Approximate Large Multiset Cardinality Using Map Reduce

Kanth Kumar
Rochester Institute Of Technology
kkd6428@rit.edu

Suman Subash
Rochester Institute Of Technology
sbs1653@rit.edu

ABSTRACT

Cardinality estimation has extensive applications particularly in finding document similarities and database frameworks. Numerous algorithms have been published to estimate large cardinality and HyperLogLog is one of them.

In this paper, we discuss about research investigation on HyperLogLog, an algorithm to estimate large cardinality using 256KB memory. We have implemented an application to support Hyperloglog using Parallel Java 2 library. We will present the sequential and parallel design considerations for the application. Additionally, empirical data for strong scaling and weak scaling will also be discussed.

1. COMPUTATIONAL PROBLEM

Cardinality Estimation is the task of determining the number of distinct elements in the dataset. Using *O(n)* space, cardinality could be calculated but is impractical for larger datasets as it is limited by memory. Hence estimating large cardinality using only 256KB memory is very important.

2. RELATED WORK

2.1 Research Paper 1

LogLog[1] discusses about the use of hash functions to estimate large cardinality.

Algorithm 1 LogLog Algorithm Version 1

```
 1: procedure getCardinality()
 2:     Initialize: $max = -inf$
 3:
 4:     for each item $i \in N$ in dataset do
 5:         Compute $h(item) =< b_1 b_2 .. b_{32} >$
 6:         $\rho$ = Leading Zeroes from $< b_1 .. b_{32} >$
 7:         $max(curr\_max, \rho)$
 8:     end for
 9:
10:     return $2^{max}$ as cardinality;
11: end procedure
```

Algorithm 1 described above, computes a 32 bit hash for each item in the multiset. Subsequently it extracts the leading zeroes from the hashed items (ρ). The maximum seen leading zeroes is returned as the cardinality.

The problem with the above approach is that it is suscceptible to outliers. If we have one element in our multiset and the leading zeroes is 20. Then we would report the cardinality with larger relative error.

To overcome the above problem, the following changes are suggested in [1]

Algorithm 2 LogLog Algorithm Version 2

```
 1: procedure getCardinality()
 2:     Initialize: $max = -inf; p = 16; m = 2^p$
 3:     Initialize: $M[0]..M[m-1]$ to 0
 4:
 5:     for each item $i \in N$ in dataset do
 6:         Compute $h(item) =< b_1 b_2 .. b_{32} >$
 7:         Extract index of the register $idx =< b_1 b_2 .. b_p >$
 8:         $\rho$ = Leading Zeroes from $< b_{p+1} .. b_{32} >$
 9:         $M[idx] = max(M[idx], \rho)$
10:     end for
11:
12:     return $E = \alpha * m * 2^{\frac{\sum_{j=0}^{2^m} M[j]}{m}}$ as cardinality
13:
14: end procedure
```

Algorithm 2 provides a way to overcome the problems in Algorithm 1. The crux of the idea is as follows: Instead of using a single measure to track leading zeroes, multiple measurements are done using registers. Here, $m = 2^p$ registers are used to put the item to the right register. The first 'p' bits of the hash value is utilized as index to the register. The remaining 32 - 'p' bits are used to compute the leading zeroes (ρ). Each register has maximum leading zero count. Finally, arithmetic mean of the registers is computed and expected cardinality E is returned.

2.1.1 *Ideas from Research Paper*

From the above research paper, we took all the ideas presented in algorithm 2.

2.2 Research Paper 2

HyperLogLog [2] discusses about enhancing the large cardinality estimation by employing harmonic mean instead of arithmetic mean. This is because harmonic mean provides a better picture of the true average is the presence of outliers.

Additionaly, small range corrections and large range corrections are suggested which will be discussed below.

Small Range Correction: When there are few elements in

Algorithm 3 HyperLogLog Algorithm

```
1: procedure GETCARDINALITY()
2:     Initialize: $max = -inf; p = 16; m = 2^p$;
3:     Initialize: $M[0]..M[m-1]$ to 0
4:
5:     for each item $i \in N$ in dataset do
6:         Compute $h(item) =< b_1 b_2..b_{32} >$
7:         Extract index of the register $idx =< b_1 b_2..b_p >$
8:         $\rho$ = Leading Zeroes from $< b_{p+1}..b_{32} >$
9:         $M[idx] = max(M[idx], \rho)$
10:    end for
11:
12:    $E = \alpha * m * 2^{\frac{\sum_{j=0}^{2^m} M[j]}{m}}$ as cardinality
13:
14:    if $E < \frac{5}{2}m$ then
15:        let V be nbr of register equal to 0.
16:        if V is not 0 then
17:            $E^* = m \log \frac{m}{V}$
18:        else
19:            $E^* = E$
20:        end if
21:    end if
22:    if $E \leqslant \frac{2^{32}}{30}$ then
23:        $E^* = -2^{32} log(1 - \frac{E}{2^{32}})$
24:    end if
       return $E^*$ as the cardinality Estimate
25: end procedure
```

