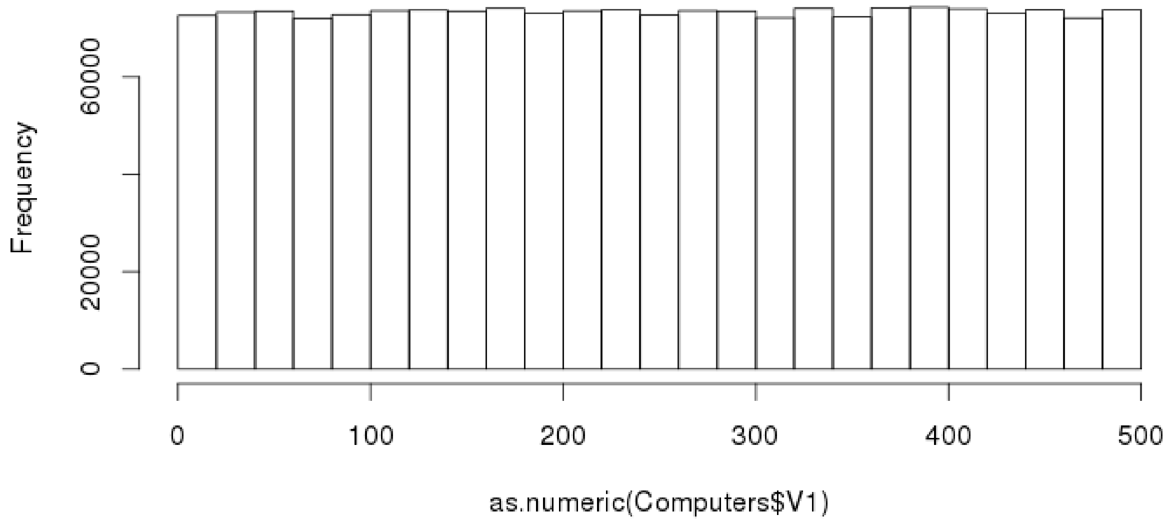
MapReduce - Assignment 2

Sahil Mehta, Behrooz Afghahi

Dataset Properties

To better understand the requirements and to find right method of median approximation we first analyzed the given dataset to see which method will provide the best approximation.
Based on our analysis, we found that the dataset is probably an artificially created data with random price numbers between $0 and $500. Furthermore the distribution of categories is equal, meaning if we read 10,000 records from the file and we have 18 categories we expect to see 10,000÷18 records from each category. An example of the data distribution follows.



Median Approximation

Based on our research we found two methods to improve our existing implementation of exact median. Both methods calculate an approximate median. Here we discuss both methods and the reason why we chose one over the other.

Method A: Histogram binning and sampling

In this method we would try to find the distribution of data and generate a histogram of price ranges in each of the mappers. Afterwards each mapper will find the bin that it believes holds the median (which in our dataset will be very close to the real median).

This approach requires the definition of bins, whether the bin sizes are uniform and also finding an optimal bin size. In our dataset (given the uniform distribution) we can simply define a fixed bin size. We also need to know how the approximate minimum and maximum of the data.
The reducer then will receive all perceived medians from the mappers and report the median for each category.

