Approximating Frequency moments from the data stream

Data Mining project T61.5060 - Aalto University 2015
by Kunal Ghosh 546247 and Jussi Ojala, 544605

Introduction

The target of the project was to calculate the frequency moments F0 (number of distinct elements) and F2 (surprise index / diversity of elements) from the sequence of elements and analyse the time and space complexity of selected algorithms. These frequency moments serve as a basis for algorithm or data storage decision when the amount of the data is very big. Our main target is to concentrate on algorithms which can handle streaming data i.e. single pass algorithms to calculate approximations of the moments

The data used in this project were file of Twitter messages. The file was collected over many months and containing different marks and languages. The compressed file size was 5.14 GB and the file included about 912 Million lines. We considered one line as a one instance in a sequence.

For the evaluation purpose and to get better understanding of the data content we calculated the exact values of the considered moments. For exact calculations we compared the space requirements of single pass algorithm to the multipass algorithm (which we can not normally use due to the streaming nature of the data or due to very slow access to the stored data). By default the multipass approach required more time, however it saved quite a lot of space compared to single pass approach.

The main task of the project were to consider single pass algorithms to approximate the frequency moments. These algorithms are used to relax the space requirements that would be required to calculate the exact value with single pass algorithm. If the knowledge is used afterwards for example efficient algorithm design to further study the properties of the data reasonable approximation of the moments is good enough to help in algorithm

decision. For F0 approximation we used algorithm introduced in [Flajolet and Martin] and for F2 approximation algorithm introduced in [Alon, Matias, Szegedy].

F0 approximation algorithm is based on hash function and it's properties. We tested couple of different hash function choices. We compared their randomisation properties, space requirement and time complexity with smaller data size. The results achieved with approximation were compared to the correct values to get estimate of the error and study improvements. We combined several hash functions of same family and tested couple of combination strategies and selected our candidate. We selected simple RSHhash based hash function family with 216 different prime number pairs. We improved our final results by post processing. We grouped the results calculated the mean of medians and averaged that over 100 random permutation of different groups.

For F2 the paper [Alon, Matias, Szegedy] presented two algorithms. Instead of selecting the more space efficient version of algorithm F2 proposed in the paper, we used the book [Leskojev Rajaram Ullman] version of F2 approximation also presented in the paper. The F2 algorithm is based on calculating several random variables from the stream and using mean median trick to them. We assumed that the required number of samples could be a lot smaller than the values used in the paper to proof that the algorithm works. This was also the case in this particular setting we studied.

Methods and selection process

In this section we go through our study process and briefly introduce the method we were using. First we show how we computed the exact values and what was the main finding in there. Then our selection process of the hash functions, where we studied properties of couple of different candidates. At the end we show how we decided to rely on much smaller sample sizes on F2 algorithm that was proposed in the proof of the algorithm.

Exact computation

For the F0 calculation we initially wrote a linux shell script to sort the values in the stream and then find the unique elements. The sorting was quite memory intensive (>10GB memory utilization) so we had to resort to shell command parameters to cache the data

to disk, this resulted in significant reduction in performance (Took around 14 hours to calculate F0).
However, while implementing the code for F2 we decided to implement F0 too, to get a better estimate of the memory utilization.
We implemented a simple hash table (unordered_map in C++ and dictionary in Python) based algorithm in C++ and Python to compare their relative execution speeds. Using basic optimization techniques (using std::ios::sync_with_stdio(false) in C++ and read directly from sys.stdin in Python) gave us comparable execution times in both Python and C++. This reduced the time requirement to less than 11 minutes and memory requirement dropped below 2GB.

By using a map we are indeed getting the exact count of the number of distinct elements because, most modern map implementations use hash tables with some sort of collision resolution technique (collision chaining in case of C++ unordered maps [ref]).

Implementation:

- Single Pass
    - source/exact/single_pass/test.cpp
    - source/exact/single_pass/test.py
- Multi pass
    - source/exact/multi_pass/run_multipass_full.sh
    - source/exact/multi_pass/exactMultiPass.cpp

Approximate computation

For calculating the approximate value of F0 we decided to implement the algorithm proposed in [Flajolet and Martin]. To select correct hash function we compared the performance two simple hash functions RSHash, Polynomial Hash and a relatively more elaborate MurmurHash Function. We also evaluated DJB2 and a simple sum of character bytes mod prime based hash, but dropped them early out due to poor randomisation properties they had.
Implementation:

- Hash Functions
    - source/approximate/hashes.cpp
    - source/approximate/hashes.h

For F2 we decided to implement the algorithm described in [Alon, Matias, Szegedy] section 2.1 since it was relatively easy to understand and implement. For starting point we chos λ = 0.35 and ϵ = 0.05 hence we would need 4Million samples and find median of (s2) = 8 variables each of which is an average of (s1 = 0.5M)

samples. In practice we averaged over (s1)/10 samples and used (s2) = 10 which gave us quite a good approximate. We also evaluated how much smaller sample would have been possible to take, since the sample requirement of this algorithms looked quite big.

