NYU

Working with Strings in Big Data

Performing Efficient Operations on Amazon Product Titles Dataset w 1.4 Million rows.

By: Evan Beck, Samarth Agarwal, Rithujaa Rajendrakumar, Varshitha Reddy Medarametla, Akshitha Kumbam, Aanand Krishnan

What are we Trying to Solve?

We want to be able to work efficiently with STRINGS! Such as: Filter, Search, Cluster

- **What is the problem?**
  - Amazon product titles are long, messy, and hard to process when you have loads of data.
  - Product catalogs contain duplicate or nearly identical entries due to inconsistent naming.
  - Misspellings, word order differences, abbreviations, and formatting variations cause identical products to appear different.
  - Manually matching entries is impossible for large datasets.
  - Traditional string comparison techniques (e.g., brute-force Jaccard or edit distance) are too slow and inefficient at scale.

NYU

Our Solution

- Automatically group (cluster) similar product titles based on more specific similarity.

- Compare baseline and optimized clustering methods to evaluate trade-offs in speed and accuracy.

- Design a scalable solution that works efficiently even on large product datasets.

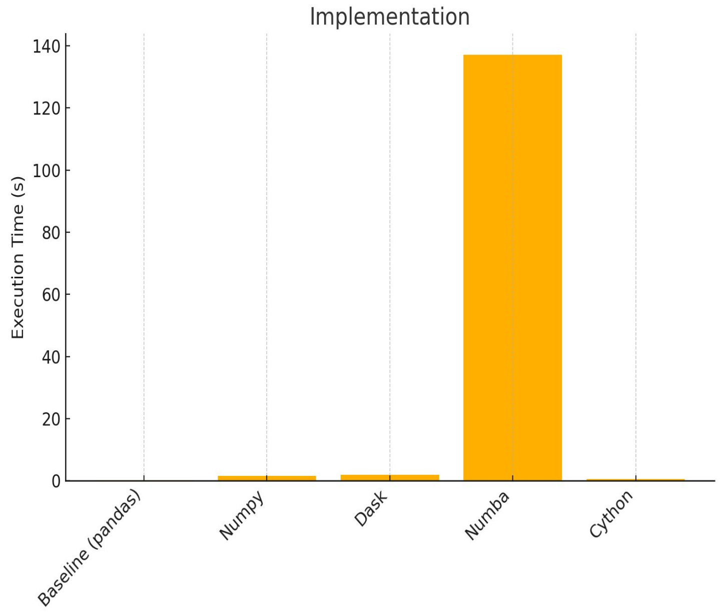
- Explore fundamental/rudimentary operations with various optimization techniques on strings

Phase 1: Optimizing Core Operations!

Sorting, Filtering, Searching & Memory Reduction was split amongst the team!

Filtering!

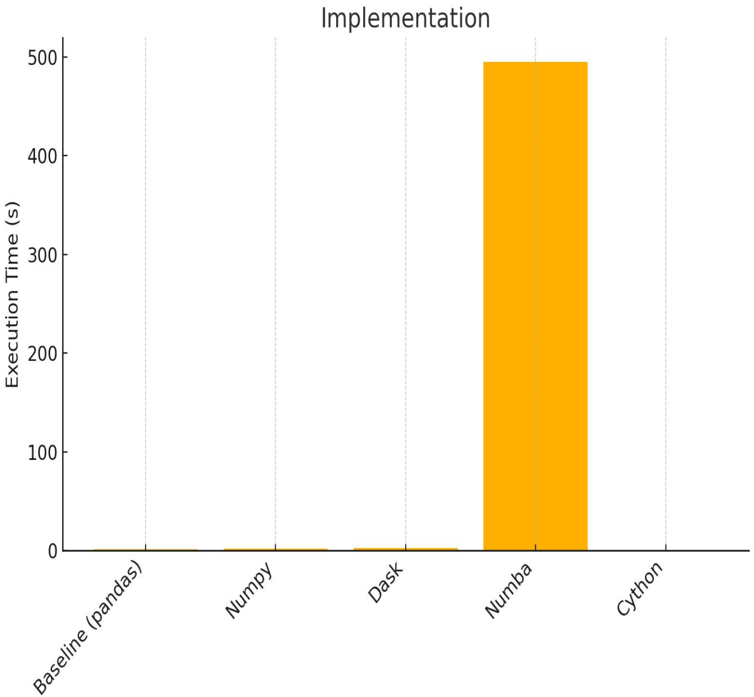
- Added **dynamic filtering** for numeric, string, and categorical columns.
- Numeric filtering included: Simple conditions (e.g., >, <, ==) & **Range sliders** for selecting min/max
- **Categorical : Dropdown/multiselect** widgets (isin()) & String filtering supported **keyword match.**
- **Cython** (~0.15s) was fastest via compiled C loops.
- **Pandas** (~0.23s) and **Numpy** (~0.25s) were close due to vectorization.
- **Dask** (~0.30s) slowed by overhead.
- **Numba** (~140s) failed due to string handling issues.



