Streaming Algorithms Evaluation: Heavy Hitters and Quantiles Estimation in Streaming Data

CASCS 543 Final Project

Yan Zhen Edwin Leck
4 May 2025

Project Overview

This project aimed to design and evaluate a streaming data system that can efficiently analyze continuous data streams to identify frequently occurring elements (heavy hitters) and estimate key quantiles such as the median and the 90th percentile. A fundamental challenge addressed in this project was the need to process virtually infinite data using only finite memory resources, which is essential for scalability and real-time analytics in high-throughput environments.

The motivation for this work stems from the growing demand for real-time analytics in domains like network security, financial markets, and recommendation systems. Traditional batch processing methods fail to handle unbounded data streams, necessitating sublinear-memory algorithms that provide approximate answers with provable error bounds. The project's dual focus on heavy hitters and quantiles reflects their ubiquity in streaming applications—whether detecting trending assets in finance or monitoring latency outliers in network traffic.

Core Algorithms

To solve the above challenges, I implemented and evaluated two main sketching algorithms: Count-Min Sketch (CMS) for frequency estimation and KLL Sketch for quantile approximation. Count-Min Sketch is a probabilistic data structure that estimates the frequency of elements in a stream using multiple hash functions to map items into sublinear-sized counters. While it guarantees an upper bound on the frequency (i.e., always overestimates due to hash collisions), a more refined version known as Conservative Update CMS was also implemented. This version only

increments the counters that are currently at the minimum value for a given item, helping to reduce overestimation errors—a critical improvement highlighted in the presentation.

For quantile estimation, the KLL Sketch emerged as the preferred algorithm due to its hierarchical compactor design, which dynamically adjusts error bounds based on available memory. Unlike the older Greenwald-Khanna (GK) Sketch, which maintains a sorted summary with explicit gap tracking, KLL uses randomized compaction and weighted merging to achieve better space-time tradeoffs. The presentation emphasized KLL's advantages, including its $O(1/\varepsilon \log^2(1/\varepsilon))$ memory complexity and constant-time updates, making it ideal for high-throughput scenarios.

Baseline methods were also included for reference. Naive Counting offers exact frequency tracking but is memory-intensive and impractical for large streams. For quantiles, Reservoir Sampling was tested as a simpler alternative, though it proved less accurate for tail quantiles compared to KLL.

Implementation Components

The codebase was organized into modular components under a unified structure. The `/algorithms/` directory contains the implementation of CMS, Conservative CMS, and KLL Sketch. Datasets, including real-world and synthetic ones, are stored in `/data/`, while all evaluation scripts are located under `/experiments/`. Utility functions such as hash generators and metric calculations reside in `/utils/`, with global parameters defined in `config.yaml`, and `main.py` serving as the project's entry point.

The key files include `count_min_sketch.py`, which performs basic frequency estimation using multiple hash functions and sublinear memory, and `conservative_cms.py`, which implements a variant of CMS that selectively updates counters to mitigate overcounting. The file `kll_sketch.py` is responsible for quantile estimation using a hierarchy of compactors. The `memory_benchmark.py` script compares how each sketch grows in memory as the stream size increases.

The incoming data is split between CMS for heavy hitters and KLL for quantiles, enabling parallel processing. This design aligns with the project's goal of unifying frequency and quantile analysis in a single framework.

How Everything Fits Together

The data flow starts with preprocessing real-world financial data, particularly Bitcoin trade data. The raw CSV file is cleaned, with timestamps converted into datetime format and relevant columns (price and amount) extracted. Outliers are filtered out based on price and trade size thresholds. The cleaned dataset (`network_logs.csv`) is then fed into the sketching algorithms.

For heavy hitters, rounded price values (to two decimal places) are treated as tokens, and CMS variants are evaluated against ground truth frequencies. The presentation noted that while CMS achieved perfect recall (1.00), its precision was low (0.18) due to overestimation—a limitation partially addressed by Conservative CMS. Quantile estimation tests used trade amounts, with KLL outperforming Reservoir Sampling at high percentiles (e.g., 90th and 99th) but lagging at the median (46.6% error vs. 2.18% for Reservoir).