multiset, most of the registers $M[idx]$ will be zero. This usually results in incorrect cardinality estimation. For example, if there are 10 registers and all of them have zero value. Although, we haven't seen any item from the multiset yet, the estimated cardinality will be greater than 0. However, to overcome the problem a small range correction is needed which is to count the number of registers V which has zero value. Then apply the correction using line 17 (Algorithm 3) which is Linear counting cardinality estimate.Now, for the same example discussed above the estimate would be zero instead of positive non-zero value.

Large Range Corrections: As the number of distinct items in the multiset nears 2^{32}; hash collisions become frequent. Hence, large range corrections are performed.

2.2.1 Ideas from the paper

From HyperLogLog research paper, we have incorporated small range correction.

2.3 Research Paper 3

In the HyperLogLog++ research paper [3], there were many improvements suggested. But we will discuss only the ideas which we used in the project.

Large Range Corrections: In the HyperLogLog paper [2], as the distinct elements count is nearing 2^{32}, collisions increase thus resulting in a larger relative error for the estimates. In order to fix this, the paper suggested $E^* = -2^{32} log(1 - \frac{E}{2^{32}})$. But in the HyperLogLog++ [3] paper, instead of using a 32 bit hash function a 64 bit hash functions are used. Now hash collisions will occur only if the cardinality reaches $1.8 * 10^{19}$. But for all practical purposes, this number is more than sufficient.

Algorithm 4 HyperLogLog++ Algorithm

```
1: procedure GETCARDINALITY()
2:     Initialize: $max = -inf; p = 16; m = 2^p$
3:     Initialize: $M[0]..M[m-1]$ to 0
4:
5:     for each item $i \in N$ in dataset do
6:         Compute $h(item) =< b_1 b_2..b_{64} >$
7:         Extract index of the register $idx =< b_1 b_2..b_p >$
8:         $\rho$ = Leading Zeroes from $< b_{p+1}..b_{64} >$
9:         $M[idx] = max(M[idx], \rho)$
10:    end for
11:
12:    $E = \alpha * m * 2^{\frac{\sum_{j=0}^{2^m} M[j]}{m}}$ as cardinality
13:
14:    if $E < \frac{5}{2}m$ then
15:        $E^*$ =getLinearCount();
16:    end if
17:    return $E^*$ as the cardinality Estimate
18: end procedure
19:
20: procedure GETLINEARCOUNT()
21:    let V be nbr of register equal to 0.
22:    if V is not 0 then
23:        $E^* = m \log \frac{m}{V}$
24:    else
25:        $E^* = E$
26:    end if
27: end procedure
```

2.3.1 Ideas from the paper

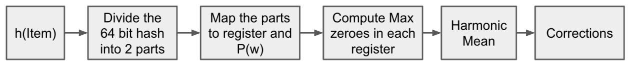
For the project, we use a 64 bit hash function. instead of 32 bit hash function as suggested by HyperLogLog++[3].

3. IMPLEMENTATION

The project uses Parallel Java 2 [4](PJMR - Parallel Java Map Reduce) library to estimate large cardinality.

3.1 Sequential Design

For our project, the sequential flow looks as show below:


h(Item)
Divide the 64 bit hash into 2 parts
Map the parts to register and P(w)
Compute Max zeroes in each register
Harmonic Mean
Corrections

1. First, we compute the 64 bit hash of the item.
2. We divide the hash into 2 parts p and 64 -'p' bits.
3. p bits is used as index to the register.
4. The value for the register index will be the leading zeroes from the 64 - 'p' bits.
5. We update the register index with leading zeroes if the number of leading zeroes is greater than the previous seen leading zeroes count.
6. Additionally, we compute the harmonic mean

7. And perform corrections such as linear counting to estimate the large cardinality with relatively less error.

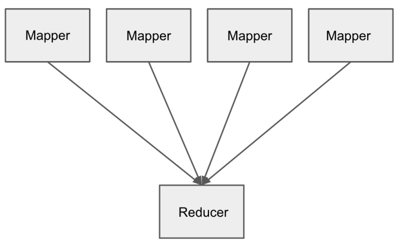
To estimate the cardinality we have used p = 16, which translates to $m = 2^{16}$ registers. If each register uses 8 bits, total memory consumption would be approximately 256KB memory usage.

