Measuring the Cardinality of Big Streaming Datasets: A Comparative Study

[Final Report]

Emaad Ahmed Manzoor Fatemah AlZayer Jumana Baghabra

Meshari Alazmi Tariq Alturkestani

1. INTRODUCTION

The cardinality of data is the number of unique elements in the set underlying the data. Cardinality measurements arise naturally in a number of applications such as optimising database queries [8], analysing sensor network data [4] and calculating internet traffic metrics [11]. Further, useful data streams are becoming increasingly accessible from sources like Twitter [3] and inexpensive sensors.

A common characteristic of these data streams is their extremely high volumes and velocities. Hence, the efficient analysis of such streams typically poses the following challenges:

1. Only a single pass over the data is allowed.
2. Though the stream is infinite, analysing it must proceed with constant memory (or sublinear in the number of data elements, at the least).

In the context of cardinality estimation, a simple approach would maintain a hash table of observed elements, and a counter tracking the number of elements in the hash table would provide the required cardinality measurement.

While this approach proceeds with a single pass in constant time (assuming a constant number of hash collisions), it uses space that is linear in the real cardinality of the data. This is unacceptable for infinite streams with potentially terascale cardinality.

We propose here a study of two approximate algorithms that mitigate this unbounded memory usage, while also satisfying the single-pass condition of efficient streaming data analysis.

2. MOTIVATING APPLICATION

As an example of an application that would benefit from constant space cardinality estimation, we consider [5]. This discusses building a classifier to detect "buzzing" queries from Twitter. Twitter is an infinite stream of text with a large and varied vocabulary.

Buzzing queries are phrases which exhibit a spike in their frequency. The buzziness of a query is measured by finding the ratio of maximum-likelihood estimates (MLE) of a query in two models separated in time.

A buzzing query is different from a popular query, for which a simple frequency count would be sufficient. The spike in a frequency of a popular query will be small due to having the same high frequency in different time models.

The buzz score of a query is defined as:

$$\begin{aligned} BuzzScore(q) & \\ &= BuzzScore(word_1, word_2, ...) \\ &= \frac{MLE(word_1|d)}{MLE(word_1|d-1)} \times \frac{MLE(word_2|d)}{MLE(word2|d-1)} \cdots \end{aligned}$$

The maximum-likelihood estimate of a word given a language model for time d is:

$$MLE(word|d) = \frac{n_d(word) + 1}{\sum_{w \in words} n_d(w) + |V_d|}$$

where $n_d(word)$ is the number of occurrences of the word in the model d, and $|V_d|$ is the size of the vocabulary for the model d (the number of distinct words). $|V_d|$ is a component of the Laplacian smoothing term, to prevent assigning a zero score to words in queries that are not present in the model.

Given the nature of the Twitter stream; it's high volume and velocity, we aim at efficiently calculating the vocabulary size of the Twitter stream by using an approximate cardinality estimation algorithm.

3. ALGORITHMS

The algorithms under study here apply different probabilistic techniques to estimate the cardinality with a bounded relative error. They assume the existence of a function that can uniformly hash values to binary strings, such that the bits of the hashed values are independent and each have a probability $\frac{1}{2}$ of occurring.

3.1 Adaptive Sampling

Introduced by Wegman [12] in a private communication that is unavailable to the general public. We refer to Flajolet's analysis [6] of this algorithm as our primary source.

This conceptually simple algorithm estimates cardinality using m memory words (a word is typically 32 or 64 bits), with an expected relative error:

$$\frac{1.20}{\sqrt{m}}$$

The algorithm requires setting m, the maximum number of memory words to use, and a sequence of predicates P_i satisfying the following conditions, assuming x is any data element:

1. $P_{i+1} \subset P_i$
2. $Probability(x\ satisfies\ P_i) = 2^{-i}$

A hashtable of at most m elements is maintained, derived by hashing each data element to a binary string. At any instant, one of these predicates is active, and all the members of this hashtable must satisfy the active predicate.

For our study, we set these predicates to be:

$$P_i = 0^i(0|1)^*$$

That is, for an element to satisfy P_i, its hashed value must begin with at least i zeroes.

