**Multi Dimensional Search Design Document**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Modified By | Summary of Changes |
| 1.0 | 04/23/2015 | Madhao Wagh | Created |

Document History

##### Table of contents

[1. Project Overview 4](#_Toc417591034)

[2. Design 4](#_Toc417591035)

[3. Reasoning behind the Data Structures Used 4](#_Toc417591036)

[4. Operations Performed 5](#_Toc417591037)

[5. Results on Sample Inputs 5](#_Toc417591038)

# Project Overview

The main objective of this project was to implement efficient multi dimensional search capabilities for the given use case. In this project I used a combination of HashMap, TreeSet and BitSet operations to be able to perform efficient insert delete and search operations on the given data.

# Design

All the information related to a customer is stored in a CustomerInfo object.

* UserID --Long field holds the customers id
* Revenue – holds the total amount spent by the customer on any category
* Categories – holds the categories that are interest to the user, this is a BitSet field. The categories of interest are set to 1 and others set to 0.
* Purchases – holds the total number of purchases done by the user

Three main data structures were used in this project.

1. UserHash : This hash map holds a pair of user\_id and customer object , it is used to find customer objects based on the user id
2. CategoryHash: This holds a hash map of the category id and a tree set of all the Customer objects who are interested in this category .The TreeSet is ordered by the total revenue spent by the customer
3. UserTree: This is a Tree set of all the users arranged according to the revenue

# Reasoning behind the Data Structures Used

* Categories BitSet
  + I used the categories bitset as bitwise operations are faster and it will make the calculations for Same-Same operation very fast
  + It is also easier to check the number of categories using the cardinality operation
  + BitSet is also space efficient
* UserHash HashMap
  + Most of the operations require users to be searched based on their ID
  + The UserHash HashMap is based on UserID and makes the access faster
  + Bieng a hashMap we don’t need to maintain it if any field in the Customer object changes
  + The UserID is not going to change this makes HashMap a good choice
* CategoryHash
  + Here we have a Tree Set of Customer objects for each Category in the Hash Map
  + This data structure is used especially for the Top Three operation
  + The Tree Set has a comparator on Revenue of the Customer object
  + This makes finding the top 3 elements easier
  + Although it adds a overhead in other operations it improves the overall time complexity
* UserTree
  + Here we have a Tree Set of Customer objects based on their Revenue
  + This data structure is used especially for the Range operation
  + The Tree Set has a comparator on Revenue of the Customer object
  + Although it adds a overhead in other operations it improves the overall time complexity

Notes:

* I have used different comparators for the two tree sets based on Revenue, because I observed in my testing that such combination was giving better running time for all the large data sets
* One disadvantage of the tree operations is that we have to manually remove and add the customer objects whenever the Revenue of a Customer object changes. Another approach could have been to write our own Tree Set that could automatically adjust to changes in underlying customer objects.

# Operations Performed

* 1. Insert(Customer id, list of categories)
  2. Find(Customer id)
  3. Delete(Customer id)
  4. AddRevenue(Customer id, Purchase amount)
  5. AddInterests(Customer id, list of categories)
  6. RemoveInterests(Customer id, list of categories)
  7. NumberPurchases(Customer id)
  8. TopThree(Category)
  9. Range(low, high)
  10. SameSame()

# Results on Sample Inputs

**Folder: P3-in-degen**

|  |  |  |
| --- | --- | --- |
| File Name | Returned Value | Time Required(milli sec) |
| In1.txt | 38 | 27 |
| Ink.txt | 419 | 446 |
| Inl.txt | 510 | 10615 |
| Inc.txt | 989 | 239 |
| Inxk.txt | 348 | 1740 |

**Folder: P3-S2**

|  |  |  |
| --- | --- | --- |
| File Name | Returned Value | Time Required(milli sec) |
| p3-s2-ck.txt | 282 | 13934 |
| p3-s2-d.txt | 802 | 449 |
| p3-s2-k.txt | 264 | 500 |
| p3-s2-l.txt | 93 | 134 |

**Folder P3-S3**

|  |  |  |
| --- | --- | --- |
| File Name | Returned Value | Time Required(milli sec) |
| p3-s3-ck.txt | 705 | 13767 |
| p3-s3-d.txt | 861 | 431 |
| p3-s3-k.txt | 990 | 584 |
| p3-s3-l.txt | 558 | 129 |

**Folder P3-S4**

|  |  |  |
| --- | --- | --- |
| File Name | Returned Value | Time Required(milli sec) |
| ss-ck.txt | 682 | 8588 |
| ss-xk.txt | 738 | 1464 |

Note: The runtimes were observed on a Linux machine . I observed that it took a little more time on my windows system.