An Implementation of HyperLogLog++ in Thrill

Tino Fuhrmann Moritz Kiefer

April 10, 2017

1 Introduction

Counting the number of distinct elements in a dataset is a problem with an evergrowing number of applications such as counting the number of distinct visitors on a website. A naive solution to this problem is to simply insert all elements of a dataset into a set datastructure and then count the size of that set afterwards. However, this requires space proportional to the number of distinct elements and is therefore not suitable for very large datasets. In most applications, e.g., counting the number of distinct visitors on a website, exact counts are not necessary and a certain error is acceptable. Approximate counting algorithms exploit this by trading accuracy of the result for a reduced space-usage. In this project, we implemented *HyperLogLog++* [5], an extension of the popular HyperLogLog [4] algorithm inside the *Thrill* [2] framework.

2 HyperLogLog++

The main idea of *HyperLogLog* is to only consider hashes of the values in a dataset and then estimate the number of distinct elements based on these hash values. This is done by using the maximum number of leading zeroes in all hash values as an approximation of the logarithm of the number of distinct elements. In the following we assume that h is a uniform hash function, i.e., bits of hashed values are assumed to be independent and to have each a propability of $\frac{1}{2}$ of occuring [4].

We use $\text{clz}(x)$ to denote the number of leading zeroes in x plus one. For a set N containing n distinct elements the maximum number of leading zeroes $e = \max_{x \in N} \text{clz}(h(x))$ provides a rough estimate of $\log_2 n$. However, this estimate can be skewed significantly. To reduce the high variability of this approximation, the input stream N is split into $m = 2^p$ substreams N_i based on the first p bits ("substream identifier") of the hashed value $h(x)$. p is a user-defined parameter and is called the *precision*. A higher precision improves the accuracy of the cardinality estimate at the cost of requiring more memory.

For each substream N_i a so called *register* $M[i] = \max_{x \in N_i}(\text{clz}(h(x)))$ stores the maximum number of leading zeroes plus one of the hashed values in this substream. *HyperLogLog* stores all register values in an array and thereby requires space proportional to the number of registers $m = 2^p$. The individual estimates for each substream are combined using the harmonic mean shown in equation (1) producing the final cardinality estimate E.

$$E = \alpha_m \cdot m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]}) \tag{1}$$

However, for small cardinalities, the relative error of this estimate can still be quite large. *HyperLogLog* mitigates that by switching to *linear counting* if the cardinality estimate E is less than $5/2m$ and there exists at least one register equal to 0. Note that a register $M[i]$ is 0 if and only if no $x \in N$ has a substream identifier equal to i. Letting V be the number of registers equal to 0, linear counting estimates the cardinality as shown in equation (2).

$$E = m \log_2 m/V \tag{2}$$

HyperLogLog++ is an algorithm suggested by Google [5] which builds upon *HyperLogLog* but increases the accuracy of the estimates while also reducing the space necessary to store the register values.

Increased accuracy mostly relies on the following two improvements

1. *HyperLogLog++* uses 64bit hash values instead of the 32bit hash values used in the original *HyperLogLog* algorithm. This reduces collisions and thereby increases accuracy. This is especially important for estimating large cardinalities.

2. Empirically calculated bias values are added to the estimate calculated using the original *HyperLogLog* algorithm. This is especially important for cardinalities close to the threshold indicating when linear counting should be used. For each supported precision there is a sequence of bias values corresponding to specific cardinalities. The bias value is then chosen by interpolating between the bias values corresponding to the cardinalities near the original estimate. For the *Thrill* implementation in this project, we reused the bias values accompanying the presentation of *HyperLogLog++* [5].

To reduce space usage, *HyperLogLog++* uses a sparse encoding which stores only non-zero registers in a sorted list instead of storing the value of all registers. To make insertion of hash values efficient, they are first inserted in a second unordered list. This list is then periodically emptied and merged into the sorted list. A further reduction in space usage is achieved by encoding the sorted list using both a difference encoding and a variable length integer encoding. If the sparse encoding would require more memory than the normal representation, the sparse representation is converted to the dense representation, i.e., all register values are stored in an array.

Finally, *HyperLogLog++* uses a higher precision when storing registers in the sparse representation, thereby further increasing the accuracy of the result. When converting from the sparse to the dense representation, this additional precision is simply discarded.

3 Details of the Thrill implementation

3.1 Hash Function

Since *Thrill* is implemented in C++ it might seem like using the standard `std::hash` functions would be appropriate. However, `std::hash` maps integers to themselves. If a dataset contains values in some limited range instead of evenly distributed over all possible integer values, the high-valued bits of the hashes will be mostly the same. Since these bits are used as the substream identifier which decides which substream a value belongs to, most values will thereby be associated with a small number of substreams even if the selected precision allows for a significantly larger number of substreams. This drastically reduces the quality of the results and thereby makes `std::hash` not suitable for our use case. Instead our implementation uses *SipHash* [1] which more evenly distributes hashes even for values in a limited range but still performs well on small inputs such as integers. We have not experimented with other hash functions but replacing the hash function is an easy task.

