Fatih Taşyaran fatihtasyaran@sabanciuniv.edu
Computer Science and Engineering,2014
Kerem Yıldırır keremyildirir@sabanciuniv.edu
Computer Science and Engineering,2015
Kamer Kaya
Computer Science and Engineering

1 Abstract

One of the biggest challenges in today's business and technology world is dealing with enormous chunks of data. One of the problems people encounter while working with big data is that it takes too much time to analyze it. Standard data structures like linked lists and hash tables may be incompetent or inefficient for some specific problems such as keeping track of the elements' frequency, or checking if an element is in the set or not. Hence, people are trying to come up with clever solutions such as data sketches. We implemented various sketches, and analyzed their performance for possible improvements.

2 Introduction

Data sketches are derived from Bloom Filters, which are arrays that consist of 1 and 0s (Cormode, 2017). Bloom Filters are used in search and validation purposes. Bloom Filters(BF) can answer a query with 2 possible outputs:

- The element is definitely not in the data
- The element is probably in the data

In ideal case, for ith element in the data, $BF[i] = 1$. On the contrary, in real data, this is rarely the case. Even on hash table applications, there exists an item i such that $h_i = h_j$. Still, BM's are compact, nice structures. (Cormode, 2017)

Sketches are data structures which are used for following the frequencies of elements in a large data set. Main distinguishing feature of a sketch is that its size does not depend on size of the data set. Size of the sketch depends on two error parameters: δ and ϵ, where ϵ is the minimum error rate user can afford and δ is the error frequency. Number of rows and columns in the sketch is computed with the following formulas:

$$nrows = \log_2(\frac{1}{\delta})$$

$$ncols = \frac{2}{\epsilon}$$

A step by step explanation for sketch mechanism is as follows:

- Define sketch size with δ and ϵ
- Insert elements into the sketch table using predefined number of hash functions and perform the operation on table in order to keep the frequency. (Different sketches use different operations.(Cormode, 2017))

There are various hash functions for this purpose, but in this paper, our choice was Tabular Hashing. The reason behind this choice is ,Tabular Hashing is simple, robust and easily implemented. We managed to improve total performance by merging the hash functions' tables, which led to a significant improvement performance-wise.

3 Implemented Data Sketches

3.1 Count-Min Sketch

In the application of this sketch, the element to be inserted is hashed n times, where n represents the number of rows. Each element is hashed by hash functions $h_1, h_2 \dots h_n$. Each hash function increments $sketch[i][h_i]$ where i corresponds to hash function's return value and $sketch$ refers to the sketch table. For queries, the minimum corresponding value of all rows is returned.(Cormode, 2017)

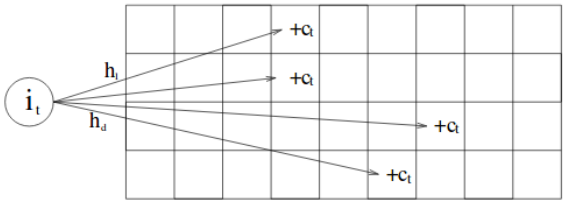


Figure 1: Illustration of Count-Min Sketch

3.2 Count-Mean Min Sketch

Initialization of Count-Mean Min Sketch(CMM) is exactly same as Count-Min Sketch(CM). It can be said that since every index is incremented for multiple elements, CM's results are *overestimating* the true values. Hence it is a biased estimator. CMM intends to solve this problem by subtracting a noise from the queried value. Noise is estimated as : $\frac{NoElements - sketch[i][h_i]}{ncols-1}$ for every row. After calculating the noise, it is deducted from each $sketch[i][h_i]$ value computed in each row. Finally, the minimum of these values is returned as query output.

$$output = \min(sketch[i][h_i] - noise_i) \quad i = 1 \dots nrows$$

where i is the row number and h_i is the index obtained from hash function. (Deng,Rafiei, n.d)

3.3 Count Sketch

In Count Sketch, 2 hash functions are maintained for each row. h_k for determining the column index value, and g_k for determining an increment value in the interval $[-1, 1]$.is done by acquiring the median of all $sketch[i][h_i]$.