3.2 Parallel Design

For the parallel design, we use parallel java map reduce framework part of the pj2 library. Below we can see the algorithm used for the project using map reduce.

Algorithm 5 HyperLogLog Map Reduce

```
procedure MAPPER()

    procedure START()
        $max = -inf; p = 16; m = 2^p$
        $M[0]..M[m-1]$ to 0
    end procedure

    procedure MAP()
        Compute $h(item) =< b_1 b_2 .. b_{64} >$
        Extract index of the register $idx =< b_1 b_2 .. b_p >$
        $\rho$ = Leading Zeroes from $< b_{p+1} .. b_{64} >$
        $M[idx] = max(M[idx], \rho)$
        add to combiner $< M[idx], max(M[idx], \rho) >$
    end procedure

end procedure

procedure REDUCER

    procedure REDUCE()
        Compute max leading zeroes for each M[idx]
    end procedure

    procedure FINISH()
        $E = \alpha * m * 2^{\frac{\sum_{j=0}^{2^m} M[j]}{m}}$ as cardinality
        if $E < \frac{5}{2}m$ then
            $E^* =$ getLinearCount();
        end if
        return $E^*$ as the cardinality Estimate
    end procedure

    procedure GETLINEARCOUNT()
        let V be nbr of register equal to 0.
        if V is not 0 then
            $E^* = m \log \frac{m}{V}$
        else
            $E^* = E$
        end if
    end procedure

end procedure
```



Mapper task:

- N items equally divided to each mapper
- Compute hash of the item
- Compute the index and maximum number of leading zeroes.
- Add register values to combiner

Reducer task:

- Reducer computes the harmonic mean out of the combiner values.
- Also remove the bias and perform corrections.

we divide the data across multiple nodes. The mapper task computes the hash of the item, leading zeroes. Now each mapper will output register index as the key and leading zeroes as the value to the combiner. Combiner will sort and perform partial reductions and the results are given to the reducer.
At the reducer, maximum seen leading zeroes will be taken for each register index. Then we compute the harmonic mean of the register values followed by small range correction.

3.3 Developer Manual

The sequential code has been tested on nessie and the parallel code has been tested on tardis cluster.

Before we start executing the project following configurations needs to be done. Set the classpath on both nessie and tardis using a bash shell.

1. export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar

2. Compile the source code *javac *.java*

3. Create a jar file *jar cf < filename.jar > *.class*

3.4 User Manual

To execute the code, the following commands could be used.
For Sequential: Pass the jar created in the previous section here. Name of the program we are executing would be HyperLogLogSeq. The command can be seen below:
java pj2 jar=<jar> HyperLogLogSeq <file> <pattern> <registerIndexSize>

For Parallel: Pass the same jar created in the previous section to pjmr.
java pj2 jar=<jar> HyperLogLogSmp <file> <pattern> <registerIndexSize> <node> [<node> ..]

1. <jar> : name of a JAR file containing all the Java class files

2. <file> : file/data-set to find the cardinality

3. <pattern> : pattern sequence in data-set whose cardinality has to be estimated

4. <registerIndexSize> : register index size, determines number of registers.

5. <node> : name of a cluster node containing dataset's chunk.

Size in MB	Cores	Running Time	Speed Up	Efficiency
N = 10	k=1	29	1	1
	k=2	14	2.071428571	1.035714286
	k=4	8	3.625	0.90625
	k=8	6	4.833333333	0.604166667
N= 20	k=1	49	1	1
	k=2	27	1.814814815	0.907407407
	k=4	15	3.266666667	0.816666667
	k=8	9	5.444444444	0.680555556
N = 40	k=1	103	1	1
	k=2	52	1.980769231	0.990384615
	k=4	27	3.814814815	0.953703704
	k=8	15	6.866666667	0.858333333
N = 80	k=1	204	1	1
	k=2	103	1.980582524	0.990291262
	k=4	52	3.923076923	0.980769231
	k=8	28	7.285714286	0.910714286
N=160	k=1	409	1	1
	k=2	208	1.966346154	0.983173077
	k=4	104	3.932692308	0.983173077
	k=8	54	7.574074074	0.946759259