Implementation:

- F0 approximation
  - source/approximate/f0_args.cpp
  - source/approximate/calculate_f0.sh
  - source/approximate/calculate_f0_bits.sh
- F2 approximation
  - source/approximate/f2_new2_1.py
- "Averaging Data"
  - source/approximate/mean_of_medians.py

Flajolet-Martin algorithm for distinct elements (m) in sequence (F0)

```
------------------------------------------------------------
FLAJOLET - MARTIN ALGORITHM (as implemented by us)
------------------------------------------------------------
  ❑ For seed in SEEDs
      ❑ APPROXIMATION = list()
      ❑ BITMAP[i] := 0 for all i
      ❑ for all line in Stream do
          ❑ index := rho(hash(line, seed))
          ❑ BITMAP[index] = 1
          ❑ end
      ❑ R := least_Significant_Zero(BITMAP)
      ❑ APPROXIMATION.append[ 2^R / 0.77351 ]
  ❑ APPROXIMATION := permute(APPROXIMATION)
  ❑ MEDIANS = list()
  ❑ for subset in subsets(APPROXIMATION,5) do
      ❑ MEDIANS.append(median(subset))
  ❑ Return mean(MEDIANS)
```

The algorithm proposed in [Flajolet and Martin] is based on the usage of hash function. The usage of the algorithm requires selection of the hash function, number of combined hashes, the combination strategy and the selection of the length of the bitstream.
The bitstrings elements of the "universal" set (lines in our file) are hashed to bitstring. The length of bitstring L need to be sufficient so that there are more outputs than there are elements in "universal" set. However we do not know the cardinality of our

universal set and we do not want to estimate that with the max number of characters in one line since this is too big overestimate. We selected L=64 which would be biggest easily available, however in this case when we knew that there are about 1000M lines to study even L=32 would have been enough.
The algorithm is based to find the "unusual" results of hash function i.e. the maximum tail length R (number of consecutive 0 of least significant bits). It is shown in the paper of [Flajolet and Martin] that the probability that 2^R is far from the real value m is small when the length of the sequence is big enough and the hash function has the perfect property to distribute uniformly lines to the whole space of the bitstream.
We do not have formula to estimate the accuracy of our estimate, but we know that accuracy can be improved always by collecting more estimates and calculating mean or median of them. We note that 2^R has the problem of only producing values of power of 2, further we know from theory that 2^R can do big overestimates of the m with small probability. WE choosed approach to combine mean and median. We select 65(216) different hash function of one family. To enable all possible values the mean should be taken over x*log(m), however we assumed that the approximation is good enough with the mean of around 10 medians to keep the running time reasonable. We also thought that taking the median first to get rid of possible big overestimates could be better strategy.

To get better understanding of this combination of estimates to groups and taking mean and median of the groups we compared the results of changing the order of mean vs median. Based on results we find out that with this limited size of 65 different hash function the and 6 size 10 groups taking the median and the mean of medians gave better results. To further reduce the effect of small groups we took 100 random groupings and calculate our final estimate as average of results from those 100 random permutations.

Comparison of Hash Function randomisation properties

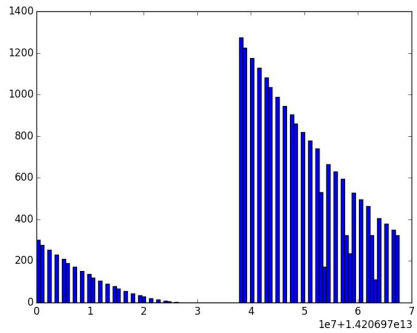
We implemented a few hash functions and following are their relative spreads achieved when run over a universe of strings. This universe was generated by taking english characters (26 lower case + 26 upper case) and getting 3 and 4 characters combinations respectively (52 Choose 3 strings and 52 Choose 4 strings respectively). The character combinations were concatenated using a hyphen (-).

- Code to Evaluate hash Functions (Including plotting code)
    - source/approximate/hash_collisions/get_plots.sh
    - source/approximate/hash_collisions/generate_strings.py

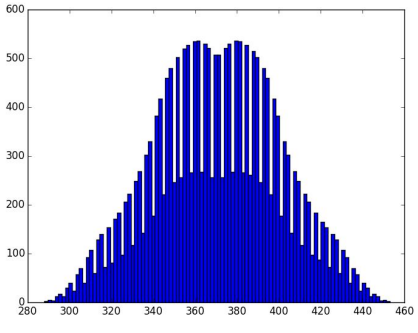
- source/approximate/hash_collisions/get_hashes.cpp
- source/approximate/hash_collisions/plot_data.py

3 Character Combinations

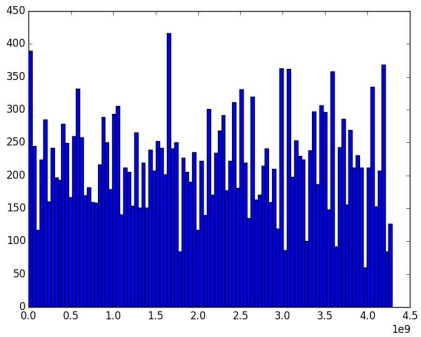
From these figures we can see that Murmur2, murmur3 and RSHash have clearly better randomisation properties than DJB2, ModPrime operation and polynomial hash.



