Identifying Frequent Items in Data Streams

Rodrigo Abreu
Mestrado em Engenharia Informática

***Abstract* – This study evaluates three algorithmic strategies for identifying frequent items in data streams, using the `release_year` attribute from the Netflix Movies and TV Shows dataset as a case study. The performance of an Exact Counter is compared against two approximate methods: a Fixed Probability Counter ($p = 0.5$) and the Frequent-Count (Misra-Gries) streaming algorithm.**

Experimental results reveal that for low-cardinality datasets ($N \approx 74$), the Frequent-Count algorithm exhibits a saturation behavior where estimation error drops to zero once the buffer size exceeds the number of unique items. While it theoretically offers bounded memory, practical implementation overheads in Python resulted in memory usage comparable to the baseline. Conversely, the Fixed Probability Counter achieved a 25% reduction in execution time and demonstrated excellent rank preservation (Spearman $\rho \approx 0.98$), despite a high mean relative error of approximately 29%. The study concludes that while Exact Counting remains optimal for limited-domain metadata, probabilistic methods offer viable trade-offs for latency-critical ranking tasks, and streaming algorithms are best reserved for high-cardinality scenarios where memory constraints are severe.

***Keywords* – Exact Counter, Fixed Probability Counter, Frequent-Count, Misra-Gries**

I. Introduction

This report evaluates and compares three algorithmic strategies for identifying frequent items, utilizing the `release_year` attribute from the Netflix Movies and TV Shows dataset [1] as a case study. The primary objective is to assess the performance trade-offs between an Exact Counter baseline and two approximate methods: a Fixed Probability Counter with a sampling rate of 50% and the Frequent-Count (Misra-Gries) streaming algorithm. Specifically, the study analyzes how bounded memory constraints and probabilistic sampling affect estimation accuracy, ranking preservation, and computational efficiency compared to the ground truth established by exact counting.

The document is organized to provide a systematic analysis of these methods. It begins by defining the theoretical foundations and formal complexity of the implemented algorithms, followed by a description of the experimental methodology, dataset preprocessing, and evaluation metrics. Empirical results are then presented to quantify errors and resource consumption, specifically highlighting the saturation behavior observed when streaming algorithms are applied to low-cardinality data. The analysis concludes with a synthesis of these findings and recommendations for selecting appropriate algorithms based on specific application requirements and memory constraints.

II. Algorithms

This study evaluates three distinct algorithmic approaches for the frequency analysis of data streams. These range from exact methods, which require linear space relative to the number of unique items, to approximate and streaming algorithms that trade accuracy for significant reductions in memory consumption.

A. Exact Counting

The Exact Counting algorithm serves as the baseline for performance evaluation. It maintains a complete frequency distribution of the data stream by utilizing a hash map (dictionary) data structure, implemented via the Python `collections` library [2]. For every item x observed in the stream, the algorithm checks if x exists in the map. If it does, its counter is incremented; otherwise, a new entry is created with an initial count of 1. This approach guarantees 0% error but scales linearly in space with the number of unique items, making it potentially infeasible for massive streams with high cardinality.

Algorithm 1 Exact Counting

```
Input: Stream $S$
Output: Frequency Map $C$
$C \leftarrow$ Empty Hash Map  for each item $x \in S$ do
    if $x \in C$ then
        $C[x] \leftarrow C[x] + 1$
    end
    else
        $C[x] \leftarrow 1$
    end
end
return $C$
```

B. Approximate Counter: Fixed Probability Sampling

To reduce the memory footprint, we implemented a Fixed Probability Counter (Bernoulli Sampling). Instead of processing every item deterministically, this algorithm samples items from the stream with a fixed probability p (e.g., $p = 0.5$).

When an item x arrives, a random number $r \in [0, 1)$ is generated. The item is counted only if $r < p$. The final estimated count for any item is calculated using the Horvitz-Thompson estimator [3], $\hat{c}_x = c_x/p$, where c_x is the number of sampled occurrences. This method provides an unbiased estimator of the true frequency but introduces variance [4]. The memory saving arises because items with low frequencies are statistically likely to be skipped entirely, thus reducing the number of entries in the hash map.

