Strings in Big Data: Efficient Operations on the Amazon Product Titles Dataset

Evan Beck
ecb9981

Samarth Agarwal
sa9016

Aanand Krishnakumar
ak11138

Akshitha Kumbam
ak11071

Varshitha Reddy Medarametla
vm2663

Rithujaa Rajendrakumar
rr4780

https://github.com/Akshi22/adv_python_final

ABSTRACT

In large-scale datasets, string-based attributes like product titles pose significant challenges due to inconsistency, redundancy, and high dimensionality. This project explores scalable methods for efficiently processing and clustering over 1.4 million Amazon product titles. Our approach is structured in three phases: (1) optimization of core string operations (sorting, filtering, searching, and memory management), (2) implementation of Locality Sensitive Hashing (LSH) using MinHash for approximate Jaccard similarity estimation across millions of pairs, and (3) clustering using TF-IDF and KMeans, both in full and LSH-filtered variations. We evaluate a range of optimization strategies including parallelism, Cython, Numba, and Dask—balancing speed, accuracy, and memory usage. Our results show that using LSH as a pre-filtering mechanism significantly improves clustering quality and runtime, particularly in large datasets. The techniques presented here enable scalable string processing pipelines and offer practical trade-offs for real-world big data applications.

INTRODUCTION

In today's data-driven world, working with textual data—or strings—is an essential yet challenging task, especially at scale. Strings are often lengthy, inconsistent, and riddled with variations that hinder effective analysis and organization. This project focuses on developing efficient methods to process a large-scale dataset of over 1.4 million Amazon product titles, ultimately providing an efficient and scalable way of clustering strings.

PROBLEM STATEMENT

The core challenge lies in handling the complex, messy nature of product title strings in massive datasets. Key issues include:

- **Inconsistent Naming Conventions:** Duplicate or nearly identical products are listed with varying formats.
- **Data Variability:** Misspellings, word order changes, abbreviations, and formatting differences lead to mismatches.
- **Scale:** Manual string matching is impractical for millions of entries.
- **Inefficiency of Traditional Methods:** Brute-force comparisons using Jaccard similarity or edit distance algorithms are computationally expensive and non-scalable.

OBJECTIVE

Our primary goal is to build a robust, scalable framework for performing efficient string operations. Specifically, we aim to:

- **Cluster** similar product titles automatically using advanced string similarity techniques.
- **Compare** traditional and optimized methods to assess trade-offs between performance and accuracy.
- **Optimize** core string operations (sorting, searching, filtering, clustering) for big data environments.
- **Reduce** memory footprint while maintaining or improving processing speed.

ANALYSIS

We designed a 3 phase workflow:

Phase 1: Optimizing Core String Operations

- **Filtering:** Designing filters to quickly eliminate irrelevant or duplicate product titles.
- **Sorting:** Implementing faster and memory-efficient algorithms for ordering large numbers of strings.
- **Searching:** Developing efficient lookup methods for string patterns within the dataset.
- **Memory Reduction:** Exploring data structure optimizations to reduce the memory footprint without compromising processing capability.

Efficient Filtering of Product Titles

Efficient filtering was implemented as a prerequisite to scalable downstream tasks like clustering and deduplication. We supported multiple filtering modes across numeric, categorical, and string-based columns.

Techniques Implemented:

- Dynamic Filtering UI: Enabled via sliders and dropdowns for numeric ranges, categorical selections (e.g., brands), and keyword searches.
- String Filtering: Used Series.str.contains() for keyword match support.
- Categorical Filtering: Handled via isin() for multiselect compatibility.

Backend Implementations Compared:

Implementation	Runtime	Notes
Cython	~0.15s	Fastest due to compiled C loops.
Pandas	~0.23s	Optimized by internal vectorization.
NumPy	~0.25s	Similar to Pandas, minor array overhead.

Dask	~0.30s	Slower due to partitioning/startup costs.
Numba	~140s	Failed for strings; lacks UTF-8 support.

The Key Takeaway is as follows:

1.) Cython provided the best performance, as it compiled filtering logic into C, minimizing Python overhead.
2.) Numba was unsuitable for string filtering, as it does not support Python object types efficiently (e.g., str.contains logic).
3.) Dask's overhead dominated performance due to unnecessary chunking for small filtering tasks.