NYU

Searching!

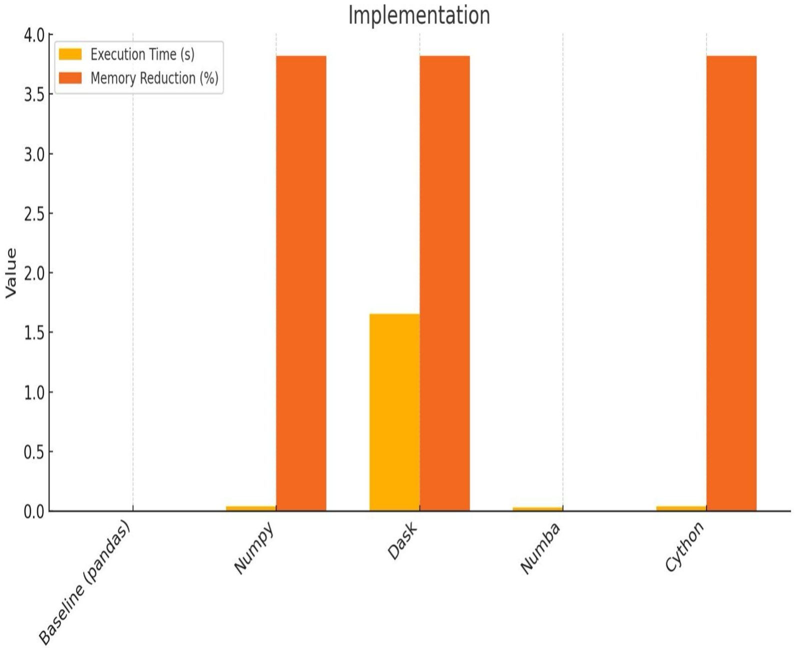
- Implemented keyword search using df[column].str.contains()
- Enabled **case-insensitive** matching (case=False)
- Handled missing values safely (na=False)
- **Cython** (~0.4s) was >3× faster than Pandas (~1.2s).
- **Numpy** (~1.0s) improved over Pandas but had array overhead.
- **Dask** (~3.8s) was slow due to startup and partitioning costs.
- **Numba** showed no major speedup due to object dtype issues.



NYU

Reducing Memory Usage

- Downcasted data types: int64 → int16, float64 → float32 where safe
- Converted string columns to category if they had **low cardinality**
- Checked unique-to-total ratio before converting
- Cython was the fastest efficient method (~3× faster than Pandas).
- Dask was slow due to its partitioning and initialization overhead.
- Numba didn't improve memory due to object dtype limitations (not suited for string-heavy data)



NYU

Blocking

1. A naive implementation to get candidates by using sliding window strategy
2. Requires creating a key based off a few important features. Subsequently, sorting and considering a window of some size (decided by the user) to get relevant candidates.
3. Runtime of operation depends largely on the runtime of the sorting algorithm

