Multi-Dimensional Search - Report and Read me

Group Name - G99
Names of members - Kanchan Waikar, Kavya Krishna Maringanty, Sruti Paku

How to Execute Project?

1. Extract zip file
2. Execute following command in order compile all java programs

```
>javac *.java
```

3. In order to execute the program, execute following command, give it input to the file that contains LP4 input as shown below.

```
>java MultiDimensionalSearchDriver
"S:\IMPL_DS\Algorithms\MultiDimensionalSearch\src\lp4-data\lp4-1.txt

1450.08
Statistics for Step Multiple Dimensional Search :
Time: 12 msec.
Memory: 3 MB / 192 MB.
```

As we can see above, it prints the overall value along with statistics of the execution.

Multi-Dimensional Search - Analysis and Implementation Report

Brief Description:

Multi-Dimensional Search typically involves search on multiple fields of the same entity in different ways. Each of the Search requirement needs to be adhered to and the overall implementation of such search features needs to be optimized considering time complexity as well as space complexity. Typically, indexing is used in order to meet requirement of Searches since complete table lookup is an expensive operation and it may not always be feasible to load complete table in memory if tables are huge. Databases needs external solution based approach since it may not always be possible to load complete data or even complete index in-memory at runtime. There are different types of indexes that are used in databases; B+ tree based index, Hash index, Inverted Index for string searching (Lucene/SOLR/Elastic Search), etc are typical solutions to problem of optimization of Search in databases. With each additional index, explicit measures need to be taken for updation of the index, every time when the item is getting updated.

Our requirement is to implement In-memory multi-dimensional indexing that uses advanced data structures and implements indexes efficiently in-memory itself. Given that data is going to be able to fit in memory at runtime, we need to do analysis of different operations to be performed, analyse and do trade off of time complexity vs space complexity and decide the data structures to be used for implementing efficient index(s).

Implementation Approach and optimization Steps:

After looking at complete set of operations, we can clearly see that the dominant operations of lookup are happening on Desc substrings, price and id. On these three columns, we need an optimal implementation that implements both, find by id as well as find by range. Although HashMap is a better data structure for key value retrieval, it does not work when we have to implement range search functionality. For Range Search, trees are most efficient.

Java implements Red black tree - which is a balanced tree structure implemented by Java - as part of TreeMap, and the same can be used in order to get the efficient implementation for lookup by key as well as range.

We need three primary indexes

1. TreeMap Index on Description substring to be used by -
    - findMinPrice(long des)
    - findMaxPrice(long des)
    - findPriceRange(long des, double lowPrice, double highPrice)
2. TreeMap Index on Price used by -
    - range(double lowPrice, double highPrice)
3. TreeMap index on id - to be used by
    - find(id)
    - priceHike(long minid, long maxid, double rate)

findPriceRange(long des, double lowPrice, double highPrice) could have used index#1 or index#2 but we decided to use #1 since that would provide more optimal solution. It is always better to do the filtering first on narrower key as it leads to smaller initial subset selection.

4. mapForSameSame - Tree index for solely doing samesame computation at runtime.

samesame() function needs two operations, comparison of all elements with description length greater than 8 - this can be done on raw data directly but that would be very inefficient, that is why we decided to create a separate map especially for optimizing this functionality. This map was further tuned, to first check only number of elements in the description, then sum of elements in the description, and when both are exactly same, we do arraylist comparison.

Space complexity of the implementation is not as high as it seems as replication of references happens here for values. In case of keys, replication does happen.

Additional optimization was done by computing hash code only once on the object in which hashcode was dependent on single immutable field.

Space complexity

Id - **replicated once** present as a key in #Index 3
Description - **replicated twice** Present as key in sameSame map as well as index#2
Price - **replicated once** present as key in index#2

Apart from storing these values as part of original object, we store them redundantly in order to improve the performance of the lookups by id as well as range.

Red Black Tree/AVL tree are efficient data structure for performing find and find by range operations. They implement balanced tree structure.

Tree Structure Performance

We can see that red black tree's performance is optimal hence we see that all above inputs execute in barely few seconds.

Time Complexities of Red Black Tree:

Insert: O(log n)
Delete: O(log n)
Search : O(log n).
Search by Range: O(log n + num_elements_returned)

Java' Treeset, Treemap implement red black tree internally. Although TreeMap may not be as good as HashMap, for larger data sizes, they do provide search by range functionality in O(log n +k) range itself - which is pretty small .

Multidimensional search requirement for vendors like amazon typically would involve 95% of searches on price, ranges and by id, Insertion and deletion operations would be considerably less, hence it makes sense to design data structures that provide high performance on Searches rather than insertions or deletions. Treemaps, TreeSets, HashSets provided by Java are kinds of data structures that can be typically used for giving high performance search results.

Execution Statistics

Please find stats for other inputs described in table below.

Sr. No.	File Name	Number of Commands	Output	Time Taken	Memory
1.	Ip4-1.txt	7	1450.08	12 msec	3 MB / 192 MB
2.	Ip4-2.txt	12	4146.32	12 msec	3 MB / 192 MB
3.	Ip4-3-1k.txt	1k	52252.36	55 msec.	7 MB / 192 MB.
4.	Ip4-4-5k.txt	5k	490409.01	145 msec.	29 MB / 192 MB.
5	Ip4-5-ck.txt	100000	173819092858.24	15785 msec	118 MB/960 MB
6	Ip4-bad.txt	100000	1016105100.00	25161 msec	1199 MB / 1762 MB.

We can clearly see here that the implementation executes in milliseconds even for large inputs and for huge inputs, time taken including file read, is barely 25 seconds.

<u>Conclusion</u>

We can clearly see that whenever we need to do a lookup by key, we use hashmaps and whenever we need to do a lookup by range, we use treemaps which are internally based on red black balanced trees. Combination of both can be used in order achieve the optimal system performance as per system needs. We can combine multiple structures, create more evolved implementations using combinations of these in order to create exact index structure that works well for our requirements. For unary structures, we can use Sets, Lists whereas for key value based requirements, we use maps.

Time complexity improvement can be done by introducing redundancy in the data in form of index only when required - A tradeoff needs to be made between time complexity and space complexity based on the execution environment.