Chuying Huo: chuying.huo@duke.edu
Emma Chua: emma.chua@duke.edu
Anushka Aggarwal: anushka.aggarwal@duke.edu

Data Set Selection

Attributes
gender (String)
age (double)
city (String)
profession (String)
degree (String)
academic pressure (double, scale 1–10)
study satisfaction (double, scale 1–10)
financial stress (double, scale 1–10)
depression (int, scale 0–10)
have you ever had suicidal thoughts? (boolean/String “Yes”/“No”)
family history of mental illness (boolean/String “Yes”/“No`)

Selection Rationale
https://www.kaggle.com/datasets/adilshamim8/student-depression-dataset
This dataset captures attributes related to student mental health, allowing for both simple and compound queries. It enables exact match queries on categorical data (e.g., gender, city), and range queries on numerical mental health indicators (e.g., academic pressure, depression level).

Data Structure Justification

Method for exact match queries
Data structure: Hashmap

Time Complexity Analysis
The exactMatchQuery method uses a Map<String, Map<Object, List<Record>>> for efficient indexing by attributes, where the outer map holds attribute names and the inner map connects attribute values to lists of matching records. Initially, when building the index, the time complexity is O(n), where n is the number of records, since it iterates over all the records. Once the index is built, the lookup for each queried attribute becomes O(1), as each attribute value lookup in the inner map is constant time. If multiple attributes are queried, the overall time complexity becomes O(m), where m is the number of queried attributes, since each attribute's lookup is constant time, and the intersection step can be done efficiently once the candidate lists are obtained. Therefore, the query time is linear with respect to the number of attributes O(m), and constant for each lookup within an attribute.

Space complexity
The space complexity of this method is O(n * m), where n is the number of records and m is the number of queried attributes. This arises from the temporary index storing lists of records for each attribute-value pair and the candidate lists for each queried attribute. While the index structure allows efficient lookups, it requires space proportional to the number of records for each attribute, leading to a space complexity of O(n * m).

Overall justification: The HashMap used in the exactMatchQuery method is efficient for attribute-based lookups because it provides constant-time (O(1)) access to records once the index is built.

Method for range queries
Data structure: TreeMap

Time Complexity Analysis
The rangeQuery method uses a TreeMap<Comparable, List<Record>> to efficiently store and retrieve records based on a specific attribute's value. Inserting all records into the tree map has a time complexity of O(n log n), where n is the number of records, due to the logarithmic time complexity of insertions in a balanced tree map. After the records are inserted, the query for the range using subMap is O(log n + k), where k is the number of records that fall within the specified range. This time complexity arises because subMap efficiently retrieves the range of records in logarithmic time, and the actual retrieval of records takes linear time in relation to the number of records within the range.

Space Complexity
The space complexity of the rangeQuery method is O(n), as the TreeMap stores all records indexed by the specified attribute value. Each record is stored as part of a list in the map, requiring space proportional to the number of records.

Overall justification: The TreeMap is an efficient data structure for range queries due to its balanced nature, ensuring O(log n) time complexity for insertions and logarithmic access times for range queries. The structure's ability to maintain ordered data allows efficient retrieval of ranges with minimal overhead, making it a good choice when querying ranges within large datasets

Method for computing a statistic
Data structure: ArrayList

Time Complexity Analysis
The getDatasetStatistics method calculates the average of a specific attribute over all records. The time complexity of this method is O(n), where n is the number of records, as it iterates through all records once to compute the sum for the given attribute. There are no nested loops, and the method only performs simple arithmetic operations during the iteration.

Space Complexity
The space complexity of the getDatasetStatistics method is O(1), as it only uses a few variables (sum, count) to compute the statistic and does not require any additional space proportional to the size of the dataset.

Overall justification: The ArrayList used in the getDatasetStatistics method offers efficient sequential access and is well-suited for iterating over records to compute statistical values. Since the method only performs a single iteration over the dataset and uses constant auxiliary space, it ensures that the time and space complexities remain minimal. The straightforward nature of ArrayList for linear traversal makes it ideal for use cases where the goal is to compute aggregates or summaries across all records

Performance Comparison

The exactMatchQuery method is the most efficient when filtering on indexed fields or highly selective criteria. It provides optimal performance for queries that eliminate many records early, such as filtering by gender before other attributes. However, its performance degrades linearly with the dataset size, especially as the number of records increases. This method works best when a few records are expected to match, making it ideal for highly selective queries.