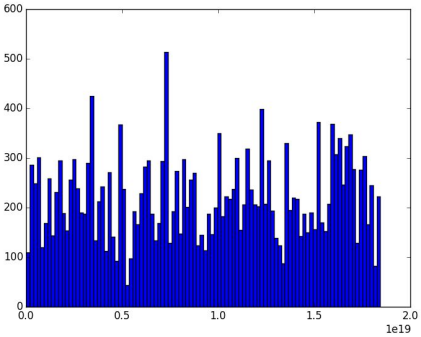
DJB2



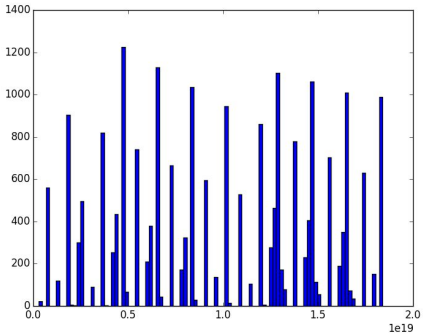
SUM_CHARS_MOD_PRIME



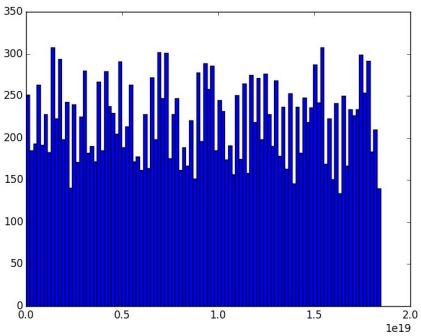
Murmur2 32bit



Murmur3 64bit



Polynomial Hash

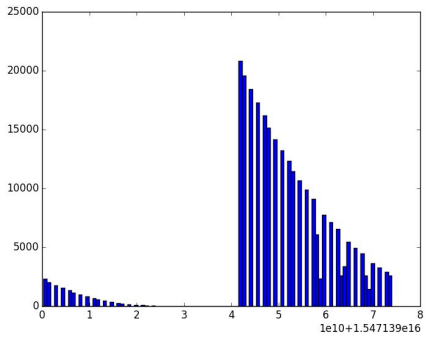


RSHash

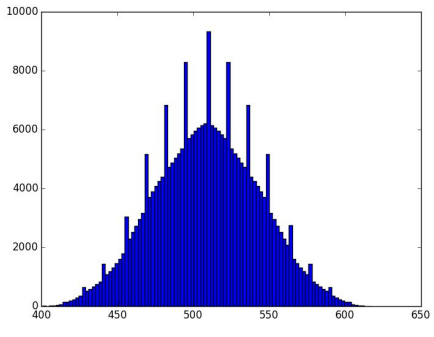
4 Character Combinations

The same phenomenon than with 3 character combinations can be seen with 4 character combination, except polynomial hash function

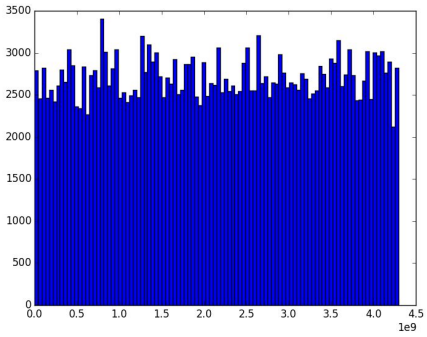
properties has improved. Thus we decided to drop DJB2 and mod_prime based approaches and not to study them further.