Optimizing sorting functions

Sorting Algorithm	How It Sorts	NumPy	Cython	Numba
Quick Sort	Recursive partitioning around pivots.	Calls fast built-in numpy.sort('quicksort'); not customizable.	Compiles recursive partition logic into C; faster pivoting and swaps.	Compiles recursion + partition into machine code; needs numeric types.
Merge Sort	Recursive split and merge.	Uses numpy.sort('mergesort'); fast stable sort, but limited customization.	Compiles split-merge steps into C; speeds up merging especially.	Compiles recursion and merging into optimized machine code.
Heap Sort	Build max/min heap, extract elements.	Uses numpy.sort('heapsort'); fast but generic.	Compiles heapify and swap loops; better heap maintenance performance.	Compiles heapify recursion into machine code; big speedup for large heaps.
Selection Sort	Nested loops; find min/max and swap to front.	Slightly faster array scanning with NumPy functions, but still slow.	Compiles double loops into C; minimizes loop overhead greatly.	Compiles selection loops into machine code; faster find + swap.
Timsort	Hybrid insertion/merge optimized for real-world data.	Uses numpy.sort('stable') (Timsort); already optimized.	Just calls Python's built-in Timsort; no extra benefit.	No real gain — just wraps Python's sort inside Numba, still fast.

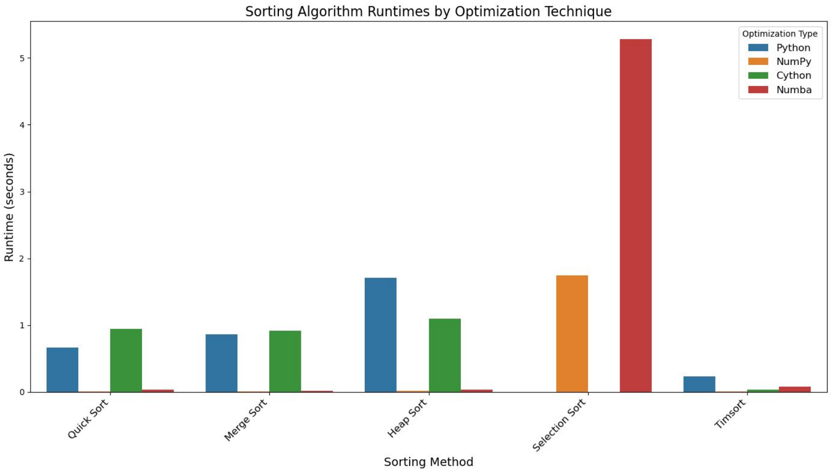
NYU

Sorting Results

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

NYU

Summary



Comparison:

- Use Numpy whenever there is a native Numpy implementation of a Python function
- If not for a native numpy implementation try Numba, regardless of recursive or nested elements
- Use Cython when a native C function can be used and you are not working with recursive elements or memory overheads

NYU

Phase 2: Optimizing Similarity Search with MinHash + LSH

Goal:

Efficiently identify similar product title pairs from **~1.4 million titles** without computing 1.9 trillion comparisons. It would be amazing if we could optimize this! This preprocessing is useful for things like clustering!

Optimizations and Experiments Tried (Advanced Python Techniques):

- **Regular Jaccard (Baseline):**
  Attempted full pairwise Jaccard similarity using set intersection and union.
  → Did not finish overnight on 100k × 100k comparisons — clearly infeasible for scaling.

- **Numba-Accelerated Hashing:**
  Implemented a custom Numba-based hash function for signature updates.
- Speed gains were minimal and negated by multiprocessing overhead.

NYU

- **Stopword Removal as a means of memory reduction and reducing false positives (had opposite effect):**
  Removed English stopwords prior to signature generation.
  → Increased false positives and reduced match quality due to over-simplification.

- **Cython for LSH Indexing:**
  Rewrote the LSH banding logic in Cython.
  → Performance was unstable; band tuning led to extreme outcomes (e.g., over 150 million pairs).

Final Method: Parallel MinHash with Minimal Preprocessing

- The most effective setup was surprisingly simple:
  - No stopword removal
  - No custom hashing
  - No special encodings

NYU

- Each title was tokenized using **basic whitespace splitting**.
  This avoided over engineering and preserved important context in the strings.

- **Parallelized MinHash signature generation** using multiprocessing.Pool across **12 CPU cores**.
  This directly targeted the bottleneck — signature computation — and cut runtime from ~13 minutes to **~ 4 minutes.**

- **Result:**
  - Identified **~22 million meaningful candidate pairs**
  - Output was clean, interpretable, and scaled easily to over a million titles
  - No extra overhead or instability from tuning external tools (Numba, Cython, etc.)

