|  |  |  |
| --- | --- | --- |
|  | **stage\_1\_a** | **stage\_2\_c** |
| **process\_record\_1** | 0.134125 | 0.001082 |
| **process\_record\_2** | 0.120616 | 0.020904 |
| **process\_record\_3** | 0.122741 | 0.022236 |
| **process\_record\_4** | 0.126221 | 0.027764 |

**Table 1** - Comparison of process time (seconds) between stage\_1\_a and stage\_2\_c.

process\_record\_1 utilizes binary search techniques to search the user files for the users that satisfy the query; that is, users with the location “Nebraska.” The initial search is a binary search of complexity log2(n). There is additional logic to search the preceding and succeeding files while the query continues to be satisfied. Assuming that the number of files containing query matches is small compared to the total number of files, this additional logic can be ignored. Thus, the algorithm complexity of process\_record\_1 is log2(n).

process\_record\_2 also involves binary search; however, when it opens a file, it searches the entire file, which includes multiple message entries. Its complexity is log2(n) \* number\_of\_messages\_per\_file, since log2(n) files are searched, and each file searched includes number\_of\_messages\_per\_file messages. process\_record\_2 also uses an array to track the users associated with the messages, so that users are not duplicated. Each time a match to the query is found, the array must be checked for duplicates, and the array also grows longer. process\_record\_2 also involves additional logic to search for matches in the preceding and succeeding files, though this logic is forced to search all entries in the preceding and succeeding files, even when the query fails. If we assume that the number of messages per file is small compared to the number of files, and we also assume that the number of matches is small compared to the total number of message records, then we can simplify the algorithmic complexity to log2(n). process\_record\_2 is more complex than process\_record\_1 when n is small, but when n is extremely large, the complexities are similar.

process\_record\_3 contains both the logic of process\_record\_1 and process\_record\_2. From process\_record\_2, it utilizes a binary search of files that includes all message entries in each file and checks an array to prevent duplicates. It also includes logic to search preceding and succeeding files. From process\_record\_1, it uses a binary search. It also compares the user IDs that match Nebraska with the user IDs that sent messages between 8am and 9am, which requires repeated iteration through an array. process\_record\_3 is thus more complex than process\_record\_1 and process\_record\_2, with a complexity of at least 2 \* log2(n).

process\_record\_4 contains all of the logic of process\_record\_3, with the use of a matrix data structure instead of an array, and the addition of a set of statements to track the largest number of messages sent between 8am and 9am by a user from Nebraska, which is updated if a larger number of messages is encountered. These additional statements are not loops and do not add complexity for a very large n. As such, process\_record\_4 has essentially the same complexity as process\_record\_3: 2 \* log2(n).

As shown in Table 1 above, each of the queries from stage\_2\_c is significantly faster than its counterpart from stage\_1\_a. From these results, we can see that sorting the data and storing it logically reduces the significantly reduces the time required to execute a query, as long as the query can take advantage of the sorting pattern. If the user data files were sorted according to user ID and we were searching for users from Nebraska, then the sorting would not be advantageous. However, when the user data files are sorted by location and we are searching for users from Nebraska, then we can use binary search to reduce the number of files we have to check, which in turn reduces the number of time-consuming file I/O operations. The binary search techniques utilized in stage\_2\_c are much more efficient than the linear search techniques used in stage\_1\_a.

The size of n has a very limited impact on our timing results when our queries can utilize the sorted nature of the data. Because our queries utilize binary search techniques, they have a complexity of about log2(n). If the size of n is doubled, we only have one more file to check on average during the initial search. For process\_record\_1, with our current set of 2,000 user data files, we have approximately 11 files to check during the initial search (log2(2,000) = 10.97), which requires about 0.001082 seconds. To double the number of files to check during the initial search, we would have to increase the number of user data files to approximately 2,000,000 files (log2(2,000,000) = 20.93). Therefore, we would expect the amount of time required for the query in process\_record\_1 to double if n is increased to 2,000,000. The behavior of the other queries would be similar.