3.2 Variable Width Integer Encoding

The sparse encoding used in our HyperLogLog implementation uses a difference encoding and a variable width integer encoding to compress data. While there is little in the implementation of a difference encoding, several variable width integer encodings exist. We have chosen to use the one already built into *Thrill* which uses the first bit in each byte to indicate if the following bytes belong to the same integer. However, we noticed during our experiments that the threshold at which the sparse representation takes up more space than the dense representation is lower than the one shown in [5]. One way to improve upon this is to use a group varint encoding [3] which uses a single byte to encode the length of the following four 32bit words. This encoding has been shown to be more efficient and to use less space than the varint encoding used in *Thrill*. Therefore it would be interesting to explore the use of group varint encoding in future work and thereby increase the threshold at which the sparse representation is converted to the dense representation.

4 Evaluation and Benchmarks

4.1 Relative Error

As *HyperLogLog* and its extension *HyperLogLog++* only provide approximate counts, an evaluation of the relative error is important to judge whether the error is acceptable in a particular application. In our evaluation, we generate a fixed number of random 64bit integers and calculate the relative error. While it might seem like evaluating only on 64bit integers is not sufficient, the result of the HyperLogLog algorithm depends only on the result of the hash function and using other types of data would only test the hash function which is not the goal of this project.

We performed two sets of tests to evaluate error rates. In the first we compared the relative errors using different precisions on five sample sizes. The results can be seen in Figure 1. They clearly demonstrate that using a higher precision increases the quality of the results: Using a precision of 4, the average relative error is about 0.1 while on the same data, the average relative error is below 0.01 using a precision of 14. The correct precision for an application is a trade-off between the quality of the results and memory usage and needs to be chosen based on the requirements of a specific application.

In the second set of tests, we performed more detailed tests on sample sizes up to 100000 using a fixed precision of 14 (14 is used to allow a comparison with the graphs shown in [5]). The results shown in Figure 2 demonstrate the increased precision using the sparse precision up to about the threshold around 5000 elements. There is a slight bump in the relative error around 18000 which is not present in the graphs in [5]. This is most likely caused by the fact that we did not calculate bias values ourselves but reused the ones provided in the paper introducing HyperLogLog++ [5] and the implementation used for the calculation of bias values might therefore slightly deviate from the *Thrill* implementation created for this project.

4.2 Performance

The performance of the implementation is an important metric to judge the usefulness in the context of different appplications. We evaluated the performance using two different tests.