Execution begins with P_0 as the active predicate and an empty hash table. Each new element is tested against the active predicate, and hashed if it is satisfied. Every time a new insertion causes the number of hashed elements to exceed m, the next predicate in the sequence is activated and the hash table is purged by deleting all elements that do not satisfy the new predicate.

The estimated cardinality E is given by,

$$E = 2^i \times l$$

Where i is the index of the active predicate, and l is the number of hashed elements at that instant.

We illustrate a run of adaptive sampling in Table 1, as presented in [6]. The predicates are defined as above, and the hashtable can have at most 3 elements (i.e. $m = 3$).

The algorithm starts with the word UDINE that will be hashed in the table at depth zero. The second and third words are NICE and PARIS, and since they have not been seen before, they will be added to give us three values in the hash table. But when BORDE arrives, the hash table will be at maximum capacity.

The shaded row in Table 2 shows the situation when the hash table is full. When this happens, the currently active predicate is incremented, and the hash table is pruned of all elements that do not satisfy the new predicate. This results in the row right after the shaded row.

The other elements continue arriving and updating the hash table in a similar manner. On the arrival of every new element, the estimated count can be maintained as per the estimation formula.

3.2 HyperLogLog

Introduced in [7]; we also refer to [9] for practical engineering guidelines that correct some implementation issues in the original algorithm.

In the algorithm, each element of the data stream is hashed to a bit string, and the maximum number of leading zeroes across such bit strings is observed. This number is an indication of the cardinality of the dataset.

Each bit in a uniformly generated bit string has a probability $1/2$ of being either 0 or 1. Therefore, as an example, the probability of starting with 2 0's (or 1's) is $1/2 \times 1/2 = 1/4$. We observe this in play across our bit strings:

- 50% of the bit strings will be of the form: $0(0|1)*$
- 25% of the bit strings will be of the form: $00(0|1)*$
- 12.5% of the bit strings will be of the form: $000(0|1)*$

Based on these observations, we can also say that:

- If we see a bit string of the form $0(0|1)*$, we can expect the cardinality to be at least $100/50 = 2$.
- If we see a bit string of the form $00(0|1)*$, we can expect the cardinality to be at least $100/25 = 4$.
- If we see a bit string of the form $000(0|1)*$, we can expect the cardinality to be at least $100/12.5 = 8$.

So if the maximum number of leading zeros we observe is k, we expect the cardinality to be at least 2^k.

However, we cannot use just this as our cardinality estimator, as it exhibits a very high variance.

INPUT	HASHED	HASH TABLE	P_{active}	ESTIMATE	EXACT
UDINE	10101	10101	P_0	1	1
NICE*	00101	10101 00101	P_0	2	2
PARIS	11011	10101 00101 11011	P_0	3	3
BORDE	01001	00101 01001	P_1	4	4
NAFPL	11101	00101 01001	P_1	4	5
PARIS	11011	00101 01001	P_1	4	5
BORDE	01001	00101 01001	P_1	4	5
MARSE	01010	00101 01001 01010	P_1	6	6
RENNE	10100	00101 01001 01010	P_1	6	7
LEIPZ	00010	00101 00010	P_2	8	8
CAEN*	10001	00101 00010	P_2	8	8
QUEBE	00111	00101 00010 00111	P_2	12	10
PISA*	00100	00010	P_3	8	11

Table 1: Adaptive Sampling: Illustration

INPUT	HASHED	HASH TABLE	P_{active}	ESTIMATE	EXACT
UDINE	10101	10101	P_0	1	1
NICE*	00101	10101 00101	P_0	2	2
PARIS	11011	10101 00101 11011	P_0	3	3
BORDE	01001	10101 00101 11011 01001	P_1	4	4
BORDE	01001	00101 01001	P_1	4	4

Table 2: Adaptive Sampling: Hashtable Overflow

To overcome this, we split our stream into m substreams, and observe the maximum number of leading zeros in each substream. We can then take the average of the estimator across each of the substreams, and then use it to estimate our cardinality:

$$Cardinality = constant \times m \times 2^{r_{avg}}$$

Where r_{avg} is the maximum number of leading zeros averaged over m substreams.