Memory benchmarks revealed sublinear growth, with KLL using only 412 bytes for 1M items—though this suspiciously low value suggested potential state retention issues. Computational efficiency was strong across the board, with CMS processing 3M items/sec, KLL at 1.14M, and Reservoir at 1.01M.

Why This Matters

This project has practical implications for real-time analytics. In network security, it could detect frequent attackers or latency spikes; in finance, it could monitor asset trends or volatility. The presentation highlighted how conservative updates in CMS and KLL's compaction strategy advance the state of the art by improving accuracy and memory efficiency.

From a technical perspective, the project demonstrated that:

1. Conservative CMS reduces overestimation errors compared to vanilla CMS.
2. KLL Sketch offers superior memory-accuracy tradeoffs over GK and Reservoir Sampling.
3. A unified framework for frequency and quantile analysis is feasible and scalable.

Debugging the Errors

Throughout the implementation, several bugs were identified and resolved. An `AttributeError` due to missing ground truth counters was fixed by implementing proper counting logic using Python's `defaultdict`. A `ValueError` emerged from YAML parsing scientific notation (e.g., `1e4`) as strings; I resolved this by explicitly defining these numbers (e.g., `10000`). Additionally, the initial memory measurements relied on shallow `sys.getsizeof()` calls, which underestimated actual usage. I replaced this with recursive memory tracking for better accuracy.

Dataset Overview: Bitcoin Trade Data

The experiments in this project leveraged Bitcoin trade data sourced from Kaggle, comprising 100,000 trade records with millisecond timestamps, trade prices, and transaction amounts. This dataset was selected for its high cardinality (31,625 unique prices) and heavy-tailed distribution of trade sizes, with a median of 0.011 BTC and extreme values reaching 20 BTC. Such characteristics made it an ideal benchmark for stress-testing both heavy hitter detection and quantile estimation under real-world conditions. The skewed distribution of trade amounts—where most transactions were small but a few were exceptionally large—posed a challenge for quantile algorithms, particularly in accurately capturing tail percentiles like the 90th and 99th.

Heavy Hitter Detection: Implementation and Challenges

The Count-Min Sketch (CMS) implementation began with a matrix of counters initialized by width (number of hash buckets) and depth (number of hash functions), using MurmurHash (mmh3) with random 32-bit seeds for hashing. Each item's frequency was estimated by taking the minimum value across all hashed counters, a design choice that inherently led to overestimation due to hash collisions. To mitigate this, the Conservative CMS variant was implemented, which only incremented counters if they were at the current minimum estimated frequency for the item.

However, evaluation results revealed identical performance between the basic and conservative CMS variants: both achieved perfect recall (1.00) but suffered from abysmal precision (0.18). Some price points were overestimated by 130 times their true frequency, indicating severe hash collisions or flawed conservative update logic. The root cause analysis suggested two primary issues:

1. Poorly seeded hash functions, which exacerbated collisions.
2. Ineffective conservative updates, where the logic failed to constrain counter growth as intended.

To address these, the project proposed switching to cryptographic hash seeds (e.g., SHA-256) and adding audit logs to verify counter updates. Synthetic streams with known heavy hitters (e.g., `["A"]*1000 + ["B"]*500 + noise`) were also recommended for ground-truth validation.

Quantile Estimation: KLL Sketch and Its Limitations

The KLL Sketch was implemented as a hierarchical compactor structure, where incoming data was buffered and periodically compressed by randomly discarding half the elements in each layer. Higher layers retained fewer but statistically weightier items, enabling memory-efficient quantile approximation. Despite its theoretical advantages over the Greenwald-Khanna (GK) Sketch, the initial implementation struggled with median estimation (P50), exhibiting a 46.6% error—far exceeding the expected <5% theoretical bound. In contrast, Reservoir Sampling outperformed KLL at the median (2.18% error) but faltered at higher percentiles (14.16% error at P90).