Table 1: Strong Scaling

Size in MB	Cores	Running Time	Speed Up	Efficiency
N = 2.5	k=1	8	1	1
	k=2	8	2	1
	k=4	9	3.555555556	0.888888889
	k=8	9	7.111111111	0.888888889
N = 5	k=1	14	1	1
	k=2	14	2	1
	k=4	14	4	1
	k=8	18	6.222222222	0.777777778
N = 10	k=1	29	1	1
	k=2	27	2.148148148	1.074074074
	k=4	27	4.296296296	1.074074074
	k=8	28	8.285714286	1.035714286
N = 20	k=1	51	1	1
	k=2	52	1.961538462	0.980769231
	k=4	53	3.849056604	0.962264151
	k=8	53	7.698113208	0.962264151
N = 40	k=1	102	1	1
	k=2	103	1.980582524	1.017114914
	k=4	104	3.923076923	1.017114914
	k=8	104	7.846153846	1.056234719

Table 2: Weak Scaling

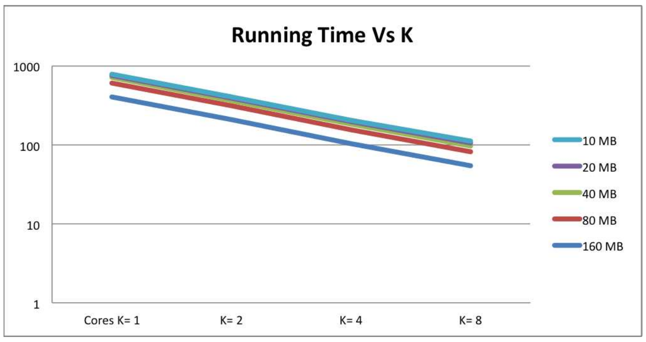


Figure 1: Running Time Vs K

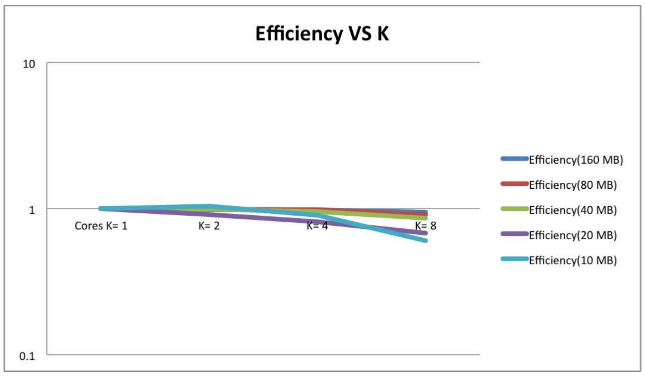


Figure 2: Efficiency Vs K

4. PERFORMANCE

Strong Scaling performance was measured with different input sizes starting from 10 MB. Running time decreased proportional to the number of cores increased. Thus providing ideal Strong Scaling.

In figure 1,for different data sizes, running time reduced proportional to the number of cores used.
In figure 2, the efficiency is very close to ideal. But if we increase the number of cores when the size of the dataset is small, the efficiency no longer remains ideal. This happens due to thread synchronization.

Weak Scaling performance was measured by doubling the size of the datasets. When the size of the input and the number cores are doubled the running time remained constant. Thus providing ideal weak scaling.

Figure 3 shows the weak scaling performance for different data sizes. Running time has remained constant when the size of the data set is increased along with number of cores. But when the size of the data is too small ($<$ 5MB) running time fluctuates as seen in the figure 3.

Figure 4 presents insights into weak scaling performance. Ideal weak scaling is obtained as we increase the size of the data and the number of cores. But when the data sizes are too small ($<$ 5MB) effiency drops slightly due to the overhead of thread synchronization.