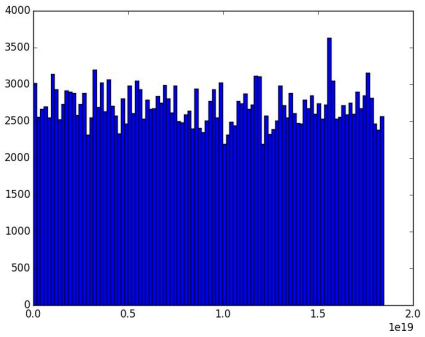
DJB2



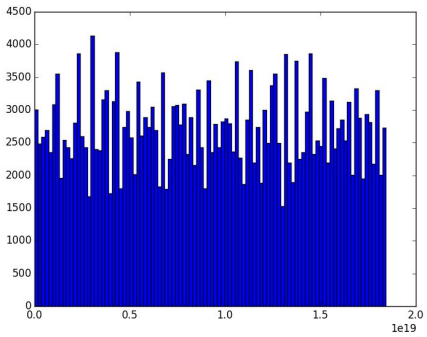
SUM_CHARS_MOD_PRIME



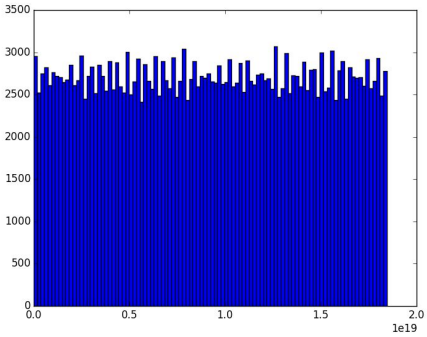
Murmur2 32bit



Murmur3 64bit



Polynomial Hash

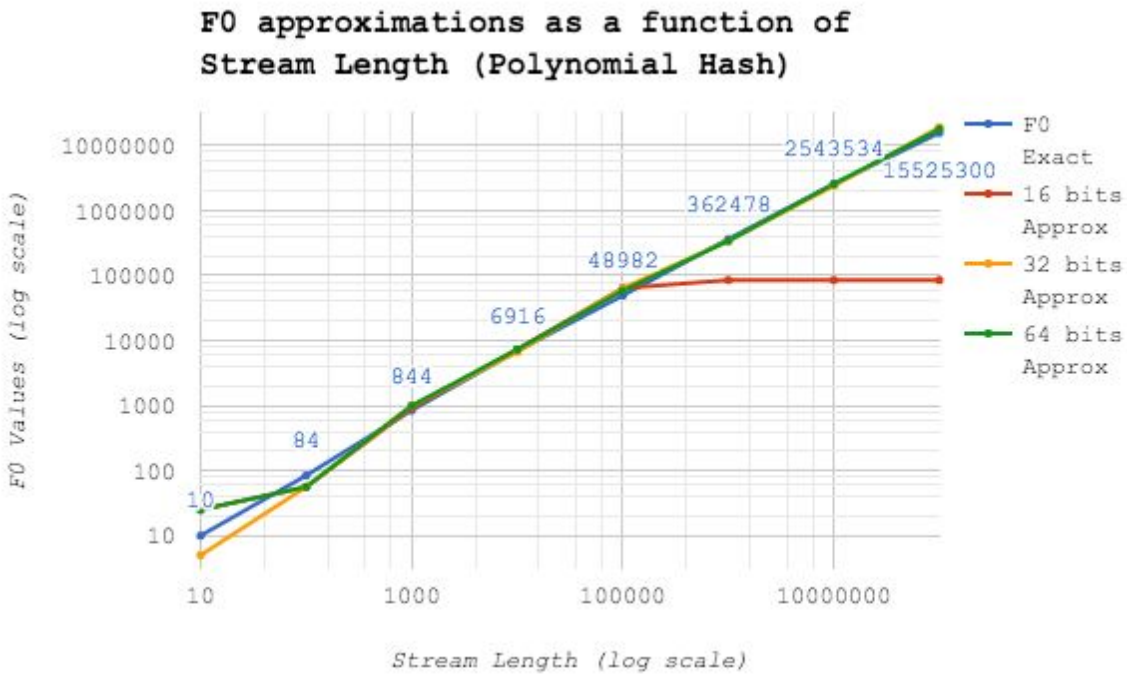


RSHash

Number of required bits for the hash function

In this section we briefly study the effect of the hashing techniques as a function of bits used. We take the example calculations for Polynomial hash function (we assume the RSHash to have similar behaviour) and murmur hash. It is clear from figure below that 16 bits is not enough. It will limit the approximation to m < 2^16/0.77315. In theory 32 bit should be enough and if we would like to be very space efficient that could be our selection. However this time we selected 64 bit to be sure that the space limitation would not be affecting accuracy of our final results.

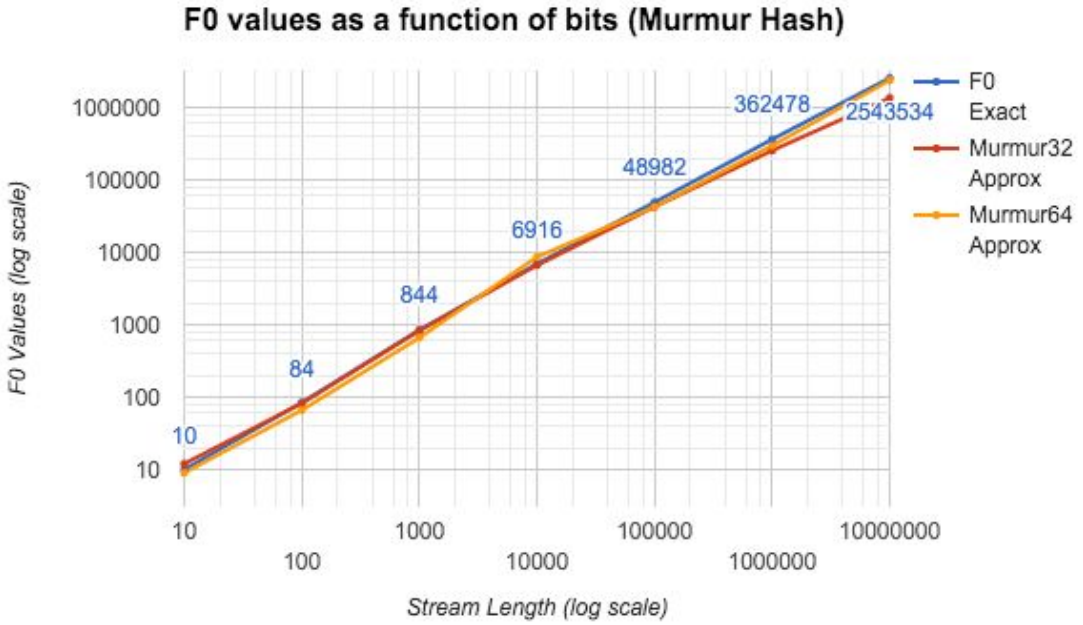
We also noticed that the standard deviation of individual hash function is very big thus we decided to increase number of hash functions to 216.



F0 values approximated using polynomial hash and varying bit lengths.(Values in Blue Indicate the exact F0) Refer to Table 3 below for exact values.

Polynomial Hash		16 bits			32 bits			64 bits		
N(stream size)	F0 Exact	16 bits Approx	STD	Time(s)	32 bits Approx	STD	Time(s)	64 bits Approx	STD	Time(s)
10	10	25	15	0.7	5	0	2.4	25	15	1.1
100	84	56	51	0.8	56	51	1.0	56	51	0.9
1000	844	909	1505	1	992	1505	1.1	992	1505	1.1
10000	6916	7280	6924	2.4	6618	6924	3.4	7280	6924	3.4
100000	48982	63543	27247	16.0	63543	72708	25.8	58248	72708	25.8
1000000	362478	84725	0	151.9	338901	280370	295.2	338901	280370	291.9
10000000	2543534	84725	0	1887.8	2291591	171221	3283.0	2541764	2291591	3045.1
100000000	15525300	84725	0	4877.7	18978505	19117567	4865.4	17622897	NA	NA

Table 1: Comparison of the F0 approximation values when using the Polynomial hash family. All approximations are mean of 5 values each of which is a median of 13 values. Totally 65 runs with different prime seeds were used to get the results and the Time (cumulative) and STD have been calculated for the 65 results.



F0 values approximated using Murmurhash and varying number of bits (32 and 64).
(Values in Blue Indicate the exact F0)
Refer to Table 4 below for exact values.

Murmurhash		32 bit			64 bit		
N(stream size)	F0 Exact	Murmur32 Approx	STD	Time(s)	Murmur64 Approx	STD	Time(s)
10	10	12	11	0.94	9	10.3	1.23
100	84	82	106	0.97	66	108	1.59
1000	844	826	911	1.26	661	898	1.23
10000	6916	6618	8047	3.29	8604	7222	3.32
100000	48982	42362	30034	23.49	42362	30457	24.45
1000000	362478	254175	317404	268.43	296538	442529	275.78
10000000	2543534	1355607	3220065	2930.35	2372313	2518415	3012.7

Table 2: Comparison of the F0 approximation values when using Murmurhash family (32bit and 64bit variants). All approximations are mean of 5 values each of which is a median of 13 values. Totally 65 runs with different prime seeds were used to get the results and the Time (cumulative) and STD have been calculated for the 65 results.