$$sketch[k][h_k] = sketch[k][h_k] + g_k \quad k = 1 \dots nrow$$

It aims to neutralize the bias that comes with overestimating. (Cormode,Hadjieleftheriou, 2008)

3.4 Count-Min Sketch with Conservative Update

This is an update for Count-Min Sketch and aims to solve the unnecessary incrementation in the sketch. Since queries return the minimum of $sketch[i][h_i]$, during the incrementation, it only increments the minimum value of all $sketch[i][h_i]$. In other words, only $\min(sketch[i][h_i])$ is incremented. This heuristic reduces the bias caused by overestimating.(Goyal, 2012)

4 Tabular Hashing

In data sketches, elements that inserted to the table must be hashed multiple times. In this implementation of sketching, tabular hashing is used. Which first introduced in an article by Zobrist to avoid recalculations of board configurations at the current state in board games.(Thorup, 2012) It is a strong hash function which grants significant level of randomness. Hashing process requires a table filled with random integers. After initializing the table, hash function can be used. Code for hashing can be examined in the following figure.

```
uint32_t hash(const uint32_t data) const {
  uint32_t i, c, x = data;
  uint32_t h = 0;
  for (i = 0; i < 4; i++) {
    c = x & mask;
    h ^= table[i][c];
    x = x >> 8;
  }
  return h;
}
```

Figure 2: Hash function written in C++

With using the fastest known hashing scheme multiplication-shift hashing. Hash function takes an 32-bit integer x as input. It divides x into 4 8-bit chunks and iterates them in order. At each iteration, logical AND operation is done between the element and an 8-bit mask which consists of zeros. With this, it is guaranteed that first 8-bits of the 32-bit integer becomes zero. This also changes the value of element x, to another integer c. Then, logical XOR operation is applied between changed element and the integer in hash table which is on the ith row where $0 < i < 4$ and cth column and assigned to a variable h. At the end of iteration, x is shifted right 8-bits. When all iterations are done a unique h value has been achieved.

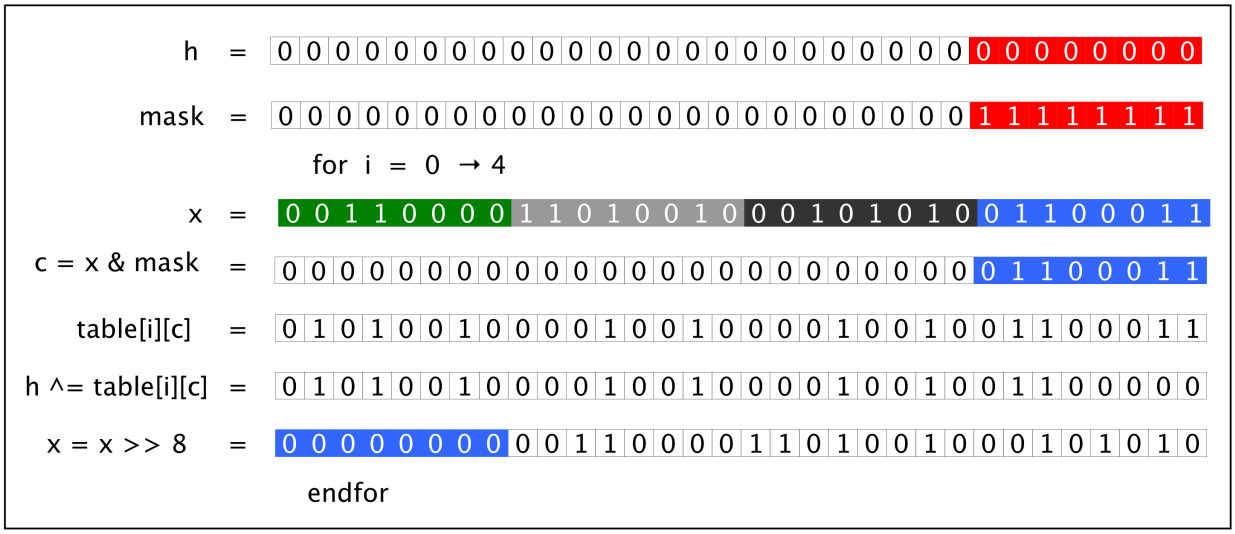


Figure 3: Tabular Hash illustrated

At the end of iteration, is shifted right 8-bits. When all iterations are done a unique h value has been achieved.