Sorting Through Large Title Sets

A naive way for candidate selection is by creating a key using multiple features and then sorting the key and creating candidate sets of some predefined sliding window size. In this strategy the bottleneck is the sorting method used. Therefore we explore the performance of optimization techniques on various sorting algorithms.

We found the following results for the sorting functions where we sorted on subset of 100 thousand project titles and ran the sort 100 times and returned the average sort time (the sorts that took more than 10 seconds on a run were Terminated):

Sorting Method	Python	NumPy	Cython	Numba
Quick Sort	0.666448	0.008041	0.940942	0.033257
Merge Sort	0.861635	0.009175	0.912483	0.013596
Heap Sort	1.712279	0.012805	1.099411	0.036767
Selection Sort	Terminated	1.740335	Terminated	5.285227

Timsort	0.235468	0.009055	0.036838	0.081683

The Key Takeaway is as follows:

1.) Utilize numpy whenever there is a native numpy implementation.
2.) Utilize numba when there isn't a native numpy implementation.
3.) Cython can also provide improvements but take note of overheads and recursive elements which can slow down the program rather than provide speedup and try to use it with native c functions for maximum efficiency.

Fast Searching Across Large Title Sets

Search functionality was critical for exploratory analysis and validation. We implemented case-insensitive keyword search over title strings using df[column].str.contains().

Enhancements:

- Case-Insensitive Search: Enabled with case=False to support flexible user queries.
- Na-Safe Handling: Used na=False to avoid match errors due to missing data.

Benchmark Results:

Implementation	Runtime	Notes
Cython	~0.4s	~3× faster than Pandas due to loop compilation.
NumPy	~1.0s	Better than Pandas, but extra array overhead.
Pandas	~1.2s	Baseline vectorized str.contains.
Dask	~3.8s	Overhead from lazy computation and task graph.
Numba	—	No speedup; object dtype not supported.

The Key Takeaway is as follows:

1.) Cython was highly effective due to tight C loop performance.
2.) Dask underperformed since parallelism overhead outweighed benefits for text search on smaller partitions.
3.) Numba could not provide gains here, as JIT-compiling object/string logic remains unsupported.

Memory Reduction Strategy for Large String Datasets

Handling 1.4M product titles posed memory challenges, particularly when combining filtering, clustering, and searching. We implemented memory optimization routines to address this.

Techniques:

- Downcasting Numerics: Used pd.to_numeric(..., downcast=...) for reducing int/float precision.
- Categorical Conversion: Converted low-cardinality object columns (e.g., brand names) to category dtype.
- String Optimization: Interned repetitive strings using astype("category") where appropriate.

Memory Usage Impact:

Step	Memory (MB)	Reduction
Original CSV	~580 MB	—
After Numeric Downcasting	~340 MB	~41%
After Categorical Encoding	~270 MB	~53% total

The Key Takeaway is as follows:

1.) Lower memory usage improved performance across all tasks, especially Dask (fewer partitions), and clustering (less RAM pressure)
2.) Enables larger subsets to be processed locally without GPU/cluster scaling.

Conclusion for Phase 1

These optimizations demonstrate how traditional Python-based workflows can be significantly accelerated and made more efficient through selective use of compiled extensions (Cython), dtype tuning, and careful memory management.

Filtering and searching operations on massive string datasets benefit most from compiled code (Cython), while Numba and Dask show limited utility for string-heavy tasks due to lack of object-type and string-specific optimizations.

Phase 2: Locality Sensitive Hashing (LSH)

Efficient similarity detection was critical for reducing the dataset size and identifying promising product title pairs before clustering. This phase focused on implementing MinHash-based Locality Sensitive Hashing (LSH) with several optimizations to improve scalability, runtime, and reliability.

1. Approximating Jaccard Similarity at Scale

At the heart of this phase is the need to compare millions of product titles to find near-duplicates or similar items. A natural metric for this task is Jaccard similarity, which measures the overlap between two sets of tokens (e.g., words in titles). However, computing the exact Jaccard similarity for all 1.4 million rows, which is approx 10^12 possible title pairs, would require prohibitive computational resources.