The rangeQuery method is more costly initially due to the O(n log n) time complexity required for inserting all records into a TreeMap. However, it becomes highly efficient for repeated queries on the same numeric attribute, as subsequent queries can be processed quickly. This method outperforms the exactMatchQuery for numeric range queries once the initial setup is completed, making it a good choice for range-based filtering over large datasets.

The getDatasetStatistics method always requires a full scan of the dataset, as it must examine each record to compute statistics such as averages. While it is not suitable for repeated queries due to its O(n) complexity, it is ideal for one-time calculations, especially when statistical analysis of an entire dataset is needed. This method provides a straightforward approach to aggregate data, albeit at the cost of scanning all records.

Current Limitations

1. Exact Match Query (exactMatchQuery)
Inefficient Indexing: The method builds a temporary index for each query, which involves iterating over the entire dataset. This can be inefficient for large datasets because the index is rebuilt every time a query is made. A more efficient approach would be to build the index once and reuse it.
Intersecting Lists: The method collects lists of matching records and intersects them in memory. This could lead to inefficiencies as the number of attributes queried increases, especially if there are many records.

2. Range Query (rangeQuery)
Initial Insertion Cost: Inserting records into the TreeMap has an initial time complexity of O(n log n), which can be slow for large datasets. This is because the TreeMap needs to maintain sorted order while inserting.
Inefficient for New Range Queries: After the initial insertion, subsequent range queries are efficient (O(log n + k)), but if we need to perform multiple different range queries on various attributes, we would need to rebuild the data structure each time, which is inefficient.

3. Dataset Statistics (getDatasetStatistics)
Full Dataset Scan: The method requires scanning the entire dataset to calculate statistics, which takes O(n) time. This is inefficient for large datasets, especially when multiple statistics need to be computed or if the data is updated frequently.
No Optimization: The method recalculates the statistic every time it's called, and it doesn't use any optimization techniques like caching or indexing. This makes it slower as the dataset grows.
General Limitations Across All Methods:
High Memory Usage: All methods rely on large in-memory data structures (like ArrayList, HashMap, and TreeMap), which can consume a lot of memory. This could be a problem for large datasets, as the memory required grows with the size of the dataset.
Lack of Efficiency for Large Datasets: The methods don't use more efficient algorithms or data structures like balanced search trees, tries, or hash-based indexing for fast lookups. This could lead to slower performance as the size of the dataset increases.

Potential Improvements

Exact Match Query (exactMatchQuery)
Persistent Indexing:
Instead of creating a temporary index for each query, we could store the index permanently after loading the dataset. This can be done using a HashMap where keys are attribute values and values are lists of matching records. This will allow faster querying since we won’t have to rebuild the index for every search.
Use Hashing:
Instead of iterating through each record to find matches, we could use a hash table (like HashMap) to store records by their attribute values. This allows for O(1) average time complexity for lookups, speeding up the query process.

Range Query (rangeQuery)
Use Sorted Data:
If the data is sorted (or indexed in a way that it can be efficiently searched), we could use a binary search to quickly find the range, reducing the time complexity to O(log n) for each query. This is more efficient than linearly iterating over all records.
TreeMap for Range Queries:
Use a TreeMap or SortedMap to store the data. Since TreeMap automatically sorts the keys, range queries can be executed more efficiently (O(log n) for insertion and O(k) for querying where k is the number of records in the range).

Dataset Statistics (getDatasetStatistics)

Running Sum and Count:

Instead of scanning the entire dataset every time we need to compute statistics like the average, maintain a running sum and count for each attribute as we load the records. This allows us to compute the average in O(1) time rather than O(n).

Use of Caching:

Store the computed statistics (like average or sum) for each attribute after the first calculation. On subsequent requests, simply return the cached value, which saves time by avoiding repeated calculations.