Oluwadarasimi Ajiboye

Nathan Levine

Zijan Li

Professor Eric Noel Fouh Mbindi

COMPSCI 201-SP25

1 May 2025

Pokemon Final Project Documentation

Project Overview

The Pokémon Dataset Explorer project was built with the goal of enabling flexible, efficient querying over a large dataset of Pokémon records. The dataset includes attributes such as name, ID, type, base stats, individual stat categories, and weaknesses to various elements. We designed a modular backend system that supports exact-match lookups, numerical range-based queries, and statistical averaging over filtered results. Our implementation focuses heavily on performance optimization, modular design, and clean error handling for unsupported queries.

Data Structure Justification

To meet performance demands and reduce time complexity, we made careful, attribute-specific choices in data structures:

- **Exact Match Indexing:**

  Initially, our exactMatchQuery method iterated over the entire list of Pokémon (`ArrayList<Pokemon>`), checking each Pokémon to see if it matched the given attribute and value. This linear approach was inefficient for large datasets. To address this, Dara Ajiboye designed and implemented a multi-layered `HashMap<String, Map<Object, List<Pokemon>>>` called `attributeIndex`, which indexes

Pokémon by supported attributes such as `"name"`, `"id"`, `"type"`, `"attack"`, etc. This allows constant-time O(1) lookups after preprocessing during data load.

- **Range Query Optimization:**

  Nathan Levine implemented the rangeQuery method using a linear scan across the `pokemonList`, but added a **min-heap (priority queue)** to efficiently collect and retain only the top **k** results within the specified range. This avoids unnecessary sorting of the entire dataset and ensures time-efficient retrieval for user-defined limits. The range query supports attributes like `hp`, `speed`, and `base_stats`.

- **Attribute Aliasing & Standardization:**

  Dara added support for aliases like `spattack` and `sp_attack`, and ensured all string-based attributes are normalized (converted to lowercase) for consistency and robustness in query matching.

- **Average Query Filtering:**

  Zijan Li led the implementation of the averageQuery method, which computes the average of one attribute (e.g., "`speed`") among all Pokémon that satisfy a threshold condition on another (e.g., `"grass_weakness" < 1.0`). This function scans the dataset linearly and is resilient to missing or unsupported attributes, returning 0.0 if no matches are found or if attributes are invalid.

Algorithmic Complexity Analysis

Method	**Time Complexity**	**Space Complexity**	**Details**
`loadDataset`	O(n)	O(n)	Parses and indexes all Pokémon from CSV during one pass
`exactMatchQuery`	O(1)	O(n)	HashMap lookup

			using pre-built attributeIndex
`rangeQuery`	O(n log k)	O(k)	Linear scan + min-heap (k = limit) to collect top results
`averageQuery`	O(n)	O(1)	One-pass filtering and summation based on conditions

Performance Comparison

- Exact Match Query (Before vs After)

Before (Linear Search): Each query required scanning the entire dataset, resulting in poor performance at scale, especially when repeatedly querying the same attribute.

After (HashMap Indexing): Each supported attribute is mapped to its possible values, and each value maps directly to a list of Pokémon. This enables constant-time access, improving orders-of-magnitude performance, particularly when executing recurring or repeated queries.

- Range Query Efficiency

Nathan's min-heap approach ensures that even with large ranges, the result set size is bounded by a user-defined limit. Instead of sorting all matching entries, we keep a dynamically updated heap of top candidates, significantly reducing overhead. This keeps the query responsive even when hundreds of entries match the criteria.

- Average Query Execution

The average query performs a single linear pass and has a small constant space footprint. Though it lacks pre-indexing (due to the flexible and arbitrary filtering), it remains efficient and scalable. Zijan's implementation avoids unnecessary exceptions by returning 0.0 for unsupported fields, ensuring robustness.

Design Evolution and Contributions

- **Dara Ajiboye:**
  - Created and maintained the GitHub repository that the group worked on, and designed and implemented the Pokémon model class with well-defined attributes and debugging support.
  - Transformed the **exact match functionality** from a naïve linear scan using an ArrayList at first into a robust, scalable HashMap-based index system. This was a key performance enhancement.
  - Added support for alternate attribute names and cleaned up input formatting so the queries work smoothly even if the attribute names vary a bit.
  - Created and maintained detailed inline comments and sample test code in `Main.java` for real-time validation.
- **Nathan Levine:**
  - Took the lead on developing the **rangeQuery functionality**, incorporating a min-heap for top-N selection within a value range.
  - Ensured that results were sorted in descending order and limited to the specified number of entries, offering both performance and usability improvements.
  - Collaborated on attribute parsing and validation for numerical fields.
- **Zijan Li:**
  - Designed and implemented the **averageQuery method** to compute aggregate statistics from the dataset.

- Added safeguards for edge cases such as empty result sets and unsupported attribute access.
- Contributed to the logic that parses and validates numerical attributes across queries.

Limitations and Future Improvements

While the current system is functional and fast, several opportunities for enhancement exist:

- **Dynamic Attribute Discovery:** The supported attributes are hardcoded into the indexing and validation logic. This could be improved by using reflection or annotations to dynamically determine valid fields based on the `Pokemon` class, making the system easier to maintain and extend.
- **Memory Usage:** While the indexing strategy improves speed, it does consume more memory due to redundant storage of references across multiple maps. In future, we could explore memory-efficient indexing strategies.
- **Advanced Filtering and Query Chaining:** A potential stretch goal is to support compound filters or query chaining (e.g., `type = "Fire"` AND `attack > 80`). This could be implemented using a query builder pattern or by converting queries into predicates for modular composition.
- **Pre-Sorting and Caching:** For frequently accessed attributes like `speed` or `base_stats`, sorted structures (like `TreeMap` or `ArrayList` with binary search) could be pre-built for faster range slicing and ranking.

Conclusion

This project evolved significantly from a basic dataset exploration tool into a highly optimized, scalable system capable of handling diverse Pokémon queries. Through effective division of labor, each team member contributed uniquely to key components of the backend logic. The resulting implementation balances usability, efficiency, and extensibility and provides a strong foundation for further development, whether through advanced analytics or integration with user-facing interfaces.