5 Parallelization

The sketch can be seen as a sum of arbitrary sketches using the same h_j for their rows j. In this manner every sketch is combinable while they use same ϵ, δ and hash function h_j for their rows j. Say in sketch A, $A[j,k]$ is the sum of updates to the location (j,k) for index values i such that $H_j(i) = k$ in the process and $a_{j,k}$ is a binary vector such that $a_{j,k} = 1$ if and only if $h_j(i) = k$. Then we can say

$$A[j,k] = \sum_{i=0}^{n} a_n[j,k]$$

(Cormode, 2017) This led us an important consequence that sketches have high potential of parallelization. Since the data is shared between threads that all have their own sketch then combined in order to form a final table by OpenMP's fork-joint parallelism, we can distribute the input set to the threads, as many as available.

The drawback here is, while the threads are increasing and subsets of the input set that computed parallel are increasing, memory required to contain their tables are also increasing. This called time-memory trading. In our project, we focus on multi-table computing of sketches, we use more memory but get improvements on performance.

5.1 Synchronization Problems With Parallelization

5.1.1 Race Condition

In multi-thread programs, threads are communicate by sharing variables. In cache, threads are interleaving. Basically saying, there is no defined schedule for which thread is processing at the time or in which sequence they will be processed. Thus when the scheduling of threads affect the output, this ambiguity may result in different outputs of the same program at different runs. This is called a race condition.

5.1.2 False Sharing

In symmetric multiprocessor systems, each processor has a local cache. And the memory system must guarantee a consistency between the data in this caches, called cache coherency protocol. When threads on different processors modify variables that sits on the same cache line, a false sharing occurs. This invalidates the cache line and forces an update on cache line. Thus the false sharing causes cache lines to go back and forth between threads and decreases performance.

5.1.3 Synchronization Tools

Synchronization is bringing one or more threads to a well defined and known point in their execution. Common synchronization tools are:

- **Barrier:** Each thread wait at the barrier until all threads arrive.
- **Mutual Exclusion:** Defining a block of code that only one thread at a time can execute.
- **Critical:** Defines a mutual exclusion for a block of code.
- **Atomic:** Uses hardware support to do a quick update on a value. If there is no hardware support, act like critical. Only use a scalar l-value and a non overloaded built-in operator.

- **Lock:** Lowest level mutual exclusion in synchronization. Lock is actually a variable. The thread that holds the lock can execute processes on the lock and the other threads doing something else while waiting for lock is available.

6 Parallelization With Synchronized Hashing

As mentioned before, we focus on multi-table sketches in our project and gain performance while using more memory. Moreover we develop a new way of tabular hashing in order to exploit the speed of fastest L1 Cache. In our model of tabular hash, instead of using different randomly-filled tables for each row of the sketch as a look-up table for corresponding row's hash function; we merged these tables by arrange the columns of tables of rows consecutively.

```
void hash(const uint32_t data, uint32_t results[8]) const {

   uint32_t i, j, c, x = data;
   uint32_t h = 0, temp;

   c = 8 * (x & mask);
   results[0] = table[0][c];
   results[1] = table[0][c + 1];
   results[2] = table[0][c + 2];
   results[3] = table[0][c + 3];
   results[4] = table[0][c + 4];
   results[5] = table[0][c + 5];
   results[6] = table[0][c + 6];
   results[7] = table[0][c + 7];
   x = x >> 8;

   for (i = 1; i < 4; i++) {
     c = 8 * (x & mask);

     results[0] ^= table[i][c];
     results[1] ^= table[i][c + 1];
     results[2] ^= table[i][c + 2];
     results[3] ^= table[i][c + 3];
     results[4] ^= table[i][c + 4];
     results[5] ^= table[i][c + 5];
     results[6] ^= table[i][c + 6];
     results[7] ^= table[i][c + 7];

     x = x >> 8;

   }

 }
```