Algorithm 2 Fixed Probability Counter

```
Input: Stream $S$, Sampling Probability $p$, Random Seed $seed$
Output: Map of estimated counts $E$
$Samples \leftarrow$ Empty Hash Map
Initialize RNG with $seed$
for each item $x \in S$ do
    $r \leftarrow \text{Random}(0, 1)$
    if $r < p$ then
        if $x \in Samples$ then
            $Samples[x] \leftarrow Samples[x] + 1$
        end
        else
            $Samples[x] \leftarrow 1$
        end
    end
end
$E \leftarrow$ Empty Hash Map
for each item $x \in Samples$ do
    $E[x] \leftarrow Samples[x]/p$
end
return $E$
```

C. Frequent-Count (Misra-Gries)

The Frequent-Count algorithm, proposed by Misra and Gries [5], is a deterministic streaming algorithm designed to identify heavy hitters (items that appear more than N/k times) using bounded memory.

The algorithm maintains a map of at most $k-1$ counters. When an item x arrives, it is incremented if present. If absent and the map is not full, it is added. However, if the map is full, a decrement-all operation is triggered: every counter in the map is decremented by 1, and any item whose count reaches 0 is evicted. This process effectively cancels out k distinct items simultaneously, ensuring that frequent items remain in the map while rare items are evicted. A second pass over the stream (or a separate tracking structure) is typically required if exact counts are needed, but for this study, we analyze the estimated counts directly produced by the single pass.

Algorithm 3 Frequent-Count (Misra-Gries)

```
Input: Stream $S$, Buffer parameter $k$
Output: Map of heavy hitters $C$
$C \leftarrow$ Empty Hash Map
for each item $x \in S$ do
    if $x \in C$ then
        $C[x] \leftarrow C[x] + 1$
    end
    else if $|C| < k - 1$ then
        $C[x] \leftarrow 1$
    end
    else
        // Buffer full: decrement all counters
        $ToRem \leftarrow \emptyset$
        for each key $y \in C$ do
            $C[y] \leftarrow C[y] - 1$
            if $C[y] = 0$ then
                $ToRem \leftarrow ToRem \cup \{y\}$
            end
        end
        for each key $y \in ToRem$ do
            Remove $y$ from $C$
        end
    end
end
return $C$
```

III. Formal Analysis

A. Exact Counting

The Exact Counting algorithm processes a data stream of length M containing N unique items. For each incoming item x, the algorithm performs a hash map lookup and, conditionally, an insertion or update operation. Assuming a collision-resistant hash function, these operations take $O(1)$ time on average. Therefore, the total time complexity for processing the stream is derived as:

$$T(M) = \sum_{i=1}^{M} O(1) = O(M)$$

The space complexity is strictly determined by the number of unique items stored. In the worst-case scenario, where every item in the stream is unique ($N = M$), the space requirement scales linearly:

$$S(N) = O(N)$$

This linear space scaling ($O(N)$) represents the primary memory bottleneck that necessitates the use of approximate methods for massive datasets.

B. Approximate Counter (Fixed Probability)

The computational cost of the Fixed Probability Counter remains linear with respect to the stream size. For each item, the algorithm performs a random number generation and a comparison, which are $O(1)$ operations. Thus, the time complexity remains:

$$T(M) = O(M)$$

The space complexity, however, becomes probabilistic. Let N be the number of unique items in the stream, and let f_i denote the true frequency of the i-th item. An item is stored in the map only if at least one of its f_i occurrences is sampled. The probability that an item with frequency f_i is *never* sampled (and thus not stored) is $(1-p)^{f_i}$.

