Course: Complex Data Management

Project: Implementation of Operators

Name: Konstantinos Vardakas

Student ID: 522

Email: pcs0522@uoi.gr

Part 1 (Merge-Join)

Initially, in the merge_join.py file, the files to be joined are read using a generator function (using the yield command), read_tsv_line (in the utils.py file), which returns one row at a time, so that entire files do not need to be stored in memory. This function assumes that the files are in TSV (Tab-Separated Values) format, with rows in the form "A\tB", where type(A) = string and type(B) = int.

In the main body of the merge_join function, two generators R and S are initialized (one for each file to be merged) and an empty TSV file (RjoinS.tsv) is created to store the results of the join. Before the merge process begins, the variables r and s are initialized with the first row of the files R_sorted.tsv and S_sorted.tsv, respectively (if either of these does not exist, their value is None). In addition, the variable max_buffer_size is initialized with a value of 0, and will be used to monitor the maximum size of the buffer, which stores the rows of S that match the current row of R.

The merging process begins and repeats as long as there are lines in both files (r and s), i.e. as long as the values of r and s are not None. When the first fields of lines r and s are equal (r[0] == s[0]), the buffer array is created, in which all lines of S that have the same first field as r[0] are stored. This process continues until the first field of the line of S is greater than r[0] (i.e., s[0] > r[0]).

After storing the lines in the buffer, the max_buffer_size variable is updated with the maximum buffer size using the command max_buffer_size = max(max_buffer_size, len(buffer)).

Next, the results are written to the output file, during which the lines of R are merged with the corresponding lines of S that are in the buffer. This process is repeated

for all lines of R that have the same first field as the key. When the first field of the current line of R is greater than the key, the join process is completed for that group and we move on to the new line of R.

Depending on how the first fields of lines r and s compare, the process continues as follows:

- If the first field of r is smaller than s, we proceed to the next line of R.
- If the first field of s is smaller than r, we proceed to the next line of S.
- If the first fields are equal, we continue with the join process.

This process is completed when one of the generators reaches the end of the file (i.e., when r = None or s = None), as merging requires that there be corresponding lines with the same first field in both files.

Part 2 (Union)

The union of two files has been implemented in the union_sorted.py file and is performed in the same way as reading files, using the read_tsv_line generator function. The initialization of the generators for files R and S, as well as the reading of the first line for each file (r and s), follows the same procedure as in the previous implementation. Similarly, an empty output file (RunionS.tsv) is created, which will store the results of the union.

To avoid duplicate entries in the output file, the variable last_written is introduced, which is initialized as None. This variable is used to track whether the current row has already been written to the output file, thus avoiding writing the same tuples multiple times.

The merging process begins and repeats as long as there are rows in either r or s, i.e., as long as at least one of the files still contains data. If the current row from file R (r) is smaller than the current row from file S (s), row r is written to the output file, provided it has not already been written (i.e., if it is not equal to the value of the last_written variable). We then proceed to the next row in file R. If, on the other hand,

the current row of S (s) is smaller than the row of R (r), the row s is written to the file, provided it has not been written again, and we set s as the next line of S.

When the strings r and s are identical (i.e., r[0] == s[0]), the string is recorded only if it has not been recorded previously, and then we proceed to the next string in both files. The process continues until the strings in both files have been exhausted.

This method ensures that there are no duplicate results, without requiring temporary data storage in a buffer.

Part 3 (Intersection)

To implement the intersection of two files (intersection) performed in the intersection_sorted.py file, the generator read_tsv_line function is used again to read the files, allowing one row to be read at a time without requiring the entire files to be loaded into memory. Similarly, the initializations and file creation are performed as in the previous cases.

The intersection process begins and repeats as long as there are rows in both files (i.e., as long as r and s are not None). Specifically:

- When the strings r and s are identical (r == s), then this common string is part of the intersection. To avoid writing duplicates, we compare the row with the last_written variable and write it only if it has not already been written. After writing, we move on to the next row in both files.
- If the sequence r is smaller than the sequence s (r < s), we proceed to the next sequence in R.
- Conversely, if the sequence s is smaller than the sequence r (s < r), we proceed to the next sequence of S.

Part 4 (Set-Difference)

To implement the set-difference between two sorted files (set difference, from the difference_sorted.py file), the same approach was used as with the read_tsv_line function and the corresponding initializations. The algorithm then runs as long as there is a subsequent row in the R file, with three possible cases:

- If the element of R is greater than the element of S (r > s), the index of S increases (we cannot know at this point whether the specific r belongs to the difference R - S or to the intersection of R and S; we only know that s belongs to the difference of S - R, which is not the function's requirement)
- If the element of R is the same as the element of S (r == s), then it belongs to their intersection, so it is not recorded and both pointers move on.
- If the element of R is smaller than the element of S (s > r), then this element belongs to their difference and is recorded, if it has not already been recorded. We then proceed to the next index of R.

Part 5 (Grouping and aggregation)

To implement the grouping and aggregation of the R tuples according to the first field (group_merge_sort.py file), the sort-merge algorithm was used, adapted to perform the merging during the merge process.

First, the "R.tsv" file is read into memory using the "read_tsv" function (in the utils.py file), which returns a list of tuples (key, value), where the first field (key) remains as a string, while the second field (value) is converted to an integer. Similar to the read_tsv_line function in the utils.py file, this function assumes that the file format is TSV (Tab-Separated Values), with the format of the rows being "A\tB", where type(A) = string and type(B) = int.

Next, the merge sort algorithm is applied through the corresponding function, recursively splitting the list until sublists of individual elements are obtained. The main difference in this implementation is the adaptation of the merge process, which is done in the "merge" function. During the merge, the following process takes place:

- If the first field of the tuple from the left table is smaller than the corresponding field of the tuple from the right table, the left tuple is added to the result..
- Similarly, if the first field of the right-hand pair is smaller, the right-hand pair is added to the result.
- If the first fields are equal, the second fields are added together and a new tuple is created with this sum.

The result of the merge is returned as a list of tuples (key, value), which can be used to write to the output file.

Finally, the "group_merge_sort" function calls "merge_sort" and writes the result to the "Rgroupby.tsv" file. This achieves the desired grouping and aggregation of tuples, while the process is performed entirely in main memory.

Instructions for execution

The following reasoning is used to execute each file:

- Open the terminal in the .py files folder
- Add the data files to the same folder
- The general command is: python {scriptname.py} args

Specifically, for all files except group_merge_sort.py, the arguments are <R_file.tsv> <S_file.tsv> [output_file.tsv] (with <> indicating mandatory fields and [] indicating optional fields), e.g.:
>>> python merge_join.py R_sorted.tsv S_sorted.tsv RjoinS.tsv

Regarding the group_merge_sort.py file, the arguments are <R_file.tsv> [output_file.tsv]