Figure 4: Code for merged tabular hashing

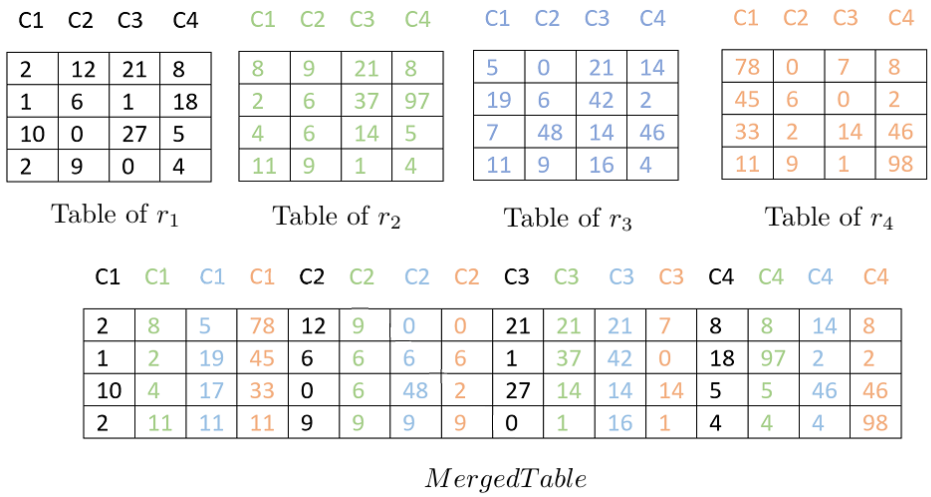


Figure 5: Illustration of Merging Tables of Rows

In the figure 4, implementation of merged hasher can be seen. Since row size has been fixed to 8, results of hash functions are put side by side in the hash table. The advantage of our method is, instead of different threads to read data that sits on different cache lines, all threads read data from the same cache line. That will be on the fastest L1 cache during process. Expecting result is reducing in cache-misses thus increase in performance. This result could be seen in experiments section.

7 Batching and Padding

Furthermore, we determined which part of the stream will be processed by which thread. Hence, instead of processing the data as a whole, we divided the stream into batches of 1024 elements. This way, we were able to assign the data to the threads in a way such that each thread writes to it's predetermined location on the sketch without overlapping, eliminating race conditions. As expected, our throughput increased significantly especially when the sketch size is large(8 by 2003).

Since we desired to see the same improvement also in a smaller sketch(8 by 211), we implemented a padding over the buffers which keep the results from hashing. To keep eight 32-bit integers, we used an array of 256 32-bit integers. With this way, we were able to provide a better cache line integrity, resulting in fewer number of threads processing on the same cache line.

8 Experiments

We ran our experiments, on a Raspberry Pi 3 Model B which has the following specs:

SoC: Broadcom BCM2837
CPU: 4x ARM Cortex-A53, 1.2GHz
GPU: Broadcom VideoCore IV

RAM: 1GB LPDDR2 (900 MHz)
Networking: 10/100 Ethernet, 2.4GHz 802.11n wireless
Bluetooth: Bluetooth 4.1 Classic, Bluetooth Low Energy
Storage: 16 GB Class10 microSD
GPIO: 40-pin header, populated
Ports: HDMI, 3.5mm analogue audio-video jack, 4 x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

To calculate the speed-up we used the following formula:

$$T_s = \text{Non-parallel execution time}$$
$$T_n = \text{Execution time using } n \text{ threads}$$
$$SpeedUp = \frac{T_s}{T_n}$$

We also fixed our sketches size to 8 by 211 in order to facilitate coding and track results more precisely while developing the new method.

Results and Conclusion

To assess our improvement's success, we ran experiments with every consecutive version we optimized in order to provide a better cache utilization.

