMS)

External K-way Merge Sorting

Prof. Srikanta Bedathur

Prakhar Jagwani
2019CS10382
Souvagya Ranjan Sethi
2019CS10405

Department of Computer Science & Engineering

April 15, 2023

1 Implementation

1.1 Classes Defined

- **Reader**
  We have defined a `Reader` class for reading the lines in the input. The Reader constructor takes two arguments: a *filename* and a *buffer size*. We set the buffer size of `ifstream`, which can improve the performance of file reading. The Reader class provides a convenient and efficient way to read files line by line, and check for the end of the file.

  The ¬Reader() function is the class destructor, which deletes the buffer and closes the file stream.

  We also tried to manage the buffer manually, using `fopen`, `fread`, etc. calls, but using the `ifstream` implementation remained faster.

- **Writer**
  The Writer class is responsible for writing strings to a file. It takes a filename and buffer size as input parameters. It contains a buffer to temporarily store the strings before writing to the file, and a buffer_pos variable to keep track of the current position in the buffer.

  The writeWord function takes a string as input, and writes it to the buffer. If the buffer is full, the function flushes the buffer to the file and resets buffer_pos. The function also adds a newline character to the end of the string to separate each string in the file.

  The destructor of the class ¬Writer() flushes the remaining buffer to the file, and deletes the buffer and closes the file.

- **TrieNode**
  The class TrieNode represents a node in a trie data structure. Each node contains a pointer to its next sibling, its first child, and its parent, as well as the character associated with the node and a count of the number of times a word ends at this node. The size of a TrieNode object turns out to be 32 bytes. We can compress this a little by using 32 bit integers to index an array instead of using pointers. But we didn't do it because we didn't expect much improvement over using the classical sort.

  The getWord function returns the word represented by the node by traversing the parent pointers, starting from the current node, and storing each character in a buffer until the root node is reached. The buffer is then reversed and converted to a string, which is returned.

  The destructor recursively deletes the next and child nodes, if they exist, to avoid memory leaks.

- **Trie**
  The Trie data structure is used to store a large number of strings and sort them efficienctly. Each node in the Trie is a TrieNode structure defined above. The Trie class contains a root pointer to the root TrieNode (which represents an empty string), as well as a size variable to keep track of the memory in use by the Trie as a whole.

  The Trie class has several member functions. insert function takes a string as input and inserts each character of the string into the Trie, creating new TrieNodes as needed. Each node in the Trie represents a unique string, and its count is incremented every time a string is inserted into the Trie.

The firstNode and nextNode functions are used to traverse the Trie in a pre-order traversal. write_to_file function writes each unique string in the Trie to a file, in lexicographic order.

When the size of the Trie exceeds a memory limit, it writes the Trie to a file and clears it.

1.2 Functions Defined

- **sort_in_runs(const string &input_file_name, int key_count)**
  - ⋆ **Description**: This function reads the input file and splits the contents into smaller chunks (runs), each containing a sorted subset of the data. The functions uses a vector to store the data. The `MEM_LIMIT_MB` constant limits the amount of memory that can be used during the sorting process. The function returns the number of runs created.
  - ⋆ **Key Points**:
    (a) Initially, we reserve the memory for the vector so that subsequent insert into the vector doesn't require reallocation of memory every time.
    Here, we assume that the avg_size of the key to be `MAX_KEY_LENGTH / 2`.
    (b) For each key, we check if the expected_usage (the number of characters in memory and the memory allocated to the vector) doesn't exceed the `MEM_LIMIT_MB`. If it exceeds, we write the vector into the temp file and clear the vector.
- **merge_runs(const vector <string> &sorted_files, const int file_count, const string &output_file_name)**
  - ⋆ **Description**: The function `merge_runs` takes in a vector of sorted file names (`sorted_files`), the number of files (`file_count`), and an output file name (`output_file_name`), and merges the sorted files into a single sorted file.
    It works by reading the first word of each file and finding the smallest word among them. The smallest word is written to the output file, and then the next word is read from the file that contained the smallest word. This process is repeated until all the files have been fully read and merged.
  - ⋆ **Key Points**:
    (a) We are iterating over a vector of size `k` before writing each word. Size `k` is small, we assumed that using a heap may not yield any benefit.
    (b) The Writer maintains a buffer of size `FILE_BUFFER_SIZE` and when the buffer gets filled, it writes the buffer into the output file and clears the buffer.
- **external_merge_stop_withstop(const char *input, const char *output, const long key_count, const int k = 2, const int num_merges = 0)**
  - ⋆ **Description**: The function external_merge_stop_withstop performs an external sorting of a large input file containing strings as key by dividing it into smaller sorted runs, and then merging those runs into larger sorted runs until all runs are merged into a single output file.
    The function sorts the input file using either a trie-based or a run-based sorting algorithm.
  - ⋆ **Key Points**:
    (a) We continue merging until the runs are merged into 1 file or we exceed the merge_count limit.

