|  |  |
| --- | --- |
| Search Application | PRABHU Rohit |

Contents

[1. Requirement: 2](#_Toc96521970)

[2. Assumptions: 3](#_Toc96521971)

[3. High Level Solution: 3](#_Toc96521972)

[Main focus: 3](#_Toc96521973)

[High level implementation: 3](#_Toc96521974)

[Classes: 5](#_Toc96521975)

[4. Test Cases: 6](#_Toc96521976)

[Test-1: 6](#_Toc96521977)

[Test-2: 7](#_Toc96521978)

[Test-3: 8](#_Toc96521979)

[Test-4: 8](#_Toc96521980)

[5. Performance Test Cases: 9](#_Toc96521981)

[Test-1: 9](#_Toc96521982)

[6. Next Steps: 12](#_Toc96521983)

# Requirement:

Below is the requirement for the command line search application and status in current implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Requirement** | **Description** | **Status** |
| 1 | Command line application | * 1. Search Data from JSON file shared by Zendesk.   2. Return Results in human readable format. | Implemented a command line tool in C++. |
| 2 | Input DATA to be parsed | * 1. tickets.json   2. users.json   3. organization.json | Implemented a parser to parse input JSONs. |
| 3 | Where the data exists, values from any related entities should be included in the results | Searching organization by id should return its tickets and users. | Implemented.   1. User search returns Users, Tickets and Organization details. 2. Ticket search returns Ticket, Users and Organization details. 3. Organization search returns Organization, User and Ticket details. |
| 4 | The user should be able to search on any field, full value matching is fine. | "mar" won't return "mary". | Implemented |
| 5 | The user should also be able to search for empty values, e.g. where description is empty. | “” or Empty attribute or field value should be searchable. | Implemented |
| 6 | Ideally, search response times should not increase linearly as the number of documents grows. | - | Implemented. Please refer to Performance Test results. |

# Assumptions:

We can navigate between records as mentioned below.

1. Organization ID -> User ID -> Ticket (Assignee ID)
2. Ticket (Assignee ID) -> User ID -> Organization ID
3. User ID

-> Ticket (Assignee ID)

-> Organization ID

# High Level Solution:

## Main focus:

1. Designing an algorithm and logic to ensure that we have
   1. Consistent search results
   2. In a consistent and predictable time
2. Related objects fetched for search record matching user search criteria

## High level implementation:

We have implemented a search mechanism that works in **two** steps:

1. Build index at start – ONCE ONLY.
   1. Index is built for DATA
   2. Index is built for RELATIONS

At index build time we parse JSON files convert them into flat records. These flat records are then indexed and stored in a unordered multimap container.

1. Runtime search - Search index for user search criteria from the already created index.
   1. Search matching record
   2. Fetch related objects

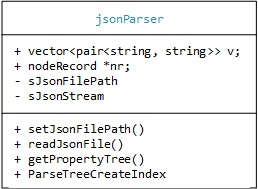
At run time search criteria from the user is searched from these containers. For each of the fetched record we then search related objects in the containers that have relater objects indexed and stored.

****

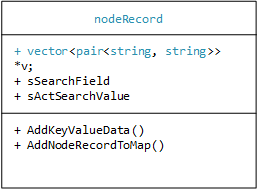
**Figure 1.0 – High-level implementation**

## Classes:

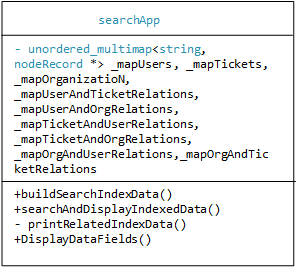
1. Class for JSON parsing



1. Flat records to be stored in MAP



1. Search Application class



# Test Cases:

Below are the files to be used to execute below test cases.

****

Test-1:

This test will verify that when a user is searching for User records, user record if found is displayed along with tickets assigned to the searched user and the organization.

**Test Input:**

users.json, tickets.json, organizations.json

**Test Steps:**

1. Launch the command line utility
2. Follow the steps prompted to user
   1. Select search category – **1** for users
   2. Select search field - **\_id**
   3. Select value - **74**

**Test Verification:**

Test result should be as shown below:



Test-2:

This test will verify that when a field is searched for an **empty value** for example “” we have results returned if there are records available with an empty field value.

**Test Input:**

users.json, tickets.json, organizations.json

**Test Steps:**

1. Launch the command line utility
2. Modify users.json for \_id “22” to have “alias” as empty i.e. “”.
3. Follow the steps prompted to user
   1. Select search category – **1** for users
   2. Select search field - **alias**
   3. Select value – “”

**Test Verification:**

Test result should be as shown below:



Test-3:

This test will verify that when a user is searching for a ticket record, ticket record if found is displayed along with the users assigned to the ticket and users organization.

**Test Input:**

users.json, tickets.json, organizations.json

**Test Steps:**

1. Launch the command line utility
2. Modify users.json for \_id “22” to have “alias” as empty i.e. “”.
3. Follow the steps prompted to user
   1. Select search category – **2** for tickets
   2. Select search field - **assignee\_id**
   3. Select value – **58**

**Test Verification:**

Test result should be as shown below:



Test-4:

This test will verify that when a user is searching for an organization record, organization record if found is displayed along with the users in the organization with the tickets assigned to those users.

**Test Input:**

users.json, tickets.json, organizations.json

**Test Steps:**

1. Launch the command line utility
2. Follow the steps prompted to user
   1. Select search category – **3** for organization
   2. Select search field - **\_id**
   3. Select value – **101**

**Test Verification:**

Test result should be as shown below:



# Performance Test Cases:

Test-1:

This test will verify that with the increase in data search time is not increasing linearly.