The expected number of stored items, $E[N_{stored}]$, can be expressed as the sum of the probabilities that each unique item is stored:

$$E[N_{stored}] = \sum_{i=1}^{N} \left(1-(1-p)^{f_i}\right)$$

This formula reveals the algorithm's behavior: for rare items ($f_i = 1$), the probability of storage is exactly p, whereas for frequent items (large f_i), the term $(1-p)^{f_i}$ approaches 0, making the storage probability approach 1. Consequently, for sparse data distributions, the space complexity is reduced by a factor of p:

$$S(N,p) \approx O(p \cdot N)$$

C. Frequent-Count (Misra-Gries)

The Misra-Gries algorithm is defined by the parameter k, which strictly bounds the memory usage. The data structure maintains at most $k-1$ counters. Thus, the space complexity is deterministic and independent of the stream size M or cardinality N:

$$S(k) = O(k)$$

The time complexity analysis is more nuanced due to the decrement-all operation. In the best-case scenario, characterized by highly skewed distributions, the buffer rarely fills, allowing operations to complete in $O(1)$ time. Conversely, in the worst-case scenario, typified by uniform distributions, the buffer fills frequently. When the buffer reaches capacity, the algorithm must iterate over all $k-1$ counters to decrement them, an operation requiring $O(k)$ time. Thus, the worst-case time complexity for processing the entire stream is:

$$T(M,k) = O(M \cdot k)$$

However, it can be mathematically proven that the expensive decrement operation occurs at most M/k times across the entire stream. Therefore, the amortized cost per item remains $O(1)$, yielding a total amortized time complexity of:

$$T_{\text{amortized}}(M) = O(M)$$

IV. Experimental Methodology

This chapter details the dataset characteristics, the preprocessing pipeline, and the evaluation framework used to assess the performance of the implemented algorithms. All experiments were conducted in Python 3, simulating a streaming environment where items are processed sequentially.

A. Dataset Characterization

The experiments were performed using the `Netflix Movies and TV Shows` dataset [1], as assigned for this project. The specific attribute selected for frequency analysis is the **release_year**. This attribute represents the year in which a movie or TV show was released. Unlike unique identifiers, release years are categorical values that naturally exhibit a non-uniform frequency distribution, certain years appear significantly more often than others. This skew makes the dataset suitable for evaluating Heavy Hitter algorithms.

B. Data Preprocessing

To ensure a consistent input stream for all algorithms, a preprocessing pipeline was implemented using the *pandas* library. The `release_year` column was first isolated from the dataset, and any non-numeric values were coerced to `NaN` and dropped. The remaining valid entries were cast to integers to ensure discrete, hashable items. Finally, the cleaned data was converted into a linear list to simulate a data stream, preserving the original order of appearance. This stream served as the identical input S for the Exact, Fixed Probability, and Frequent-Count algorithms.

C. Evaluation Framework and Metrics

To quantify the trade-offs between accuracy and resource usage, we employed several metrics comparing the approximate results against the ground truth established by the Exact Counter.

Accuracy Metrics We calculated the *Mean Absolute Error (MAE)* to measure the average magnitude of count errors, and the *Mean Relative Error (MRE)* to express the error as a percentage of the true count. Additionally, the *Spearman Rank Correlation (ρ)* was used to evaluate how well the algorithms preserved the relative ordering of items, where a value close to 1.0 indicates that the most frequent items were correctly identified by rank, even if absolute counts differed.

Computational Metrics: The memory footprint (in Kilobytes) of the tracking data structures was measured using a deep traversal of the object graph via `sys.getsizeof`. Execution time was recorded using `time.perf_counter()` to capture the wall-clock time required to process the entire stream.

D. Experimental Setup

All computational tests were performed on a Lenovo LOQ Essential 15IAX9E laptop equipped with a 12th generation Intel Core i7 CPU and 16 GB of RAM, running Ubuntu 24.04.