(b) If the input file is empty, the function creates an empty output file. If the input file contains only one run, the function renames the temporary file to the output file. If the number of runs is larger than one, the function merges the runs into larger runs until there is only one run left. Finally, the function returns the number of merges performed.

(c) Also, during writing output into the merged file, we check if there is only one file to merge (j == 1) and if the `USE_SYMLINK` flag is set to true, and if the output file is not being written to directly (`writing_to_output == false`). If all these conditions are true, a symbolic link is created from the single input file to the output file, which is an efficient operation that does not involve copying the file. If any of the conditions are false, or if creating the symbolic link fails, the merge_runs function is called to copy the input file to the output file.

```
if (j == 1 && USE_SYMLINK && !writing_to_output) {
  remove(merged_file_name.c_str());
  // expensive copy, create symlink instead
  if (symlink(sorted_files[0].c_str(),
      merged_file_name.c_str()) != 0) {
    // can't create symlink, copy
    merge_runs(sorted_files, j, merged_file_name);
  }
}else {
  if (j > 1) {
    merge_count++;
  }
  merge_runs(sorted_files, j, merged_file_name);
}
```

2 Evaluation

2.1 Environment

A Ubuntu 16.04 Server Edition VM with 1GB RAM, 1CPU and 40GB Virtual HD (on an HDD).

2.2 Data

Provided test data: `english-subset.txt` and `random.txt`.
Generated test data: `random10M.txt` 10 million strings with random ascii characters.

2.3 Memory limit

```
g++ --std=c++17 main.cpp -O3 -DUSE_MEM_LIMIT=$MB
```

The maximum number of bytes that can be fit in memory was varied. Value of k was set to 8 and reader/writer buffer sizes were set to `1MB`.

	Time(s)		
Memory Limit (MB)	english-subset.txt	random.txt	random10.txt
16	0.83	38.66	582.89
64	0.70	20.17	551.42
512	0.69	10.65	363.13

For a smaller memory limit, we fit less number of strings in memory before sorting. Hence, the number of chunks are large. Since, disk i/o is of the order $log_k(num\_chunks) * size\_of\_input$, the program is slower.

2.4 Read/Write Buffer

```
g++ --std=c++17 main.cpp -O3 -DFILE_BUFFER_SIZE=$BYTES
```

Buffer sizes of the reader and writer objects were varied. Memory limit was set to dynamic (60% of available memory) and k was set to 8.

	Time(s)		
Buffer Size (KB)	english-subset.txt	random.txt	random10.txt
16	0.64	19.32	459.11
512	0.65	16.25	426.92
1024	0.80	15.64	387.82
4096	0.82	17.53	422.33
8192	0.68	15.06	401.05

The buffer size decreases up to a point (`1MB` here). There is not a lot of change on further decrease. It decreases because we are writing less frequently and hence, the number of seeks should be less. For larger values, we do not see much difference because it is likely that data transfer becomes the bottleneck rather than the seek.

2.5 K

```
g++ --std=c++17 main.cpp -O3
```

The maximum number of files that can me merged at one step was varied. Memory limit was set to dynamic (60% of available memory) and buffer sizes were set to `1MB`. Only tested on `random10.txt`, because other files are too small to require a merge phase.

K	Time(s)
2	652
4	437
8	371
16	244

The disk i/o is of the order $log_k(num\_chunks) * size\_of\_input$. Hence, the runtime should decrease with increase in k.

2.6 Sorting Phase

```
g++ --std=c++17 main.cpp -O3 -DUSE_TRIE=true
```

Sorting could be done using a trie. Memory limit was set to $128MB$, buffer sizes were set to `1MB` and k was set to 8. Only tested on `english-subset.txt`.

Sorting Method	Time(s)	Chunks
trie	3.5	4
stl::sort	0.6	1

Trie took a lot of memory and could only fit a fourth of the input file within a `128MB` limit.

3 Final Parameters

From the analysis before, we decided to keep the following parameters as default for our code.

Parameter	Value
Sorting	stl::sort
Memory Limit	60% of available memory
Buffer Size	1MB