Calculation of Space complexity of hash functions used.

Murmur2 32 bit hash function always hashes input strings to a 32 bit Number and Murmur3 64 bit hashes the strings to a 64 bit Number.

However, the space complexity of RSHash, which takes in 2 prime numbers apart from the input string, can be expressed as a

function of the input string length (N bytes) , and the number of bits used (we present the upper bound, because we are considering that using p bits highest representable value is $2^{p+1}-1$ however in reality the values might be lower.) to represent the two prime numbers (m) and (p).

$$log_2\left(2^8\left[\sum_{i=0}^{N-1} 2^{m(N-1-i)+\left(\frac{N(N-1)}{2}-\frac{i(i-1)}{2}\right)p}\right]\right)\%\ 64$$

Similarly for a string of N bytes, using a seed of p bits. The space complexity of the hash function (bits used to represent a string of length N bytes) is

$$\left[8\ +\ \frac{N(N-1)}{2}p\right]\%\ 64$$

We are calculating the modulus 64 value because in our implementation we have used 64 bit data types. In practice this should be changed with the length of the data type used.

(derivations in appendix)

Comparison of F0 approximations using various hash functions and different group averaging techniques.

We calculated the exact values and compared them to the approximation achieved with different combination strategies of 65 hash function of the same family. We small file sizes to test our strategies and included some results to the Table 1 and 2 below. From the results it was clear that the family of RSHash function provided best results. It was fastest algorithm to run and the accuracy was best among this small test set. Moreover even the space complexity was better with 64 bit RSHash compared to the Murmur64.

N(stream size)	F0 Exact	F0 Approximation							
		Murmur32				Murmur64			
		mean of medians from 5 groups.	median of means from 5 groups.	mean of 65 such seeds	Time (s)	mean of medians from 4 groups.	median of means from 4 groups.	mean of 65 such seeds	Time
10	10	12	11	13	0.94	9	9	10	1.23
100	84	82	127	126	0.97	66	82	89	1.59
1000	844	826	1116	1167	1.26	661	1029	1030	1.23

10000	6916	6618	8935	9205	3.29	8604	9529	9469	3.32
100000	48982	42362	48319	47250	23.49	42362	48124	48716	24.45
1000000	362478	254175	317875	348025	268.43	296538	399797	424278	275.78
10000000	2543534	1355607	2881911	2862417	2930.35	2372313	2547993	2841561	3012.7

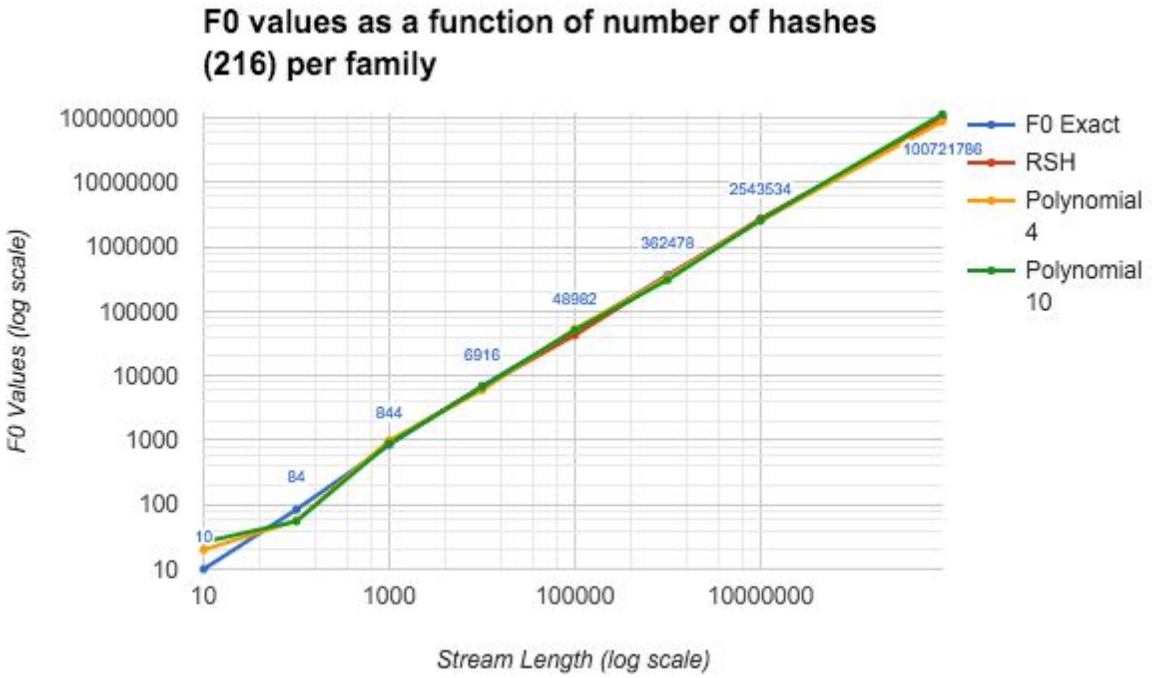
Table 3: Comparing Murmur32 and Murmur64

N(stream size)	F0 Exact	F0 Approximation							
		Polynomial Hash Function Family				RSHash Function Family			
		mean of medians from 4 groups.	median of means from 4 groups.	mean of 65 such seeds	Time(s)	One group mean/median	6 group median of means.	6 groups and mean of medians	Time(s)
10	10	25	24	24	0.9	16/10	14.8	15	1.49
100	84	56	69	71	1.0	118/82	108	82	1.37
1000	844	992	1488	1335	1.5	1155/661	1088	936	1.17
10000	6916	7280	8764	8573	1.6	9286/5295	8163	7060	1.61
100000	48982	58248	79118	80488	7.2	75438/42362	74134	52952	6.47
1000000	362478	338901	352635	373687	61.6	507050/338901	501291	423626	52.95
10000000	2543534	2541764	2764168	2904128	589.8	4108533/ 2711215	3773107	2485280	522
100000000	15525300	17622897	22691474	22523940	5539.3	24150669/ 21689720	22480491	19882243	4969.0