The algorithms were configured to analyze behavior under different constraints. The **Exact Counter** was run once to establish the ground truth. The **Fixed Probability Counter** used a sampling probability of $p = 0.5$ and was repeated over 20 trials to account for stochastic variance. The **Frequent-Count (Misra-Gries)** algorithm was tested for a range of top-n targets ($\{5, 10, \ldots, 50\}$). For each n, the buffer size was set to $k = 2 \times n$, ensuring the buffer was sufficiently larger than the target while strictly bounding memory.

V. Results and Analysis

This chapter presents the experimental results obtained from the three algorithmic approaches. The analysis focuses on accuracy, memory efficiency, and the trade-offs inherent to each method, substantiated by the data collected from the `netflix_titles.csv` dataset.

A. *Exact Counter Baseline*

The Exact Counter algorithm successfully processed the entire stream of Netflix release years, establishing the ground truth for all subsequent comparisons.

A.1 Frequency Distribution

The analysis reveals that the dataset exhibits a highly skewed frequency distribution, characteristic of Heavy Hitter problems. The distribution is dominated by content released in the last decade. As shown in Table I, the year 2018 appears most frequently (1,147 occurrences), accounting for 13.02% of the entire dataset. Remarkably, the top 10 most frequent years combined account for approximately 80.6% of all entries. This extreme concentration indicates that the vast majority of unique release years form a long tail of rare items.

Rank	release_year	Count	Perc.
1	2018	1147	13.02%
2	2017	1032	11.72%
3	2019	1030	11.70%
4	2020	953	10.82%
5	2016	902	10.24%
6	2021	592	6.72%
7	2015	560	6.36%
8	2014	352	4.00%
9	2013	288	3.27%
10	2012	237	2.69%

TABLE I: Top-10 Most Frequent release_year (Exact)

Figure 1 illustrates the cumulative distribution of item frequencies. The steep ascent of the curve confirms the sparsity of the data: the top 5 items alone cover over 50% of the stream.

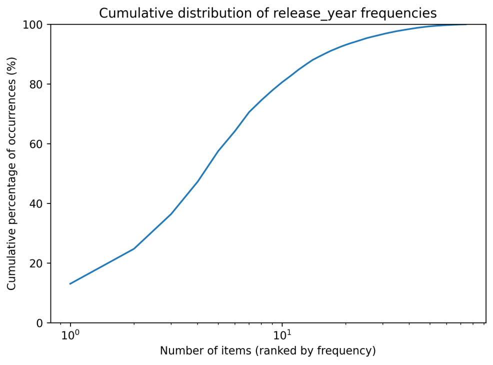


Fig. 1: Cumulative distribution of release_year frequencies. Note the logarithmic scale on the X-axis, highlighting that a handful of years constitute the majority of the dataset.

B. *Fixed Probability Sampling*

The Fixed Probability Counter (with sampling rate $p = 0.5$) was evaluated over 20 independent trials. This method's performance is stochastic, meaning accuracy varies based on which specific items are sampled during execution.

B.1 Estimation Accuracy

Table II presents the top-10 frequent items estimated by a representative run of the Fixed Probability algorithm. While the estimated counts deviate from the exact values (e.g., estimating 1,168 for 2018 vs. the exact 1,147), the algorithm successfully identifies the correct set of heavy hitters. The ranking order is largely preserved, with the top 10 years remaining in the top 10 positions.

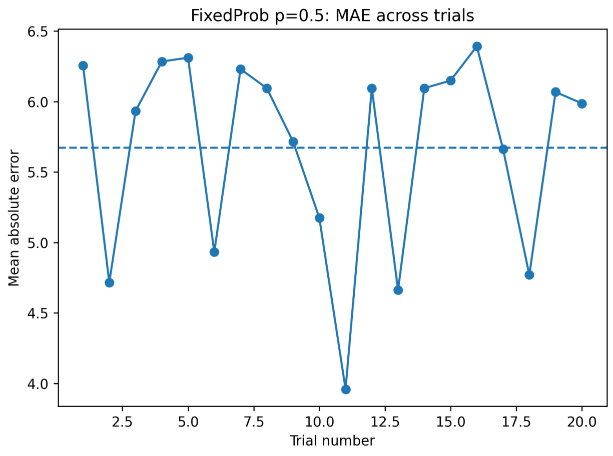
Rank	release_year	Est. Count	Perc.
1	2018	1168.00	13.26%
2	2017	1072.00	12.17%
3	2019	1072.00	12.17%
4	2020	966.00	10.97%
5	2016	912.00	10.36%
6	2021	604.00	6.86%
7	2015	546.00	6.20%
8	2014	330.00	3.75%
9	2013	318.00	3.61%
10	2012	216.00	2.45%