To make this feasible, we used MinHash, a probabilistic hashing technique that generates a fixed-length signature for each title. The probability that two MinHash signatures match at any index approximates their true Jaccard similarity. These signatures are then passed through Locality Sensitive Hashing (LSH), which clusters similar signatures into the same buckets—dramatically reducing the number of candidate pairs needing full comparison.

This method makes it possible to efficiently retrieve likely-similar title pairs, enabling scalable duplicate detection and clustering across millions of strings.

2. Sequential MinHash + LSH (Baseline):

A basic implementation using the datasketch library to compute MinHash signatures and insert them into an LSH index one at a time. This method was computationally intensive and completed in approximately 13 minutes on a MacBook Pro M3. It proved infeasible for large-scale datasets.

3. Parallelized MinHash Signature Generation:

Using Python's multiprocessing module, we parallelized signature generation across 12 CPU cores. Although LSH indexing and querying remained serial, the performance gains were substantial: over 22 million candidate pairs were found in ~245 seconds, striking a good balance between speed and resource usage.

4. Stopword Removal (Rejected):

We tested removing common English stopwords prior to signature generation to improve runtime and memory usage. Although runtime improved slightly (~290 seconds), the quality of results degraded due to many informative tokens being dropped—resulting in excessive false positives.

5. Numba-Based Custom Hashing (Rejected):

We implemented a JIT-compiled hashing function using Numba to accelerate MinHash signature creation. However, multiprocessing overhead and poor integration with the datasketch LSH class caused overall performance to decline, with runtimes over 360 seconds and no improvements in output quality.

6. Cython-Based LSH (Abandoned):

We reimplemented the LSH indexing step in Cython for theoretical speedups. However, the tuning was highly unstable: small changes to the number of bands could result in either 0 matches or 150+ million, making this approach unsuitable for production use.

Final Implementation

The best-performing pipeline used parallelized MinHash + LSH, without stopword removal, Numba hashing, or Cython acceleration. This version reliably identified high-similarity title pairs with acceptable runtime and minimal tuning requirements.

Method/Approach	Runtime	Pairs Found	Outcome / Notes
Jaccard Similarity (True)	~Years	~10^12	Infeasible to compute directly on full dataset
Sequential (Baseline)	~780 s	~1.5 million	Too slow for production
Parallel MinHash (Final)	~245 s	~22 million	Best tradeoff of speed and reliability
Stopword Removal	~290 s	~26 million	↓ Precision; increased false positives
Numba Custom Hashing	~360 s	~23 million	No performance gain due to multiprocessing overhead

Cython-Based LSH	Variable	0 or 150+ million	Unstable tuning and extreme result sensitivity

This phase allowed us to reduce the dataset from over one million titles to a focused subset of high-probability matches.

One main part of this project is the implementation of LSH to clustering, which effectively reduces the size of the set needed to cluster through and even has other benefits. Please see the later Part of phase 3.

Conclusion for Phase 2

MinHash and LSH allowed us to efficiently approximate Jaccard similarity across millions of product titles, avoiding the infeasible cost of exact pairwise comparisons. Parallelizing MinHash signature generation provided the best balance of speed and reliability, identifying over 22 million high-confidence candidate pairs in under 5 minutes. While other variants offered marginal gains or theoretical speedups, they introduced trade-offs in precision or robustness. This phase effectively reduced the dataset to a focused subset, enabling faster and more accurate clustering downstream.

Phase 3: Clustering Product Titles – Two Approaches

To address the problem of inconsistent and redundant product titles in large catalogs, we explored clustering techniques to group similar items effectively. Our focus was on testing different implementations of a baseline method using TF-IDF vectors combined with K-Means clustering.

Approach 1: TFIDF

We began with a **sample of 50,000 product titles** from the full Amazon dataset. The steps included:

1. **Text Vectorization:**
   Each product title was transformed into a TF-IDF vector with a vocabulary size limited to 1,000 terms. Common stop words were removed to focus on meaningful content.

2. **Clustering Techniques:**
   We applied K-Means clustering using four different implementations:

- **Pure Python:** A basic, loop-based implementation for educational insight.
- **NumPy:** Leveraged vectorized operations for faster distance calculations.
- **Numba:** A JIT-compiled version of the loop-based code for performance optimization.
- **Scikit-learn:** A highly optimized library-based implementation used as the benchmark.