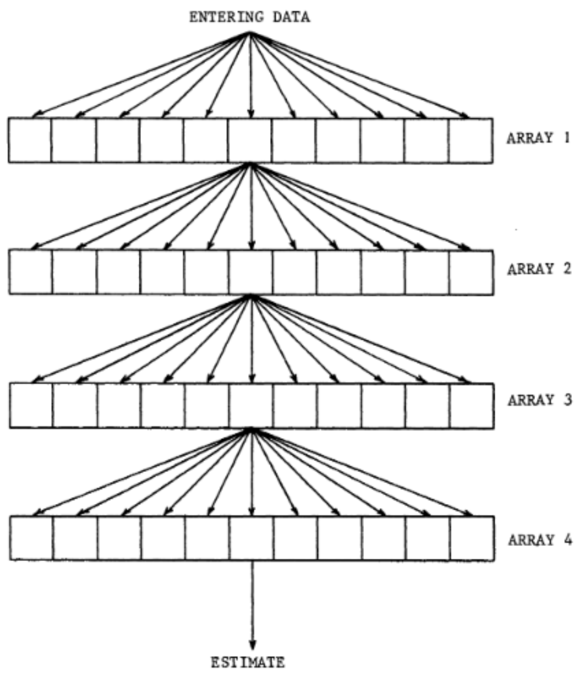
Pros

1. Minimizes the list of values the reducer receives.
2. very low profile mapper and reducer code

Cons

1. Requires knowledge about the underlying distribution which in the general case requires two passes over the data or keeping a portion of the data in memory as sample (under the assumption of uniform data distribution).
2. Requires finding optimal values for bin size for each dataset (which might change over time).
3. The distribution of the data might change over time introducing calculation errors that will go unnoticed.

After evaluating this method we decided against using it.



Method B: Remedian Calculation (used)

Remedian is a median calculation algorithm suitable for both streaming (our case) and fixed sized data. The original proposed in 1989 [1] and further analyzed in 2011 [2]. In this method a multiple levels of fixed sized bins are used. As the inputs come in, we fill the first level. Once the first level is full, we calculate the median of it and put it in the first empty slot of the next level. We continue this approach until we've seen all the data. The median of the last level is the approximation we're looking for.
The proof of correctness and error tolerance can be found in both papers [1] and [2] of references.

With this method we can cover the whole dataset using 4 arrays of size 101. The reason behind the even number is that the calculation of the median for each step is then simplified (we simply sort and pick the middle element).

Pros

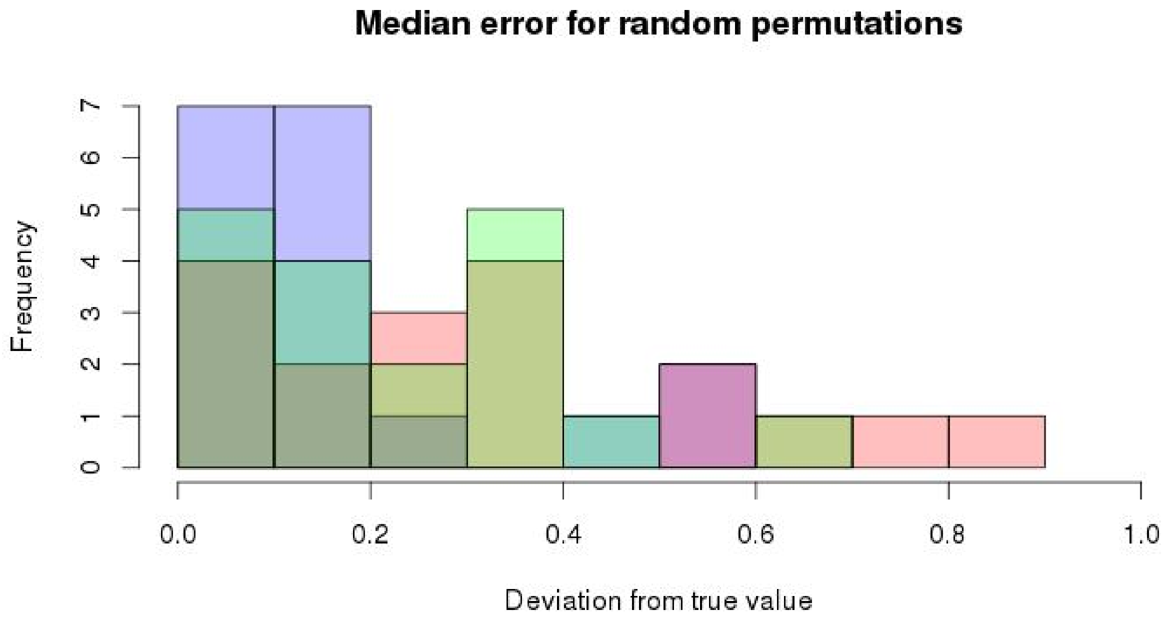
1. Fast algorithm which does not require any prior knowledge about the distribution of data
2. Minimizes the list of values the reducer receives.
3. As Discussed further down, it doesn't require parameter estimation.
4. Requires a single pass over the data.