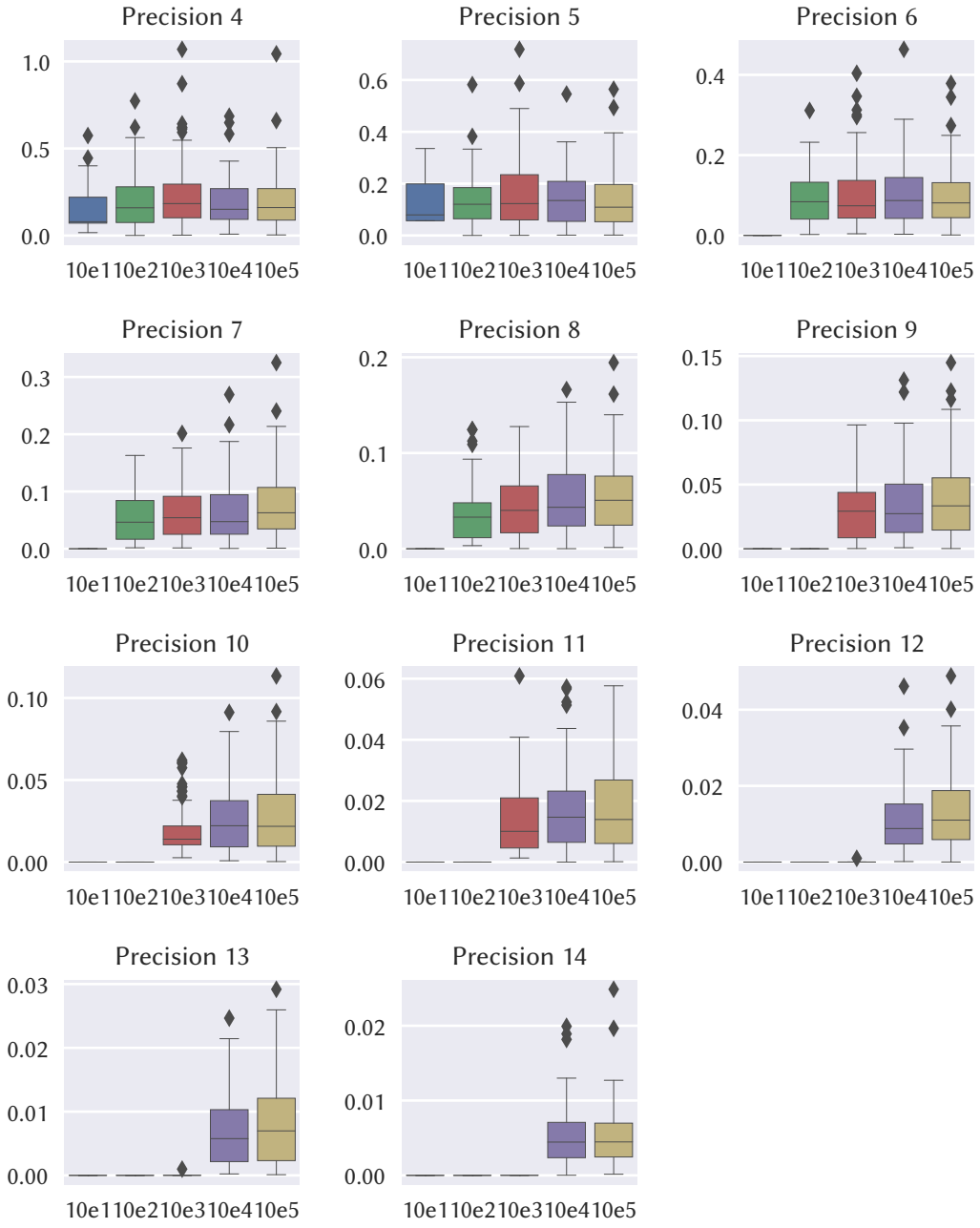

Figure 1: Relative error (y-axis) for different precisions and sample sizes (x-axis)

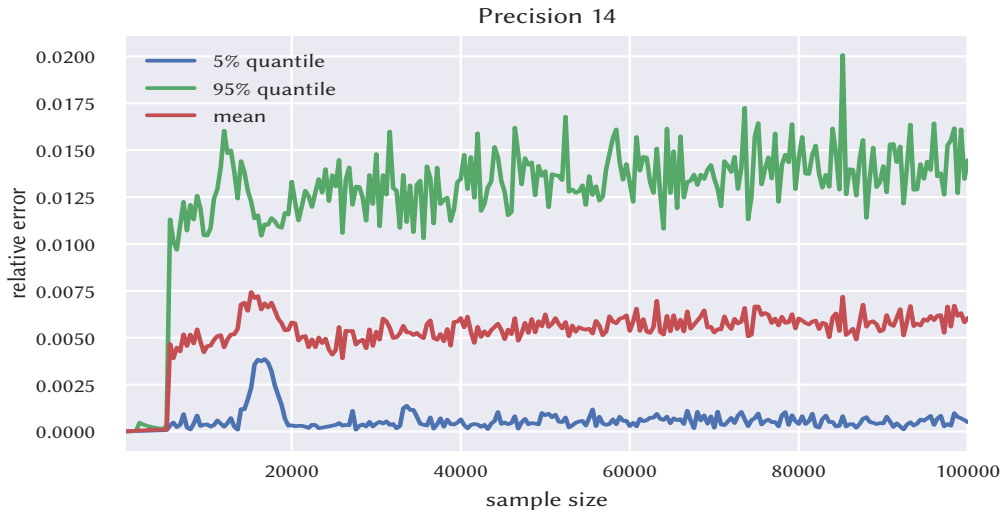

Figure 2: Relative error of our implementation using a precision of 14

In both tests we measured the mean execution time for a precision of 14. In the first test we used different sample sizes from 10 to 10^7 and in the second test we varied the number of used cores and measured the execution time for 10^6 elements. In order to establish a base line for the performance of our implementation, we ran the same tests using Apache Spark's *countApproxDistinct* function. We performed all tests on an 8 core Intel Xeon E312xx (2.6 GHz) with 16GB RAM.