This is actually the estimator for LogLog counting. With this estimator, the distribution of maximum leading zeros for m registers was observed to be skewed to the right, and there could be some outliers that render this average a bad estimate.

In HyperLogLog, the average is replaced by the harmonic mean, providing us the following estimator:

$$E = \frac{\alpha_m m^2}{\sum_{i=1}^{m} 2^{-C_i}} \quad \text{where } \alpha_m = \frac{1}{m \int_0^{\infty} \left(\log_2(\frac{2+u}{1+u})\right)^m \mathrm{d}u}$$

This provides an estimate of the cardinality using m *counters* (typically short bytes), with an expected relative error:

$$\frac{1.04}{\sqrt{m}}$$

4. THEORETICAL RESULTS

To ease the theoretical analysis, the following assumptions are made for both the algorithms:

1. A uniform hash function is used. This ensures that every hashed value is equally likely. This is attainable in practice if the hash function we use is a reasonable one; for example, multiplicative hashing or the Jenkins hash function [10], and if the data is random enough; that is, the data must not be inherently skewed towards particular values.

2. No collisions occur. This is attainable in practice if the cardinality we are estimating n, using an l bit hash value, is such that $n \ll 2^l$.

4.1 Adaptive Sampling

For adaptive sampling, we will sketch derivations of the cardinality estimate, the time and space complexity, and the relative accuracy of the estimation.

4.1.1 Cardinality Estimate

For every hashed value currently in our table, represent its substring starting after the i^{th} zero as h_j, where $0 \leq j \leq l$. For example, for a hash value 010100 and active predicate P_2, the substring is 0100.

When the i^{th} predicate is active, our hash table will only have hashed values that begin with at least i zeros. So we are

assuming having already counted all unique hashed values of the forms $(0|1)(0|1)h_j$.

The number of such unique strings for a given h_j is 2^i. Since there are currently l values in our hash table, we are assuming having already seen $2^i \times l$ such strings, and this is our cardinality estimate.

4.1.2 Time Complexity

Adaptive sampling could perform any of the following operations:

1. For every element in the stream, check its existence in the hash table.
2. If an element is not present, insert it into the hash table.
3. Increment the active predicate.
4. Update the cardinality estimate.
5. Prune the hash table.

Operations 1, 2, 3 and 4 take place in constant time per element, assuming no collisions. Operation 5 takes place in $O(m)$ time, where m is the maximum size of our hash table. This is a fixed constant. So the overall time complexity is $O(n)$, when n input elements have been observed.

4.1.3 Space Complexity

The maximum amount of space occupied is $O(m)$, since the hash table grows to at most $m+1$ elements and is pruned after that. Since m is a constant that is fixed for the algorithm, space complexity is $O(1)$.

4.1.4 Relative Accuracy

We assume that hashed values are infinitely long bit strings that are independently and uniformly distributed over $0, 1^\infty$. This is another way of stating our uniform hashing and no collisions assumptions.

Given these assumptions, the probability of drawing n bit strings such that k of these start with a 0-bit is:

$$B_{n,k} = \frac{1}{2^n}\binom{n}{k}$$

Let ω be a finite subset of $\{0,1\}^\infty$. Define two subsets of ω:

1. $\omega/0 = \{y \in \{0,1\}^\infty \mid 0y \in \omega\}$
2. $\omega/1 = \{y \in \{0,1\}^\infty \mid 1y \in \omega\}$

This is similar to our definitions of h_j in the derivation of the cardinality estimate. As an example, if $\omega = \{010, 100, 101\}$, then $\omega/0 = \{010\}$ and $\omega/1 = \{100, 101\}$.