Cons

1. Doesn't work well with very small bin sizes (5 and less) or very small datasets.

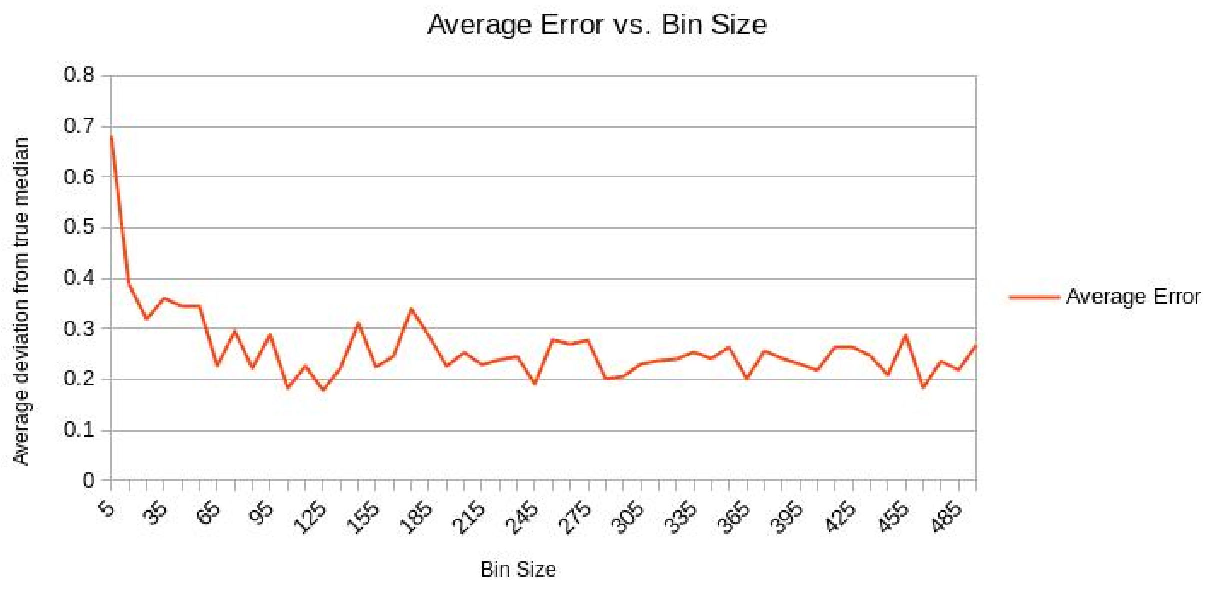
After implementing this method we did two tests to analyse the correctness.

Error rate (deviation from the exact median) if we permute the dataset randomly:



Here we see that regardless of permutation (*blue:original, red:random permutation 1, green: random permutation 2*). We get very low error rates (less than 0.2%).

Error rate vs bin size:



Here we see that the algorithm is very tolerant of bin sizes, where any choice of bin size above 60 give roughly the same results.

Local Performance

As a comparison of this new version, v4, with older versions on a local machine[1] the following table is provided. AWS performance is mentioned in the next section.

	V1 – Java	V2 – Pseudo	V3 – Pseudo	V2 – Single	V3 – Single	V4– Single
Run 1	64.13	168.15	472.7	163.04	469.77	60.43
Run 2	65.62	173.26	457.16	161.14	450.62	64.21
Run 3	64.57	169.69	466.38	156.91	461.3	62.48
Run 4	63	168.34	469.06	156.32	451.2	62.34
Run 5	63.92	171.67	473.1	163.28	453.38	65.33
Average	**64.248**	**170.222**	**467.68**	**160.138**	**457.254**	**62.96**

[1] 64bit Linux in idle state; 4 cores; 8Gb memory

AWS Runs:

Run Configurations:	**Run Time:** 287.23 (avg)
2 core instances / 7 reducers / c3.xlarge :	256 seconds
2 core instances / 3 reducers / m1.small :	500 seconds
2 core instances, 2 task instances / 1 reducer / c1.xlarge :	178 seconds
2 core instances, 2 task instances / 6 reducer / c1.xlarge :	215 seconds

References

[1] The Remedian: A Robust Averaging Method for Large Data Sets; Peter J. Rousseeuw and Gilbert W. Bassett, Jr.*; link
[2] Further analysis of the remedian algorithm; Domenico Cantone, Micha Hofri; link