The results of the first test can be seen in Figure 3. For up to a sample size of 10^4 the *Thrill*

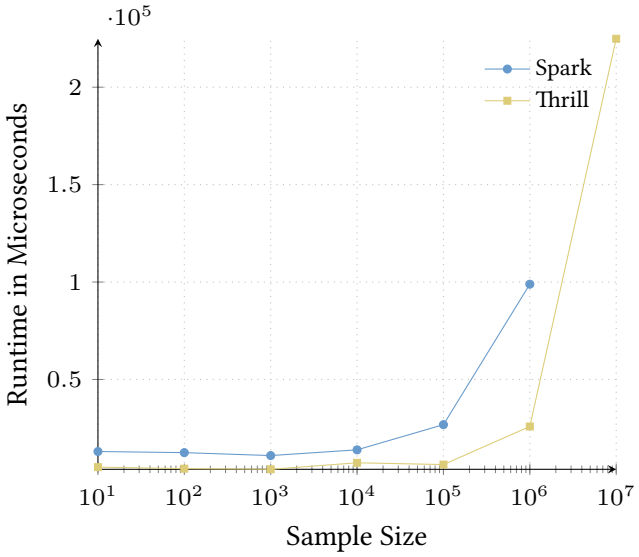

Figure 3: Runtime of Spark and Thrill for sample sizes from 10^1 to 10^7

implementation is about 2 times faster than Spark. For larger sets the execution time of Spark increases far more than the execution time of *Thrill*. We were not able to execute the Spark implementation with 10^7 elements due to high execution times. The *Thrill* implementation required 0.22 seconds to compute the estimated cardinality for 10^7 elements. Overall our implementation was about 4 times faster than the Spark implementation for sample sizes greater than 10^4.

The results of the second can be seen in Figure 4. Note that the scale of the y-axis of the two diagrams is different. Both implementations' performance increases significantly when using multiple cores. For example, our implementation's execution time decreases from 0.07 seconds using only a single core to 0.03 seconds using three cores. Similar behavior can be observed for Spark. An increase beyond 3 cores does not improve the performance of the Spark implementation: the execution time remains at about 0.1 seconds using three to eight cores. By contrast, the *Thrill* implementation's performance improves even after that threshold (although only very little): The execution time decreases from 0.03 seconds at three cores to 0.023 seconds at eight cores.

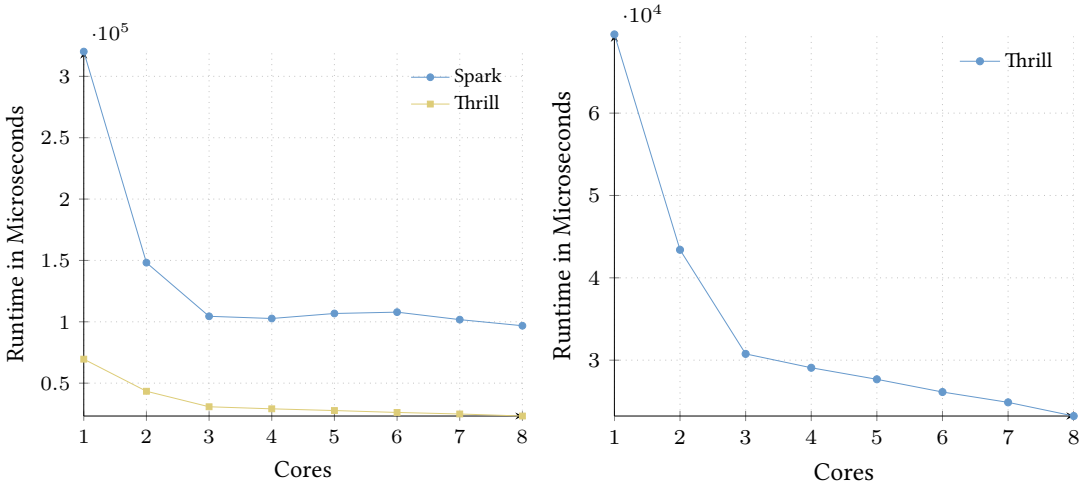

Figure 4: Execution time of Spark and Thrill for 10^6 elements using 1 to 8 cores

References

[1] Jean-Philippe Aumasson and Daniel J Bernstein. "SipHash: a fast short-input PRF". In: *International Conference on Cryptology in India*. Springer. 2012, pp. 489–508.

[2] Timo Bingmann et al. "Thrill: High-Performance Algorithmic Distributed Batch Data Processing with C++". In: *CoRR* abs/1608.05634 (2016). URL: http://arxiv.org/abs/1608.05634.

[3] Jeffrey Dean. "Challenges in Building Large-scale Information Retrieval Systems: Invited Talk". In: *Proceedings of the Second ACM International Conference on Web Search and Data Mining*. WSDM '09. Barcelona, Spain: ACM, 2009, pp. 1–1. ISBN: 978-1-60558-390-7. DOI: 10.1145/1498759.1498761. URL: http://doi.acm.org/10.1145/1498759.1498761.

[4] Philippe Flajolet et al. "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm". In: *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*. 2007.

[5] Stefan Heule, Marc Nunkesser, and Alex 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.