Let $K(\omega)$ be the estimate provided by adaptive sampling. Then $K(\omega)$ can be defined recursively as:

$$K(\omega) = \begin{cases} 2K(\omega/0) & \text{if } |\omega| > m \\ |\omega| & \text{otherwise.} \end{cases}$$

If a random n-subset ω is chosen from $\{0,1\}^\infty$, then $K(\omega)$ is a random variable. Let $E[X]$ represent the expected value of a random variable X, then we define the following:

1. $K_n = E[K(\omega)]$
2. $L_n = E[K^2(\omega)]$

Expanding these for $n > m$, using the recursive definition of $K(\omega)$, the definition of expectation, and the Bernoulli probability distribution formula:

$$\begin{aligned} K_n &= E[K(\omega)] \\ &= E[2K(\omega/0)] \\ &= 2E[K(\omega/0)] \\ &= 2\sum_{k>=0} \frac{1}{2^n}\binom{n}{k} K_k. \end{aligned}$$

$$\begin{aligned} L_n &= E[K^2(\omega)] \\ &= E[4K(\omega/0)] \\ &= 4E[K(\omega/0)] \\ &= 4\sum_{k>=0} \frac{1}{2^n}\binom{n}{k} L_k. \end{aligned}$$

The initial conditions would be $K_n = n, L_n = n^2$, when $n \leq m$.

The rest of the analysis proceeds in [6] by introducing the exponential generating functions for K_n and L_n, and finally deriving the variance of the estimate provided by adaptive sampling when applied to n random binary strings:

$$V_n = \frac{n^2}{(m-1)log2} + n^2 P(log_2 n) + o(n^2)$$

Where $P(u)$ is a periodic function of u with mean 0 and Fourier expansion $P(u) = \sum_{k \in Z0\} p_k e^{-2iku}}$ such that:

$$p_k = \frac{1}{log2}\Gamma(-1 + 2ik\pi/log2)\binom{2ik\pi/log2 + m - 2}{m-1}$$

If we neglect the periodic fluctuation of this variance, we get:

$$V_n \approx n^2/((m-1)log2)$$

We would like to know the standard error of the estimate, which is defined as the quotient of the standard deviation of the estimate by the exact value n. So we get:

$$\begin{aligned} \frac{\sqrt{V_n}}{n} &= \frac{\frac{n}{\sqrt{((m-1)log2)}}}{n} \\ &= \frac{1}{\sqrt{log2}\sqrt{m-1}} \\ &\approx \frac{1.20}{\sqrt{m}} \end{aligned}$$

4.2 HyperLogLog

4.2.1 Time Complexity

In the HyperLogLog algorithm, we only deal with the following operations

1. Count the number of leading zeros in a bit string.
2. Select one of m registers to update, based on the input bit string.
3. Update the count in a register.
4. Compute the cardinality estimate.

Operation 1, 3 and 5 are trivially constant time operations per element. Operation 2 can be implemented by a hardware operation over a binary word, and is hence effectively a constant time operation per element.

Hence, the overall time complexity is $O(n)$, if we have observed n elements.

4.2.2 Space Complexity

Space is required for only a fixed number of registers m, each of which store the maximum number of leading zeros seen in the m^{th} substream. So the space complexity is also constant: $O(1)$.

4.2.3 Relative Error

For the relative error, we state the theorem proved in [7]:

The standard error satisfies as $n \to \infty$:

$$\frac{\sqrt{V_n}}{n} = \frac{\beta_m}{\sqrt{m}} + \delta_2(n) + o(1)$$

Where $|\delta_2(n)| < 5.10^{-4}$ as soon as $m \geq 16$, the constants β_m being bounded, with $\beta_{16} = 1.106, \beta_{32} = 1.070, \beta_{64} = 1.054, \beta_{128} = 1.046, \beta_\infty = 1.03896$.

Here too, we can neglect the oscillating function of small magnitude $\delta_2(n)$, In the limiting case, with an infinite number of registers, we substitute the value of β_∞ to get the relative error $\approx 1.04/\sqrt{m}$.

5. IMPLEMENTATION DETAILS

5.1 Hash Function

We choose a hash function based on the following criteria:

1. Size of the hashed value
2. Distribution of hashed values
3. Computation time for a hashed value

Based on a study in [1], we use MurmurHash3[2] as our hash function, which is a fast non-cryptographic hash function with a low collision rate. We set the size of our hashed value to be 32 bits.