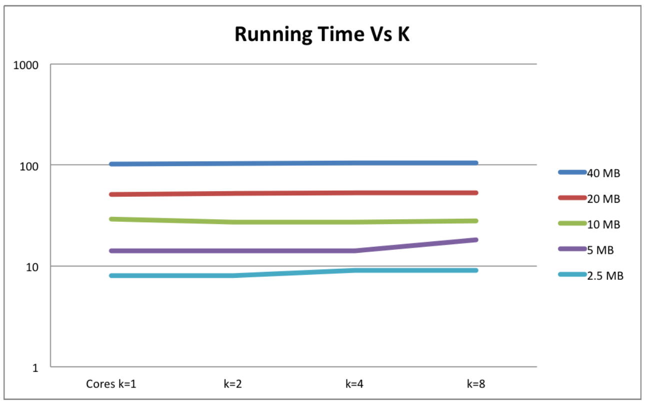


Figure 3: Running Time Vs K

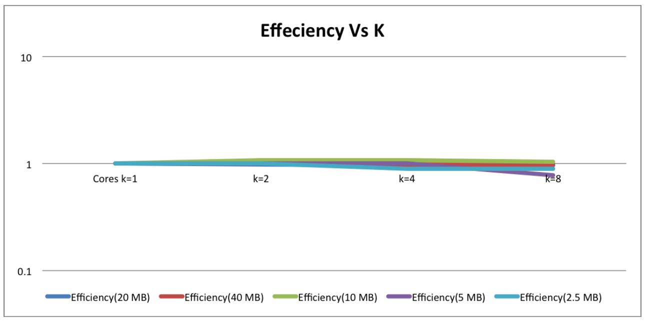


Figure 4: Efficiency Vs K

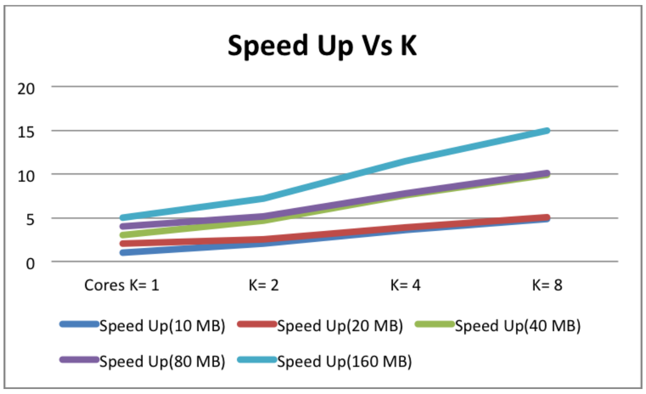


Figure 5: Speed Up Vs K

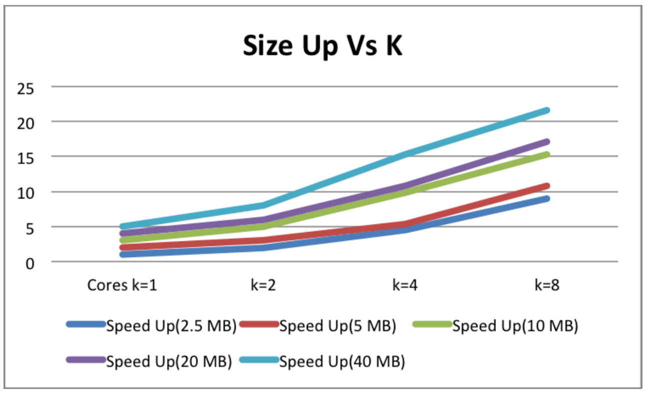


Figure 6: Size Up Vs K

5. FUTURE WORK

There are many improvements which could be applied to the HyperLogLog algorithm described above. Some of them include maintaining a higher precision sparse matrix with compression. As part of the project we also developed an application used to measure Jaccard similarity of two documents. We noticed that cardinality estimation of set intersection for two documents had large relative error if one of the documents was relatively smalled in size. Improvements could be done on set operations as there are many use cases.

6. LEARNING'S

1. We learned about the cardinality estimation of large datasets with little memory.
2. Map reduce,for HyperLogLog, provides ideal Strong Scaling and Weak Scaling for large datasets.
3. Strong Scaling and Weak Scaling will not be ideal if we increase the number of cores when the data sizes are relatively small.
4. Large cardinality could be estimated with relative error less than 1%.

7. TEAM CONTRIBUTION

Parallel implementation was written by both Kanth and Suman. Sequential implementation of the code was written by Kanth. Additionally, the dataset preparation for different test cases was done by Kanth. HyperLogLog Application for calculating Jaccard Index to measure document similarity was done by Suman. Measuring the Strong Scaling and Weak Scaling for different test cases and plotting Log-Log plots for the same was done by Suman. Presentations and Final report were done by both Kanth and Suman.

8. REFERENCES

[1] M. Durand and P. Flajolet. Loglog counting of large cardinalities. *European Symposium on Algorithms (ESA), volume 2832*, (5):605–617, 2005.

[2] O. Gandouet P. Flajolet, Eric Fusy and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *In Analysis of Algorithms (AOFA)*, pages 127–146, 2007.

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

[4] A. Kaminsky. Parallel java: A unified api for shared memory and cluster parallel programming in 100*RIT Scholar Works*, 2007.