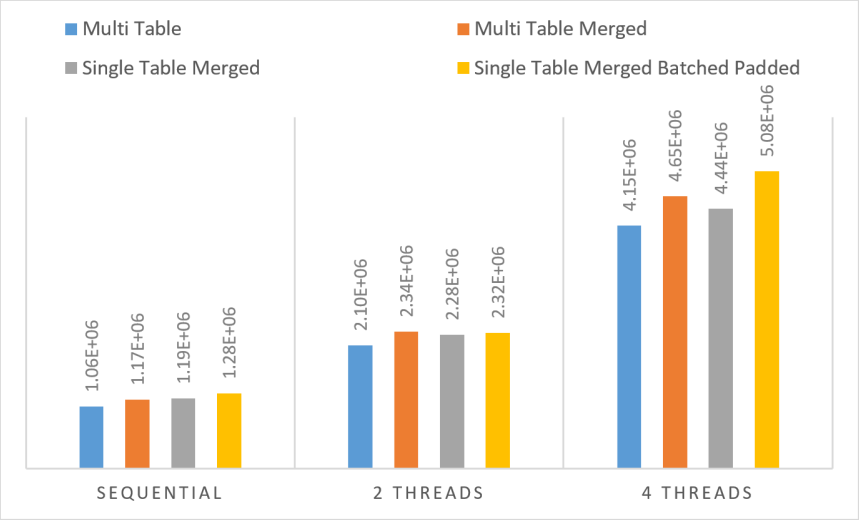
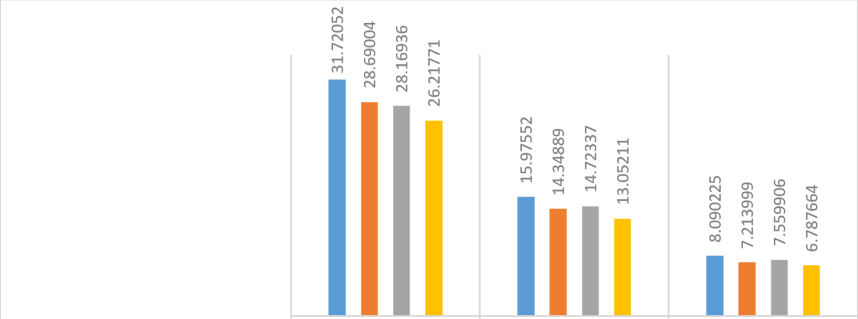


Figure 6: Throughputs of Different Versions Averaged for 10 Experiments

Multi table is what we've started with, then we merged the hash tables to achieve a better spatial locality. Next step was to made threads work on a single sketch. Low performance on single sketch is due to false sharing caused by threads. To overcome this we batched the stream and padded the medium result arrays.



	Sequential	2 Threads	4 Threads
Multi Table	31.72052	15.97552	8.090225
Multi Table Merged	28.69004	14.34889	7.213999
Single Table Merged	28.16936	14.72337	7.559906
Single Table Merged Batched Padded	26.21771	13.05211	6.787664

Figure 7: Completion Times of Different Versions Averaged for 10 Experiments

Finally, reduced cache miss rates based on the well determined workspaces of threads on the sketch and stream led to a visible performance increase on task.

Future Work and Further Improvement

For future improvement, our aim is to increase the performance even more, we plan to run our application on other single board computers with multiple cores such as Odroid and Parallela. For the purpose of reproducible researching, source code of the project will be posted on BitBucket and GitHub .

9 References

- Goyal, A., Daume, H., Cormode, G. (2012, July). Sketch Algorithms for Estimating Point Queries in NLP. Retrieved from http://www.umiacs.umd.edu/ amit/Papers/goyalPointQueryEMNLP12.pdf
- Pătraşcu, M., Thorup, M. (2012). The Power of Simple Tabulation Hashing. Journal of the ACM, 59(3), 1-50. doi:10.1145/2220357.2220361
- Thorup, M. (2017). Fast and powerful hashing using tabulation. Communications of the ACM, 60(7), 94-101. doi:10.1145/3068772
- Cormode, G. (2017). Data sketching. Communications of the ACM, 60(9), 48-55. doi:10.1145/3080008
- Cormode, G., & Hadjieleftheriou, M. (2008, August). Finding Frequent Items in Data Streams. AT&T Labs–Research. Retrieved from http://www.vldb.org/pvldb/1/1454225.pdf
- Deng, F., & Rafiei, D. (n.d.). New Estimation Algorithms for Str eaming Data: Count-min Can Do More. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.420.449&rep=rep1&type=pd