5.2 Measurements

We use system calls on Windows/Linux to get the exact CPU time and memory used by a specific process. This ensures that our running time measurements are not affected by context switches to other processes.

5.3 Datasets

We evaluate adaptive sampling on 2 datasets: *Wuthering Heights* and *Shakespeare*, which are freely available books. We evaluate HyperLogLog on 3 datasets: *Wuthering Heights*, *Shakespeare* and *Sherlock Holmes*. Since adaptive sampling fails to estimate datasets of large cardinality without using too much memory, we do not evaluate it on *Sherlock Holmes*.

While processing the input data, we remove all punctuation and convert all text to lower-case.

We measure input size at a time as the number of lines read from these datasets at that time.

5.4 Adaptive Sampling

We use the following predicates:

$$P_i = (0|1)^*0^i$$

That is, for an element to satisfy P_i, its hashed value must end with at least i zeros. This enables efficient implementation using bit operations.

6. EXPERIMENTAL RESULTS

6.1 Exact Cardinality Estimation

From Figure 1, we can observe that the running time is linear in the input size.

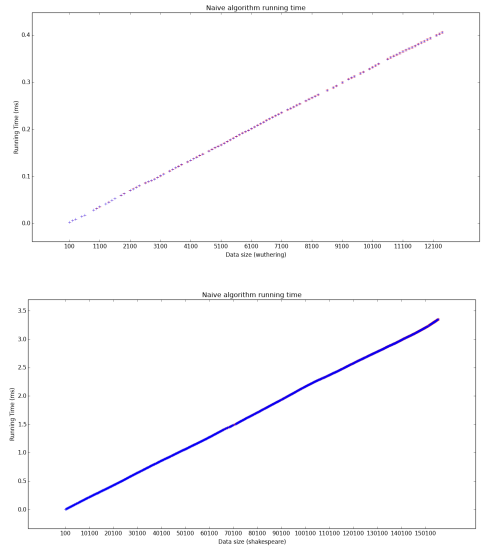

Figure 1: Exact Cardinality Running Time

From Figure 2, we observe a step function for the memory usage. This is an artifact of the implementation, which uses Python dynamic arrays.

Dynamic arrays are preallocated with a fixed number of elements. On exceeding capacity, the array is resized to twice the old size. This is why the graph displays a step function instead of a straight linear curve. However, the space complexity is in fact, linear in the cardinality of the input.

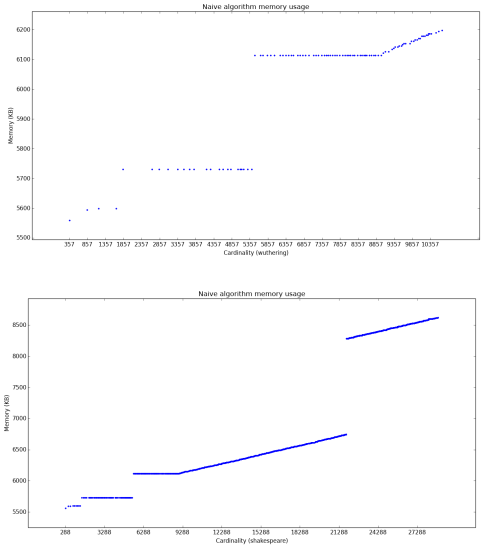

Figure 2: Exact Cardinality Memory Usage

The exact cardinality of the two datasets increases with the input size as shown in Figure 3.