The KLL's poor median accuracy was attributed to over-aggressive compaction and improper merging of compactor layers, which disproportionately discarded critical data points from lower levels. Proposed fixes included:

1. Dynamic compactor size (k): Increasing `k` when error thresholds were exceeded to retain more data in early layers.
2. Weighted merging: Prioritizing retention of high-precision data from deeper layers during compaction.
3. Debugging logs: Tracking layer-wise item counts to validate compaction behavior.

Computational Efficiency and Tradeoffs

Benchmarks revealed stark tradeoffs between algorithms:

- CMS processed 3 million items/sec with fixed memory (20.2 KB), but it's overestimation rendered it unreliable for precise heavy hitter detection.
- KLL achieved 1.14 million items/sec with minimal memory (412 bytes for 1M items), though its suspiciously low memory usage hinted at implementation flaws.

- Reservoir Sampling balanced speed (1.01 million items/sec) and accuracy for median estimates but lacked KLL's theoretical guarantees for tail quantiles.

Key Innovations and Practical Implications

The project's most striking finding was the failure of Conservative CMS to improve upon basic CMS, underscoring the importance of rigorous hash function design and update logic validation. Meanwhile, Reservoir Sampling's unexpected robustness suggested its viability as a lightweight alternative for real-time monitoring, despite its lack of formal error bounds.

For crypto exchanges, these results highlighted the inadequacy of vanilla CMS for detecting true "hot" prices due to false positives, while algorithmic trading applications demanded fixes to KLL's median estimation for reliable volatility analysis. A hybrid approach—combining KLL for tail quantiles and Reservoir Sampling for speed—was proposed as a pragmatic solution.

Debugging the Errors: A Summary

The primary errors identified in the project fell into three categories:

1. Hash Collisions in CMS: Poor seeding led to artificial frequency overestimation, necessitating cryptographic hashes.
2. Conservative Update Ineffectiveness: The logic failed to enforce minimal counter increments, requiring audit trails and synthetic validation.
3. KLL's Over-Compaction: Aggressive discarding of lower-layer data skewed median estimates, mandating dynamic compaction thresholds.

Conclusion and Future Directions

The project underscored that theoretical guarantees alone are insufficient without meticulous implementation. Future work would focus on:

- Algorithmic refinements: Adopting SHA-256 hashing for CMS and dynamic `k` for KLL.
- Real-world validation: Testing on live financial data or social media streams.
- Hybrid systems: Combining the strengths of Reservoir Sampling and KLL for balanced speed-accuracy tradeoffs.

By addressing these gaps, the project aims to advance the state of streaming analytics, offering robust tools for high-throughput environments like finance and cybersecurity.

Extending the Project

There are several directions for future work. First, real-world datasets such as the Twitter firehose or live exchange feeds could be incorporated to further test robustness. A distributed version of the system could also be developed to support horizontal scaling. Additionally, anomaly detection modules could be added to leverage quantile estimates for spotting rare or suspicious behavior.

Advanced variants such as Cuckoo Sketch could be explored for better collision handling, while DDSketch may offer improved quantile estimation, particularly for skewed data distributions.

Targeted Improvements for Algorithm Differentiation

If more development time were available, several enhancements would be pursued. For CMS, I would replace the hash functions with cryptographically seeded variants such as SHA-256 to reduce artificial collisions. I would also introduce audit logs during counter updates to verify that the conservative logic is functioning correctly. To test improvements, I would construct synthetic streams with known heavy hitter frequencies and measure the reduction in overestimation error.

For KLL, I would revise the compaction trigger to use a dynamic k that adapts based on observed error thresholds. Merging policies would be made weight-aware to preserve more significant data in upper compactor layers. Logging compactor statistics would also help in debugging uneven distribution of elements.

These targeted improvements could significantly enhance differentiation between algorithm variants and improve the reliability of results in both controlled and real-world settings.