Table 4: Comparing Polynomial Hash and RSHash

Increasing the number of hash functions to 216

Since we notice that the standard deviation of one sample from the expectation value were quite high we increase the number of the hash functions to 216 and produced final results with that. Due to sufficient calculation time we also produced the results with Polynomial Hash function to compare whether our initial thinking of taking the RSHash would hold in the case of full data. The results showed that at least compared to polynomial hash RSHash seem to be consistently better, thus we assume that this is the case compared to murmur as well.



Corresponding Data

N(stream size)	F0 Exact	216 hashes per family					
		(mean of median from 4 buckets)				(mean of median from 10 buckets)	
N(stream size)	F0 Exact	RSH	STD of 216 values	Polynomial 4	STD of 216 values	Polynomial 10	RSH 10 buckets
10	10	15	11	20	15	26.3	
100	84	82	95	56.3	52	55.3	
1000	844	909	1114	992	1308	892	925
10000	6916	6618	9065	5956	7170	6618	7413
100000	48982	42362	56202	52952	66151	50834	48716
1000000	362478	338901	383610	338901	324940	305010	372791
10000000	2543534	2711215	3487633	2541764	2786752	2575654	2982336
911740903	100721786	97603743	150690224	86758883	138796424	112786548	99772715

Table 5: Table comparing the F0 values for 216 hashes per family (RSH and Polynomial) as a Function of stream length. It can be observed that standard deviation of the 216 results is still quite high. Buckets here refer to groups of the 216 data points created by first randomly permuting the points and then grouping them into groups of different sizes (4 or 10)

For the complete data set we also tried to Permute the 216 results 100 times and take mean of median from 10 buckets, each time. Of these 100 resultant values, we found the mean of median from 10 buckets.Performing this "averaging" results in much lower standard deviation of the result.

N(stream size)	F0 Exact	RSH	STD of 100 permutations	Polynomial	STD of 100 permutations
Full (911740903)	100721786	108882398	6806426	113654137	6871089

Table 7 : Comparison of F0 approximations of the complete dataset using RSH and Polynomial hash using the averaging described above. It can be observed that the standard deviation of the approximation values is reduced significantly when compared to the corresponding values in Table 6.

Alon-Matias-Szegedy algorithm for surprise index F2

```
---------------------------------------------------------------
AMS ALGORITHM (as implemented by us)
---------------------------------------------------------------
   ❑ X_LIST_IDX = map()
   ❑ X_XCOUNT = list()
   ❑ N := -1
   ❑ S1 := 4,100,000
   ❑ for all line in Stream do
        ❑ N := N + 1
        ❑ if N < S1
             ❑ X_XCOUNT.append([line, 1])
             ❑ X_LIST_IDX[line].append(N)
        ❑ else
             ❑ PROB := S1/N
             ❑ if rand() < PROB
                  ❑ ENTRY,IDX := pick_uniform_rand(X_XCOUNT)
                  ❑ LINE_STR := ENTRY[0]
                  ❑ X_LIST_IDX[LINE_STR].remove_from_list(IDX)
                  ❑ X_XCOUNT[ENTRY] := [line,1]
                  ❑ X_LIST_IDX[line].append(IDX)
             ❑ else
                  ❑ IDXs := X_LIST_IDX[lne]
                  ❑ for all IDX in IDXs do
                       ❑ X_XCOUNT[IDX][1] := X_XCOUNT[IDX][1] + 1
                       ❑ end
   ❑ X_COUNTS = []
   ❑ for COUNT in X_XCOUNT do
        ❑ X_COUNTS.append(N*(2*COUNT[1]-1))
        ❑ end
   ❑ Return mean(X_COUNTS)
```

The paper [Alon, Matias, Szegedy] provided two algorithms to approximate F2. They are both based on idea to calculate moments of random items in the sequence and use algorithm specific function to generate new random variable. Collect S1, such random variables take the ("weighted") mean to form new random variable Y and from S2 Y select the median value for the approximation of the F2. The algorithm we chose to use collects S1 samples of value X=n*(r^2-(r-1)^2)=n*(2r-1), where n is the sample size and r is the count of this sample.

The paper presented proof for the space requirement of these two algorithms to get any lambda confidence interval of F2

approximation as a function of the epsilon confidence probability. The space requirement is function of unknown "universal" element space.

We selected the algorithm which were more space hungry according the proof. We did the selection since the algorithm were more simpler and we were interested to see whether this could produce accurate results with much smaller sample sizes that were required in the proof.

Number of samples S2 and S1

Theoretical requirements of sample sizes and space requirements are function of unknown element space cardinality n. To get the some estimate of sample size requirements we need to estimate that as well as select the lambda confidence interval and the confidence level epsilon. Since we know exact value of the F0 is on order 100M we can use that as our estimate of n (not to overestimate too much). Then if we select lambda = 0.15 and epsilon to be 0.05 when our estimate would be in interval [1.15F2 0.85F2] with 95% probability. Using the equations in the paper S1 is around 7M and S2 around 9. With the other (main)algorithm presented in the paper S1 would be in the order of 1000. The total number of needed samples is S1*S2. The needed sample size of first algorithm is clear overestimate it can be easily dropped to less than half which is done e.g. some lecture notes in internet [Amit Chakrabarti]. However we assumed that the dropping could be much bigger just by comparing to the other algorithm requirement which required less than 10000 moment samples and then did random/hashed weighted calculations, collect averages and take median of those.