6.2 Adaptive Sampling

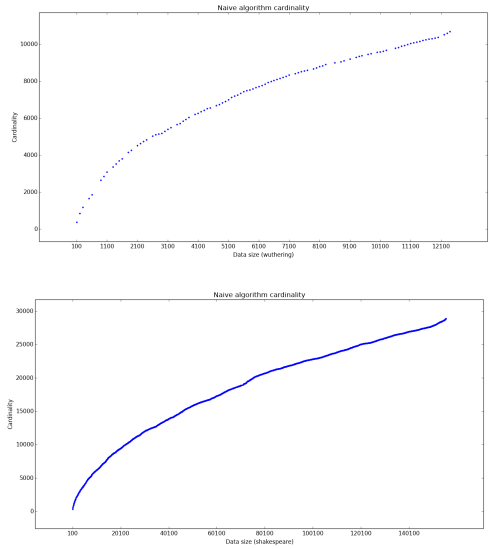

Figure 3: Exact Cardinality Value

We plot the metrics for adaptive sampling with different values of m, the size of the hash table. The value of m is encoded as a colour, with the value provided by the colourbar to the right of each plot.

From Figure 4, we see that the running time is linear in input size for all values of m. However, among different values of m, there is a variation in the running time of the algorithms. For the *Wuthering Heights* dataset, we observe that the running time decreases with increasing m. But for the *Shakespeare* dataset, we observe varied running times.

The reason is that the running time of adaptive sampling is affected by two operations:

1. The number of elements in the hash table to prune on predicate update.

2. The number of times the hash table is pruned; i.e. the number of predicate updates.

There is a trade-off between these two operations depending on the value of m and the actual cardinality. If the value of m is large, then we will require fewer pruning operations, but each pruning operation will take time. If the value of m is small, pruning will not take time but there will be many predicate updates if the cardinality is large.

From Figure 5, we observe the memory usage of adaptive sampling for a given value of m to increase to a certain maximum and then remain constant. The maximum memory used increases with increasing m. For a very large m greater than the actual cardinality, adaptive sampling degenerates to the exact algorithm, with no pruning operations.

Figure 6 shows the relative error of adaptive sampling for different values of m. The horizontal lines are contours of

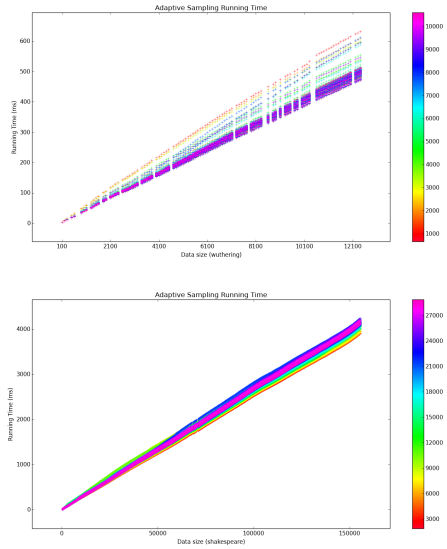

Figure 4: Adaptive Sampling Running Time

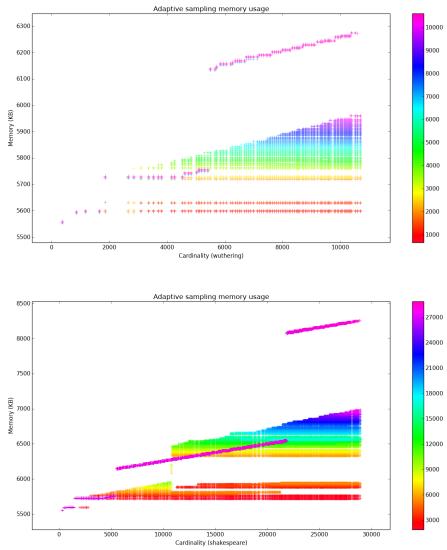

Figure 5: Adaptive Sampling Memory Usage

the expected relative error $\frac{1.20}{\sqrt{m}}$ for different values of m, as indicated by their colour in the colourbar.

Since the cardinality of *Wuthering Heights* is low, we observe that the relative error tends to be lower than the expected value. Figure 7 shows a more detailed view of the relative error for different ranges of m.

Shakespeare has a high cardinality, so we observe the relative error to vary with high probability around its expected value. This is displayed in detail in Figure 8 for different values of m.