Optimization Attempt Results

Summary:

Parallel MinHash worked best.
It worked best because signature generation was bottleneck.

Stopword Removal did not work. This is because it led to overly generic strings and or shorter strings. Because of this it increased the amount of matches needed to compare and reduced quality of similarity measure

Numba made things worse! The goal was to speed up individual minhash updates using JIT hash function. Minor local gain. This was offset by the overhead combined with MP. Numba compiles functions when ran Multi Process led to multi compilation

Cython for LSH to speed up banding and LSH indexing. Rewrote LSH logic in cython. Did not work. It was hard to tune, results were unstable, returned up to over 150 M moisy pairs

NYU

Baseline: Sequential MinHash	Standard datasketch MinHash + LSH; one title at a time	~13 minutes	Too slow to scale; infeasible for 1.4M titles
Regular Jaccard (Baseline)	Full pairwise Jaccard comparisons (not MinHash-based)	Infeasible	Tried on 100k × 100k — didn't finish overnight; completely unscalable
Stopword Removal to reduce memory WITH MP.	Removed common English stopwords before hashing. MP on 12 cores	~290 seconds	Introduced false positives and noise; slightly slower; not worth it
Numba-Accelerated Hashing WITH MP.	Used @njit compiled custom hash function with MP	~363 seconds	Local hash speed improved, but gains offset by multiprocessing and LSH overhead
Cython for LSH WITH MP	Rewrote LSH indexing in Cython and manually tuned banding with MP	Variable / unstable	Hard to tune; up to 150M matches; inaccurate or excessive; ultimately abandoned
Simple MP with no stopword removal or added ops (Final Method)	Used multiprocessing. Pool across 12 cores	~245 seconds	Balanced speed and simplicity; ~22M clean pairs; scalable and interpretable

Phase 3: Clustering Product Titles – Two Ways

1. TF-IDF + K-Means Clustering (Baseline Approach)

Steps:

- Sampled **50,000 titles** from Amazon product dataset
- Converted titles to TF-IDF vectors (limited to **1,000 terms**, stopwords removed)
- Applied **four KMeans implementations**:
  - **(a)** Pure Python: basic loop-based logic
  - **(b)** NumPy: vectorized distance calculations
  - **(c)** Numba: compiled loop-based version
  - **(d)** scikit-learn: fast, library-based solution

Findings:

- scikit-learn was fastest
- NumPy/Numba offered transparency + performance
- TF-IDF clusters had full coverage but sometimes grouped unrelated titles

NYU

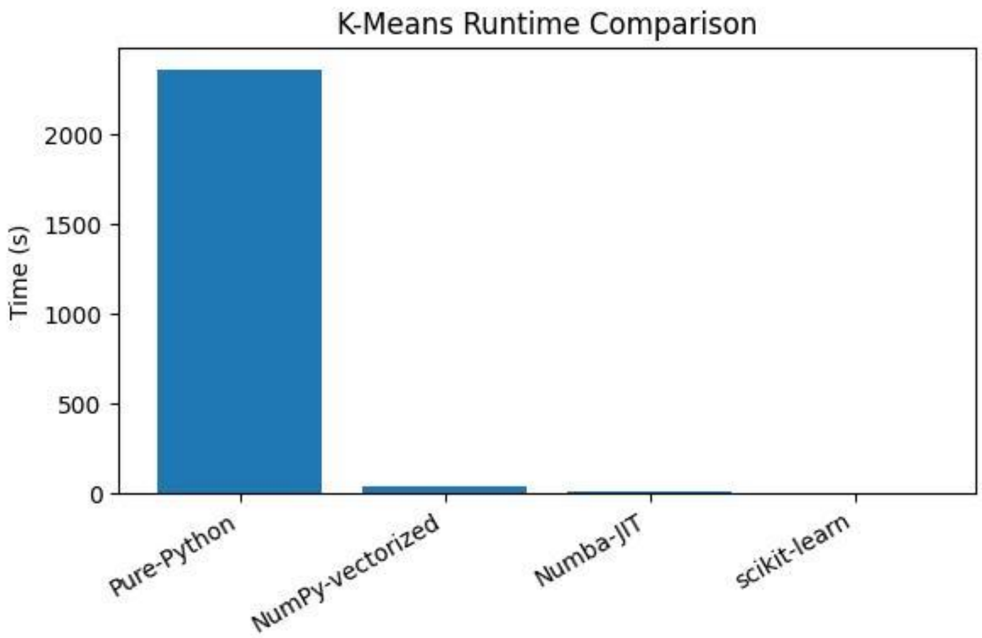
Clustering Similar Product Titles
Clustering 50000×1000 TF-IDF vectors into k=10