We decided to make three parallel run and collect 10000, 1000 and 200 samples, and estimate the error from that sample size. We group the samples to the groups of size 1000, 100, 20 samples take the average using the mean median trick with random would have been enough even we would not have the theoretical background for that.

Approach	F2 value (1)	error of (1)	F2 value (2)	error of (2)	actual time	time complexity	space complexity (Bytes)
Exact value - singlepass	1.95E+15	NA	1.95E+15		same	630s	1611548576
Exact value multipass	1.95E+15	NA	1.95E+15		same	NA	1600000
Approximation (200 samples)	1.34E+15	3.13E-01	1.05E+15	4.62E-01	same	0.5h	412000
Approximation (1000 samples)	1.92E+15	1.76E-02	1.85E+15	5.05E-02	same	3210.1s	544000

Approximation (10 000 samples)	1.84E+15	5.88E-02	1.83E+15	6.35E-02	same	4142.4s	2000000

NOTE:The F2 value (1) is the mean of samples, whereas F2 value(2) is based on group averages and then median with 100 randomised groups. For some reason the approximation with Python implementation required 1.4Gb of memory (for all sample sizes), whereas the C++ implementation was more understandable level of 400 kB when only 200 samples were collected.

In this particular case the 1000-10000 values seem to give us good enough approximations with less than 10 percent error rate. Even the direct mean get a better results to us we used the median of the means with grouping to 10 groups and randomised selection of the group member over 100 different shots. This did not give as good results but the variation of this should be smaller, thus when we decided to use limited number of samples we thought that this approach would limit the random variation between different samples.

Summary of results

The main results of our selected choice are collected to this section. For F0 approximation we selected the RSHash with 216 different seeds. We postprocess the results by grouping the results to the group size of approximately 10 samples taking the median of those samples and then mean of the 20 groups. To get rid of the effect of group selection we calculated the average over hundred random group selection.

Approach	F0 value	error	actual time	time complexity	space complexity (Bytes)
Exact value - singlepass	100721786	NA	NA	612s	1611548576
Exact value multipass	100721786	NA	NA	NA	1600000
Approximation (RSHash 216)	108882398	0.08	2.5h	41.5h	4000

NOTE : The space used in multipass is implementation dependent and the value reported is the one we chose (arbitrarily).

For F2 results we used a lot of smaller data sample sizes than proposed in the original algorithm proof and find out that the estimates were good enough. We used the smaller sample size since bigger sample sizes would have increased the space and time complexity too much. To avoid unnecessary variation of small sample size we used 100 random grouping and averaged over them.

Approach	F2 value	error	actual time	time complexity	space complexity (Bytes)
Exact value - singlepass	1.95E+15	NA	same	630s	1611548576

Exact value multipass	1.95E+15	NA	same	NA	1600000
Approximation (10 000 samples)	1.85E+15	0.0505	same	4142.4s	2000000
Approximation (1000 samples)	1.83E+15	0.0635	same	3210.1s	544000

NOTE: If the values would be equally distributed i.e. about 9.05 values of each 100721786 distinct values the surprise index would have been in the order of 8.25E+9.

Conclusion and Discussion

For calculating the exact value when possible and timewise doable the multipass algorithm is much more space efficient than the single pass algorithm. In the single pass algorithm avoiding sorting and using the unordered maps improves the space and time complexity significantly.
The F0 approximation algorithm was very time consuming compared to the calculation of the exact value. Thus separate hash functions were run in parallel to keep the computation time reasonable. We concluded that RSHash is our suggestion for the hash function family, due to superiority among the tested hash function families. We selected the 64 bitstring for final calculations but in theory 32 bitstring should have been enough. As a further improvement we would suggest to check whether the theory that 32 bit RSHash would give as good results is solid.
The F2 approximation algorithm provided reasonable good approximations with reasonable small sample size. We find out that 1000-10000 samples would be enough. The space and time requirements of such a size approximation algorithms can be kept reasonable. The space requirement is smaller or same order than with the multipass algorithm however time requirements were much higher. For further study it would have been interest to see whether this smaller sample size is reasonable assumption in general and whether the accuracy would improve if we would increase the sample size from 10000 to 100000.

References

N. Alon, Y. Matias, and M. Szegedy. “The space complexity of approximating the frequency moments.” In Proceedings of the twenty-eighth annual ACM symposium on theory of computing (STOC), pages 20–29, 1996.

P. Flajolet and G. N. Martin. “Probabilistic counting algorithms for database applications.” Journal of computer and system sciences, 31(2):182–209, 1985.

J. Leskovec, A. Rajaraman, and J. Ullman. “Mining of massive datasets.” Cambridge University Press, 2014.

Amit Chakrabarti Data Stream algorithm lecture notes.

Appendix

1. Derivation of Space complexity of RSHash
2. Derivation of Space complexity of Polynomial Hash

Derivation of Space Complexity of RSHash

RS Hash Space Complexity.

Let us assume a string n bytes long. = 8×n bits.

RSHash takes in 2 prime numbers as argument let them be m & p bits long.

```
unsigned long long.
RSHash (str, a, b) {. // a = b = unsigned long long
        unsigned long long hash = 0;
        for (int i=0 ; i < str.length() ; i++){
                hash = hash * a + str[i];
                a = a * b;
        }.
        return hash; // same as hash % 64 ∵ ~~return~~ hash is an unsigned long.
}.                   // long.
```

n	hash
0	0

1. hash ⇒ integer representation of char. ⇒ max 8 bits.

2. iter1: $hash = hash * a + z^8$

   $hash = 0 \times 2^m + z^8$ // if ~~m~~ a is m bits long. max # representable // is $2^{m+1} - 1$ writing it as 2^m to get // rid of constants. same with 2^8.