TABLE II: Top-10 Estimated Frequencies (Fixed Prob $p = 0.5$). Comparison with Table I reveals that while absolute counts fluctuate, the relative importance of items is captured accurately.

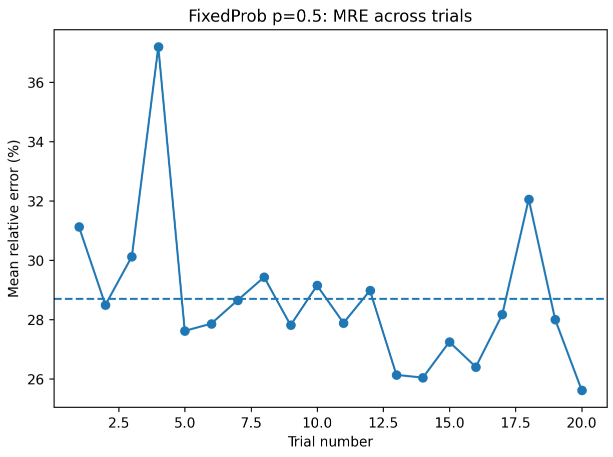
B.2 Error Stability and Rank Preservation

Figures 2a and 2b illustrate the fluctuation of error metrics across the 20 trials. The Mean Absolute Error (MAE) oscillates around an average of 5.7, which is a minor deviation given the magnitude of the top counts (> 1000). The Mean Relative Error (MRE) averages approximately 28.7%, indicating significant volatility for lower-frequency items.

Despite this variance, the Spearman Rank Correlation (Figure 3) remains consistently high, averaging **0.975**. This confirms that the Fixed Probability counter is highly effective for ranking tasks, even if the absolute counts are approximations.



(a) MAE across trials



(b) MRE across trials

Fig. 2: Error stability for Fixed Probability ($p = 0.5$). While absolute error is low, relative error is significant due to variance in low-frequency items.

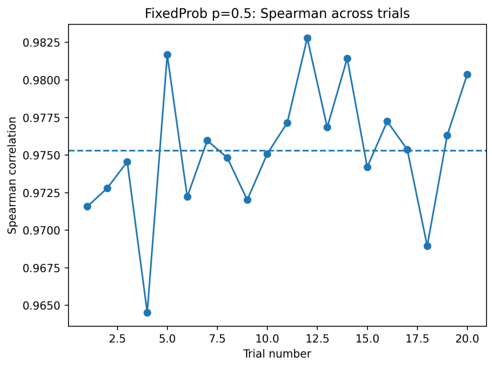


Fig. 3: FixedProb $p = 0.5$: Spearman rank correlation across 20 trials. The high average (0.975) indicates excellent preservation of the relative order of frequencies.

C. *Frequent-Count (Misra-Gries)*

The Frequent-Count algorithm was evaluated using a reduced buffer multiplier ($k = 2 \times n$) to analyze the trade-off between memory and accuracy.

C.1 Saturation and Perfect Accuracy

As shown in Table III, the Frequent-Count algorithm configured with $k = 100$ achieved **perfect accuracy**, producing counts identical to the Exact Counter baseline (Table I). This result is explained by the saturation phenomenon observed in Figure 4.