**Test Input:**

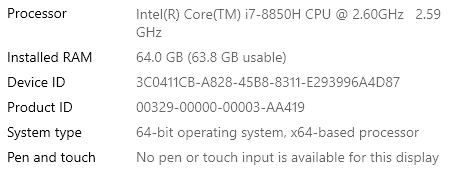


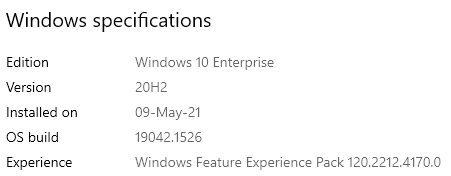
**Test Steps:**

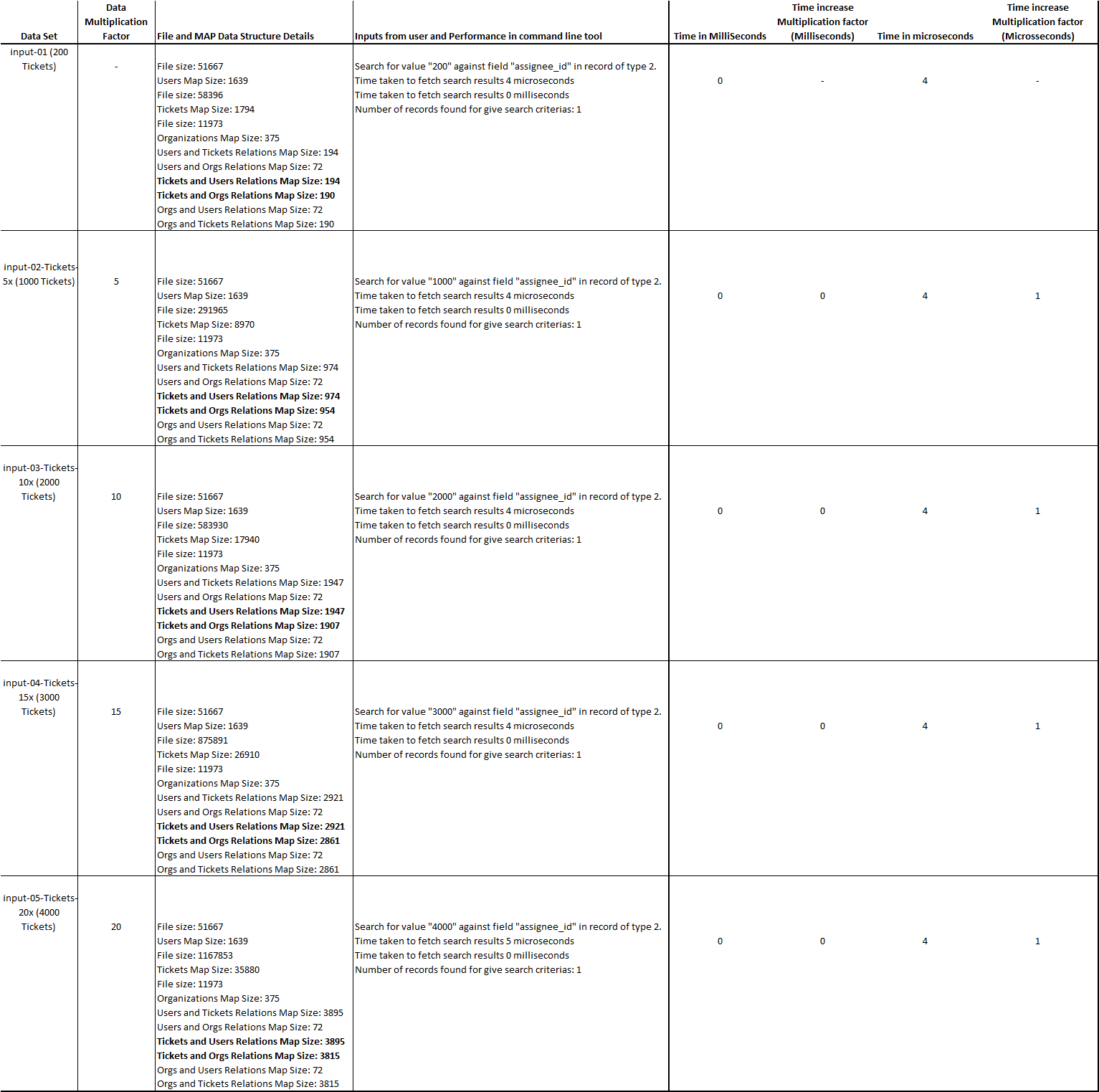
1. We have duplicated tickets records and made one record unique to be searched in tickets records with no relations to users and organizations data. This is done to ensure that we have linear increase in DATA records. And that there is a consistent search result.
2. Run the tool with search options in column “*Inputs from user and Performance in command line tool*” in below table.

**Test Verification:**

* + 1. Test results should be as shown below.
    2. Below results are generated by executing the program on machine with below configuration.







# Next Steps:

This implementation is prototype designed and implemented in a limited time-frame.

Search like mentioned in the coding challenge can be very complex. It can be very resource intensive also.

Some possible improvements in design:

1. Index can be persisted on disk for quick load at run time.

Pros:

1. Indexed build time is minimized.
2. No need to parsing again.

Cons:

1. We need to design a logic to check data modifications and re-build index.
2. Involves a lot of disk read and write.
3. Creation of index for relations needs more careful design to implement an optimized data structure to store and retrieve relation data.
4. Based on a priori know of customer data and related objects INDEX can be partitioned to minimize search data records to search from.
5. Index can be optimized based on knowledge of more frequently queried data by users. We can implement a CACHE mechanism based on most recently or least recently or most frequently used data.

------------------ X ------------------