Run-time for each implementation

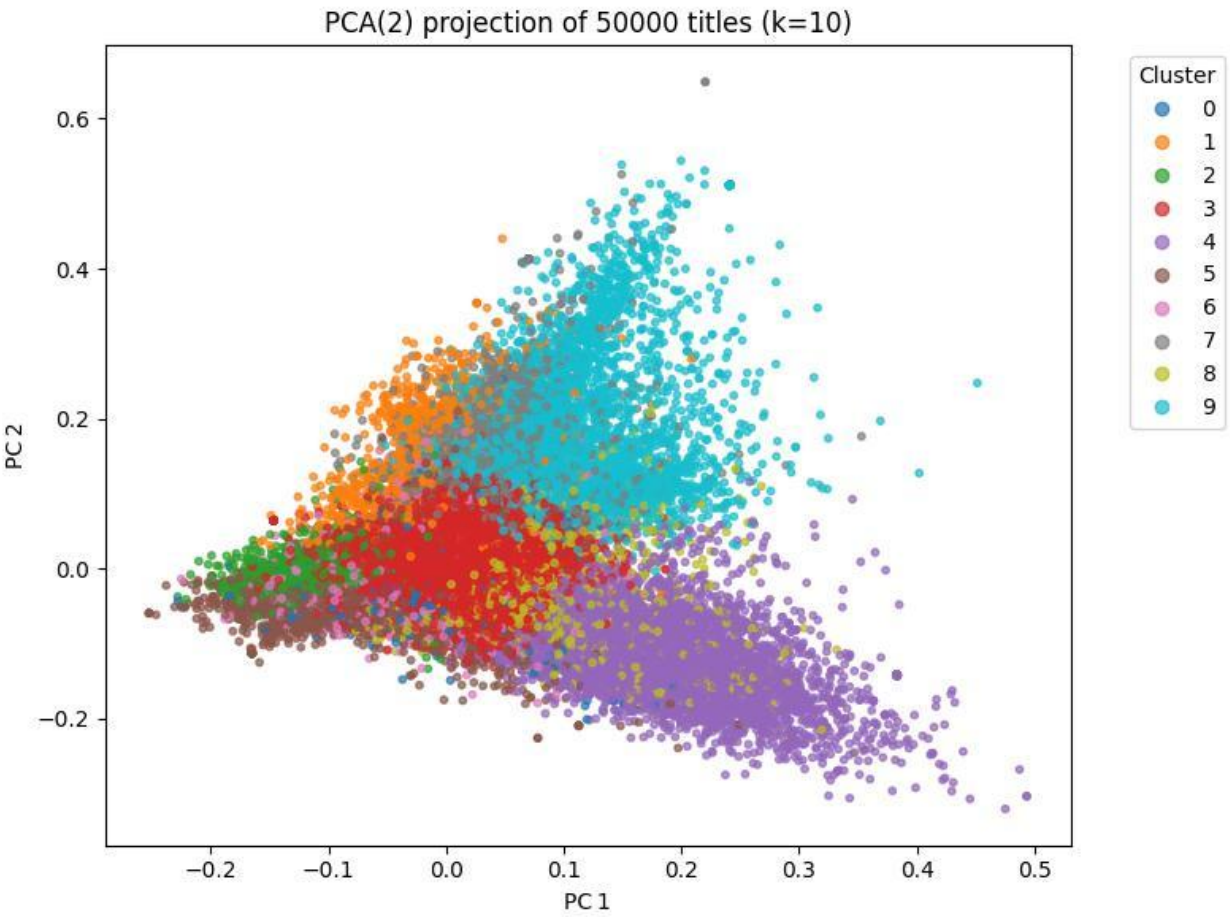
- Pure-Python : **2363.261s**
- NumPy-vectorized : 34.468s
- Numba-JIT : 8.995s
- scikit-learn : **2.699s**