At lower settings ($n = 5, k = 10$), the algorithm exhibited high error due to frequent evictions. However, as n increased, the error dropped steeply. A critical inflection point was reached at $n = 40$ ($k = 80$). Since the dataset contains only ≈ 74 unique release years, a buffer size of $k = 80$ is sufficient to store every unique item without ever triggering the decrement-all eviction mechanism. Consequently, the algorithm transitioned from an approximate estimator to an exact counter.

Rank	release_year	Count	Perc.
1	2018	1147	13.02%
2	2017	1032	11.72%
3	2019	1030	11.70%
4	2020	953	10.82%
5	2016	902	10.24%
6	2021	592	6.72%
7	2015	560	6.36%
8	2014	352	4.00%
9	2013	288	3.27%
10	2012	237	2.69%

TABLE III: Top-10 Frequencies (Frequent-Count $k = 100$). Due to the buffer size exceeding the number of unique items, the algorithm acted as an exact counter.

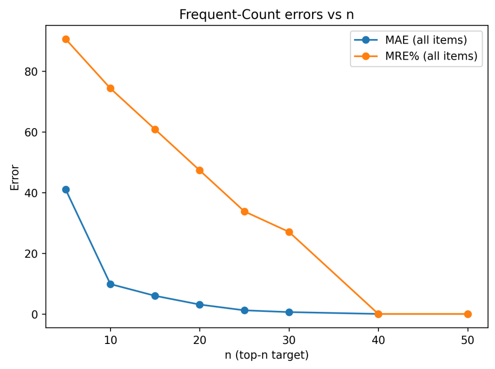


Fig. 4: Frequent-Count errors vs n. The error drops to zero once the buffer size k exceeds the dataset cardinality (≈ 74).

C.2 Buffer Utilization

Figure 5 confirms this behavior. The number of actual tracked items (blue line) rises linearly with k until it hits the ceiling of distinct items in the stream. Beyond $k = 80$, the buffer is underutilized, meaning memory is allocated but never used for evictions.

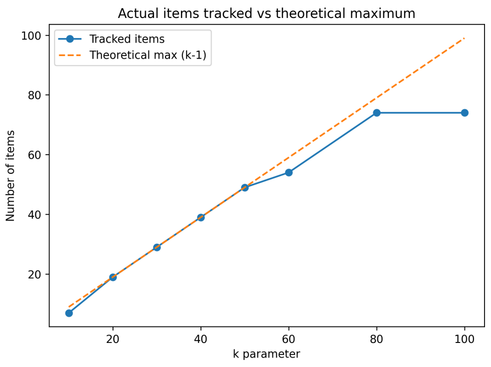


Fig. 5: Actual items tracked vs theoretical maximum $(k-1)$. The plateau clearly marks the point where the buffer size exceeds the dataset's unique item count.

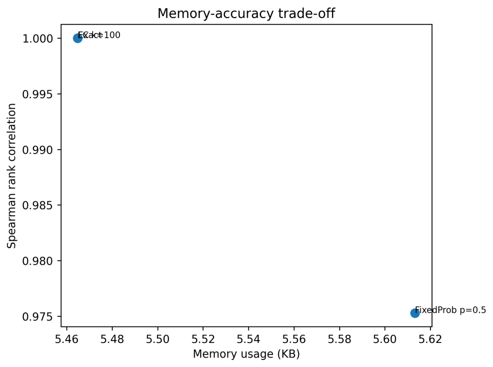


Fig. 7: Memory-accuracy trade-off. Due to the small dataset cardinality, the approximate methods overlap with the Exact counter in memory usage.

D. Comparative Analysis

D.1 Computational Efficiency

Figure 6 compares the execution times. The Fixed Probability counter was the fastest algorithm (≈ 0.86 ms), outperforming the Exact Counter (≈ 1.14 ms) by roughly 25%. This efficiency stems from processing fewer dictionary insertions. The Frequent-Count algorithm (≈ 0.94 ms) performed comparably to the exact method, as the overhead of eviction logic was rarely triggered in the saturated state.