6.3 HyperLogLog

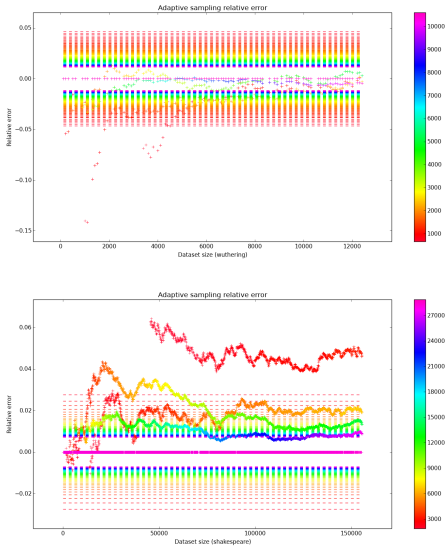

Figure 6: Adaptive Sampling Relative Error

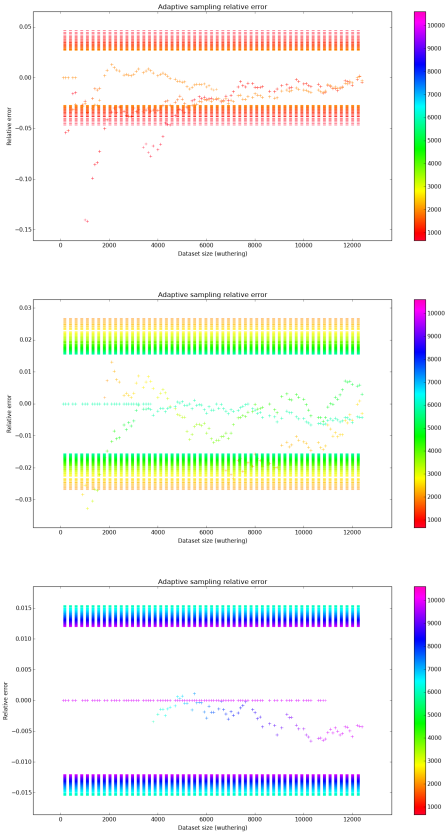

Figure 7: AS Error Details (Wuthering Heights)

In Figure 9, we see that the HyperLogLog uses constant memory for a given value of m, while the exact (naive) algorithm uses memory linear in the actual cardinality.

We also observe from Figure 10 that increasing the size of m

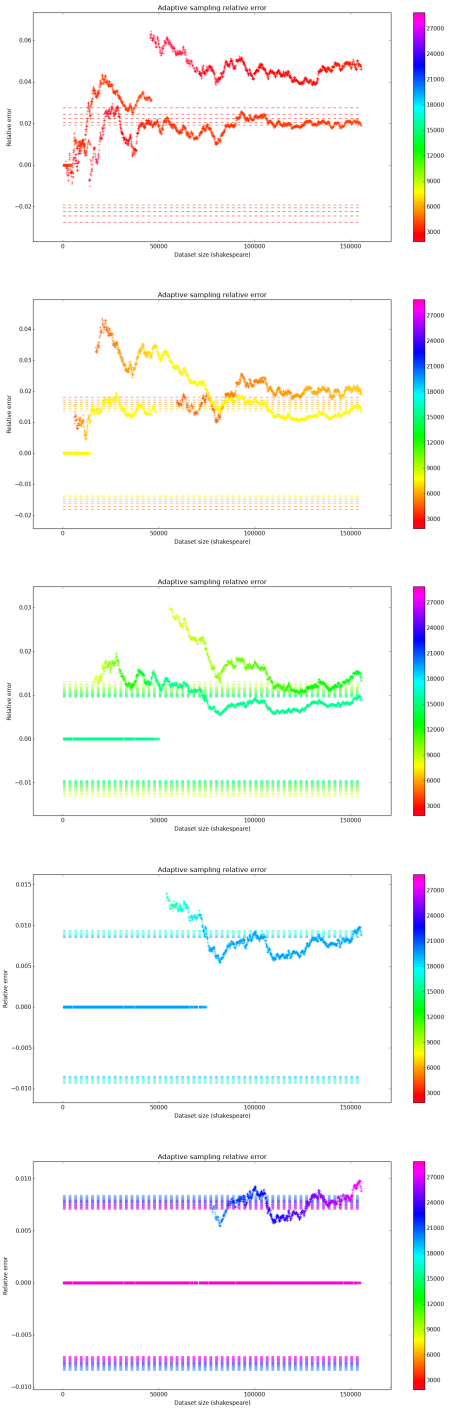

Figure 8: AS Error Details (Shakespeare)

increases the accuracy of the estimated cardinality. We do do not need to make m very large; an m of size 8192 would be sufficient and would give us an estimated cardinality with a small relative error.