Findings and Performance Analysis

All methods successfully clustered the 50,000 × 1,000 TF-IDF matrix into **k = 10 clusters**, but with significant variation in runtime performance:

Implementation	Runtime (s)	Speedup vs. Pure Python
Pure Python	2363.261	1× (baseline)
NumPy	34.468	68.6×
Numba	8.995	262.7×
scikit-learn	2.699	875.6×

The Key Takeaway is as follows:

1.) **Scikit-learn** emerged as the fastest and most scalable approach, completing the task in under 3 seconds
2.) Both **NumPy** and **Numba** versions provided valuable insights into the inner workings of the algorithm while still offering substantial performance improvements over the baseline.
3.) One caveat of the TF-IDF approach was that, although it ensured **full dataset coverage**, it occasionally clustered **semantically unrelated titles**, highlighting the need for deeper semantic similarity measures in future work.

Approach 2: LSH-Enhanced TF-IDF

To improve both efficiency and interpretability, we introduced an LSH-based pre-filtering step before applying traditional TF-IDF + KMeans clustering. The motivation was twofold: (1) reduce the size of the input space to clustering, and (2) restrict clusters to semantically similar product titles.

The pipeline consisted of:

1. **MinHash + LSH filtering:**
   We used a thresholded LSH (Jaccard ≥ 0.8) to identify product title pairs likely to be near-duplicates. This drastically reduced the dataset size by isolating a small subset of titles worth clustering.

2. **TF-IDF Vectorization:**
   Only the unique titles from LSH were vectorized using TF-IDF, with a vocabulary size of 5,000 terms.

3. **KMeans Clustering:**
   We applied KMeans with k = 10 clusters to group similar product titles. Cluster assignments were visualized using PCA, and word distributions were analyzed via word counts.

Results

Step	Value / Outcome
Titles Clustered	~45,823 titles (from ~1.1M input)
LSH Candidate Pairs	22,185,000 pairs
LSH Filtering Time	~104 seconds
PCA Plot	Saved to pca_clusters_lsh_filtered.png

Word Counts	Saved to cluster_word_counts.txt

We observed cleaner and more compact clusters, with dominant product-related terms consistently grouping within clusters. This was especially evident in domains like “school supplies,” “toys,” and “kitchen tools,” where minor naming variations typically obscure similarity in raw TF-IDF clustering.

When LSH-Enhanced TF-IDF is Useful

This hybrid method is ideal when:

- The dataset contains millions of short, noisy strings (e.g., product names, search queries).
- There is a high likelihood of duplicated or near-duplicated items.
- Downstream clustering or manual review is too costly to perform over the full dataset.
- If the data is truly huge, you really have no choice but to use LSH or some filtering mechanism.

Benefits & Trade-offs:

Benefit	Trade-off
Faster clustering for large data	Requires tuning LSH thresholds and band size… LSH can also take a while but is way faster and makes a large difference as data size increases
Improved cluster purity	Some semantic diversity may be filtered out
Less noise in PCA & visualization	Risk of missing sparse but meaningful titles

Smaller memory and compute footprint	Potential for under-clustering if LSH too strict

Conclusion for Phase 3

LSH-enhanced TF-IDF strikes a practical balance between precision and scalability, enabling efficient clustering of massive string datasets. By first narrowing focus to only highly similar titles, we preserve semantic quality while drastically reducing computational overhead. It is important to note that one method is not better than the other. If you have small data, using LSH might not be worth it because it requires more time than if you just use TF-IDF and scikitlearn. This method makes a huge difference though when the data size scales up. Additionally, one might choose to always use LSH as a filtering mechanism if they wish to have more focused clusters.

CONCLUSION

This project demonstrated that efficient handling of large-scale string data requires both algorithmic innovation and practical system-level optimizations. From filtering and memory tuning in Phase 1, to scalable similarity detection with MinHash and LSH in Phase 2, and finally to clustering in Phase 3, we showed how each phase incrementally improved the performance and interpretability of downstream tasks. Our experiments confirm that combining approximate techniques like LSH with traditional models like TF-IDF and KMeans yields meaningful improvements in speed and semantic coherence, especially at scale. Ultimately, this multi-phase pipeline provides a generalizable framework for tackling messy, redundant, and high-volume string datasets in domains ranging from e-commerce to search indexing.