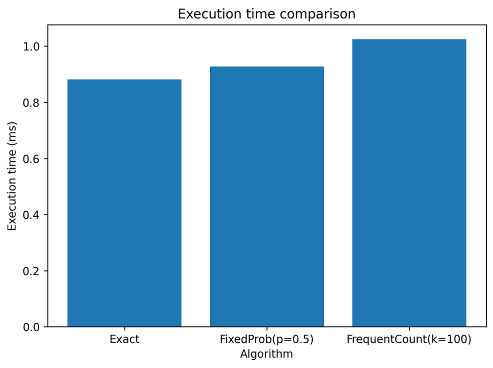


Fig. 6: Execution time comparison. Fixed Probability sampling offered the lowest latency.

D.2 Memory-Accuracy Trade-off

Figure 7 plots the algorithms in the memory-accuracy space. A counter-intuitive result is that the approximate methods did not save memory compared to the Exact Counter for this specific dataset. All three algorithms consumed ≈ 5.5 KB. The memory overhead of the Python dictionary structure dominated the storage cost of the actual integers, rendering the reduction in tracked items negligible for this low-cardinality dataset.

VI. Discussion

The experimental results highlight a strong dependency between algorithmic performance and dataset cardinality. A central finding was the saturation behavior of the Frequent-Count algorithm. While theoretically offering significant memory savings, this benefit is contingent on the dataset cardinality N being significantly larger than the buffer size k. In our experiments with the `release_year` attribute ($N \approx 74$), even small buffers captured all unique items, causing the algorithm to behave identically to an Exact Counter. This confirms that streaming algorithms are most advantageous for high-cardinality attributes rather than limited-domain metadata.

Additionally, the Fixed Probability Counter ($p = 0.5$) demonstrated a clear speed-accuracy trade-off. It achieved a 25% reduction in execution time compared to Exact Counting but at the cost of high relative error ($\approx 29\%$). However, its high Spearman correlation ($\rho \approx 0.98$) indicates robustness for ranking tasks where relative popularity is more critical than precise counts. Finally, implementation details matter: Python's object overhead meant that theoretical memory savings were not realized in practice.

VII. Conclusions

This study evaluated three strategies for identifying the items with the most frequent values in the `release_year` attibrute in the Netflix Movies and TV Shows dataset. The **Exact Counter** served as the optimal baseline for this low-cardinality dataset, providing perfect accuracy with minimal resource cost. The **Fixed Probability Counter** offered the best computational efficiency and excellent rank preservation, making it suitable for trend detection despite higher variance. The **Frequent-Count** algorithm demonstrated effective scaling, with error dropping to zero once the buffer size exceeded the number of unique items. Ultimately, for datasets with small domains, exact counting is preferable, while streaming algorithms are best reserved for massive, high-cardinality streams where memory constraints are a genuine bottleneck.

References

[1] Shivam Bansal, "Netflix Movies and TV Shows", `https://www.kaggle.com/datasets/shivamb/netflix-shows/code/data`, 2021, Accessed: 2025-12-26.

[2] Python Software Foundation, "The Python Standard Library: collections — Container datatypes", `https://docs.python.org/3/library/collections.html`, 2025, Python 3.10 Documentation.

[3] Daniel G Horvitz and Donovan J Thompson, "A generalization of sampling without replacement from a finite universe", *Journal of the American statistical Association*, vol. 47, no. 260, pp. 663–685, 1952.

[4] Graham Cormode and Marios Hadjieleftheriou, "Finding frequent items in data streams", *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[5] Jayadev Misra and David Gries, "Finding repeated elements", *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

Appendix

Algorithm	Time (ms)	Mem (KB)	MAE	MRE (%)	Spearman
Exact	1.14	5.46	0.00	0.00%	1.0000
FixedProb ($p = 0.5$)	0.86	5.61	5.67	28.70%	0.9753
FrequentCount ($k = 100$)	0.94	5.46	0.00	0.00%	1.0000

TABLE IV: Algorithm comparison summary (Netflix release_year). Note that for FC, the saturated configuration ($k = 100$) was selected for comparison.