7. CONCLUSION

Figure 9: HyperLogLog Memory Usage

We have empirically evaluated the running time, memory usage and relative error of two approximate cardinality estimation algorithms.

Both algorithms use lesser memory than the exact algorithm. For both algorithms, the memory used is constant, while for the exact algorithm it grows linearly with the exact cardinality.

We find that both algorithms follow the same time and space complexity. However, for a fixed value m of space to be used, HyperLogLog can estimate higher cardinalities accurately,

As *m* increases, HLL counts better
Actual Cardinality 60457

m	16	32	64	128	256	512	1024
Error	79.58%	30.69%	19.96%	5.50%	5.67%	7.87%	6.93%
Accuracy	22.42%	69.31%	96.94%	94.50%	94.33%	92.13%	93.07%

m	2048	4096	8192	16384	32768	65536
Error	4.89%	1.63%	0.16%	1.03%	1.66%	0.62%
Accuracy	95.11%	98.37%	99.84%	98.97%	98.34%	99.38%

Shakespeare's All's Well That Ends Well

As *m* increases, HLL counts better
Actual Cardinality 75161

m	16	32	64	128	256	512	1024
Error	47.10%	49.42%	29.77%	5.94%	5.27%	3.15%	1.97%
Accuracy	52.90%	50.58%	70.23%	91.06%	94.73%	96.85%	98.03%

m	2048	4096	8192	16384	32768	65536
Error	9.60%	1.09%	0.42%	0.37%	1.04%	0.21%
Accuracy	99.04%	98.90%	99.58%	99.63%	98.94%	99.79%

The Adventures of Sherlock Holmes

As *m* increases, HLL counts better
Actual Cardinality 18364

m	16	32	64	128	256	512	1024
Error	22.61%	14.47%	3.36%	6.71%	11.63%	4.73%	1.28%
Accuracy	77.39%	85.53%	96.94%	93.29%	88.37%	95.27%	98.72%

m	2048	4096	8192	16384	32768	65536
Error	2.48%	0.23%	0.37%	0.28%	0.04%	0.49%
Accuracy	97.52%	99.77%	99.63%	99.72%	99.96%	99.51%

Wuthering Heights

Figure 10: HyperLogLog Relative Error

while adaptive sampling fails for very high cardinalities.

8. REFERENCES

[1] Choosing a good hash function. `http://blog.aggregateknowledge.com/2012/02/02/choosing-a-good-hash-function-part-3/`, 2012. Retreived December 6, 2013.

[2] MurmurHash3. `http://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html`, 2012. Retreived December 6, 2013.

[3] The Twitter Streaming API. `https://dev.twitter.com/docs/streaming-apis`, 2013. Retreived September 24, 2013.

[4] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 449–460. IEEE, 2004.

[5] A. Dong, Y. Chang, Z. Zheng, G. Mishne, J. Bai, R. Zhang, K. Buchner, C. Liao, and F. Diaz. Towards recency ranking in web search. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 11–20. ACM, 2010.

[6] P. Flajolet. On adaptive sampling. *Computing*, 43(4):391–400, 1990.

[7] P. Flajolet, E. Fusy, O. Gandouet, F. Meunier, et al. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *Analysis of Algorithms 2007 (AofA07)*, pages 127–146, 2007.

[8] P. Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[9] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the EDBT 2013 Conference*, Genoa, Italy, 2013.

[10] B. Jenkins. Hash functions. September 1997.

[11] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of internet traffic. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 85–96. ACM, 2005.

[12] M. Wegman. Sample counting. Private Communication, 1984.