   $a = 2^{m+p}$.

   iter 2: $hash = hash * a + z^8$

   $= (2^8)\, 2^{m+p} + 2^8$.

   $\boxed{hash. = 2^8(2^{m+p} + 1)}$

   $a = 2^{m+p} \cdot 2^p = 2^{m+2p}$.

3 Characters. $hash = hash * a + z^8$.

$= 2^8(2^{m+p} + 1) \cdot 2^{m+2p} + 2^8$

$\boxed{hash. = 2^8(2^{2m+3p} + 2^{m+2p} + 1)}$

$a = 2^{m+3p}$.

Similarly

4 Characters $hash = 2^8(2^{3m+6p} + 2^{2m+5p} + 2^{m+3p} + 1)$.

5 Characters $hash = 2^8(2^{4m+10p} + 2^{3m+9p} + 2^{2m+7p} + 2^{m+4p} + 1)$

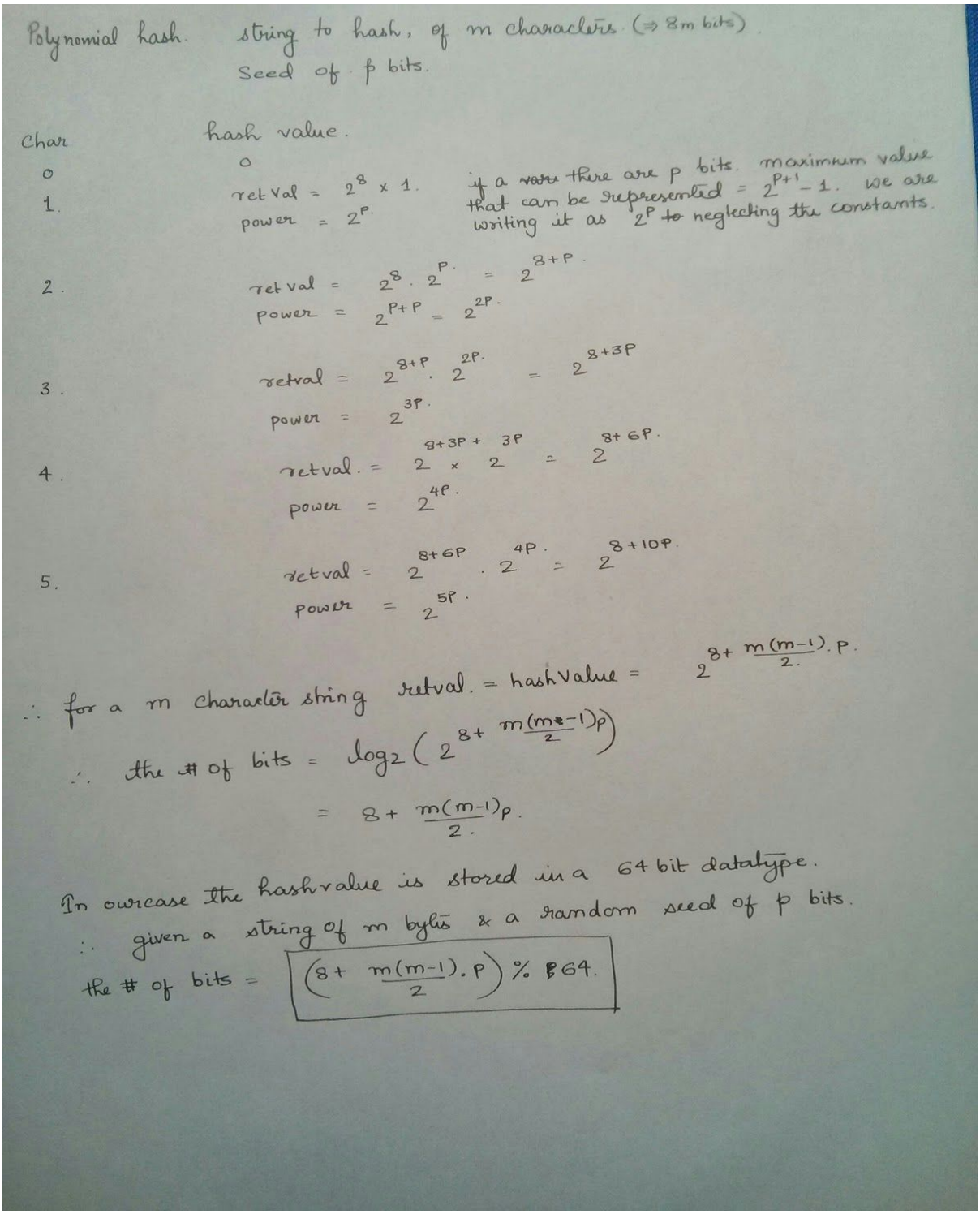
6 Characters $hash = 2^8(2^{5m+15p} + 2^{4m+14p} + 2^{3m+12p} + 2^{2m+9p} + 2^{2m+5p} + 1)$.

So given an N character string the hash value ⇒ $2^8\left[\sum_{i=0}^{N-1} 2^{(N-1-i)m + \left(\frac{N(N-1)}{2} - \frac{i(i+1)}{2}\right)p}\right]$

however, the number of bits is (for a string of length N with primes represented in m & p bits).

$$\log_2\left[2^8\left[\sum_{i=0}^{N-1} 2^{(N-1-i)m + \left[\frac{N(N-1)}{2} - \frac{i(i+1)}{2}\right]p}\right]\right] \% 64.$$

Derivation of Space complexity of Polynomial hash



Mod 64 is what we have implicitly used because unsigned long long in c++ is 64 bits long (at least). However, it can be changed to get hashes of desired bit lengths.