Speed ups

NumPy-vectorized : 68.6x
Numba-JIT : 262.7x
scikit-learn : 875.6x



NYU


PCA(2) projection of 50000 titles (k=10)
Cluster
0
1
2
3
4
5
6
7
8
9
PC 2
0.6
0.4
0.2
0.0
−0.2
PC 1
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
0.5

2. LSH-Filtered Clustering (Optimized Approach)

Steps:

- Used full set of **~1.4 million product titles**
- Applied MinHash + LSH to extract **~22 million strong candidate pairs!**
- Only clustered through these items!!! Significantly reducing the set to be searched

Findings:

- Clusters were smaller, tighter, and more interpretable!
- Filtering reduced noise and computation
- Tradeoff: lower coverage, but higher cluster quality

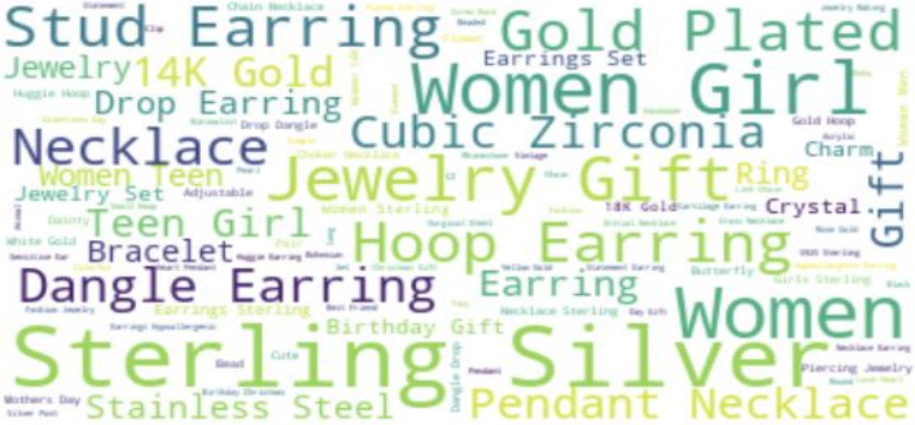
Takeaway:

TF-IDF + KMeans offers full-dataset clustering, however, requires brute-force.
LSH-based clustering targets high-confidence pairs and yields cleaner, more meaningful groupings.

Visualizing the clustering results!

Clustering with only TFIDF

Clustering with TFIDF using LSH



TF-IDF w kmeans (no lsh preprocessing) clusters everything from the sample and is faster on smaller datasets, but results can be noisy and harder to interpret. Mixed results: demographics / clothing / jewelry (?)... Words have more variation of size / importance / frequency in cluster!

LSH-based then TFIDF w kmeans filters to only highly similar items, producing cleaner, more specific clusters — ideal for large, messy datasets.

Optimizing LSH (e.g., with multiprocessing) makes it scalable and practical, unlocking accurate clustering at scale without compromising performance and arguably increasing the performance.

NYU

Conclusion!!!

We set out to solve a key problem: how to efficiently process large-scale string data, which is often messy and computationally expensive.

We focused on how to efficiently process large-scale string data, which is messy and slow to work with.

We first optimized core tasks like sorting and filtering. Simple tools like NumPy and multiprocessing gave solid speedups — and sometimes, it's hard to beat what's already optimized.

Then we compared two clustering approaches:

- **TF-IDF + KMeans**: Fast and easy to apply, but clusters everything — including unrelated or noisy items.

- **Using TF-IDF with LSH FIRST**: Slower to set up, but filters for high-similarity items first. This gave us tighter, cleaner clusters at **SCALE**.

So which is better?
It depends. TF-IDF is broad but noisy. LSH is focused and scalable.
In practice, combining both gave us the best results: LSH to narrow the data, then TF-IDF to cluster meaningfully.