High Throughput Heavy Hitter Aggregation for Modern SIMD Processors

Orestis Polychroniou
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

ABSTRACT

Heavy hitters are data items that occur at high frequency in a data set. They are among the most important items for an organization to summarize and understand during analytical processing. In data sets with sufficient skew, the number of heavy hitters can be relatively small. We take advantage of this small footprint to compute aggregate functions for the heavy hitters in fast cache memory in a single pass.

We design cache-resident, shared-nothing structures that hold only the most frequent elements. Our algorithm works in three phases. It first samples and picks heavy hitter candidates. It then builds a hash table and computes the exact aggregates of these elements. Finally, a validation step identifies the true heavy hitters from among the candidates.

We identify trade-offs between the hash table configuration and performance. Configurations consist of the probing algorithm and the table capacity that determines how many candidates can be aggregated. The probing algorithm can be perfect hashing, cuckoo hashing and bucketized hashing to explore trade-offs between size and speed.

We optimize performance by the use of SIMD instructions, utilized in novel ways beyond single vectorized operations, to minimize cache accesses and the instruction footprint.

1. INTRODUCTION

Databases allow users to process vast amounts of data. Nevertheless, due to the limitations of human perception, the conclusions we draw from this volume of information are often summarized in a few words or charts. One way to narrow down the volume of information presented is to focus on the most important items among those being analyzed.

One measure of importance is the total contribution an item makes to the whole. Items that contribute the most are called *heavy hitters*. Heavy hitters can be defined in absolute terms (e.g., items occuring more than 1% of the time) or in relative terms (e.g., the top 100 items). In the scope of this paper we use the top-K definition, but our approach can easily be modified to account for other definitions. In many real-world datasets, skew in the data means that aggregate data about a small number of heavy hitters convey a lot of information. Our goal is to identify the heavy hitters and calculate exact aggregates (count, sum, etc.) for those items.

Now that systems with very large main memories are available, the performance bottleneck has shifted from I/O to CPU and memory [11]. Modern commodity processors are multi-core systems. Parallelism and the ability to scale to many execution units have become primary performance considerations. Many database algorithms have been redesigned in the context of in-memory multicore platforms. With such issues in mind, we focus on parallel computation of heavy hitters from a memory-resident dataset.

Recent work on in-memory aggregation has shown that sharing a common aggregation data structure among many cores is a bad idea when there are heavy hitters [4]. Contention for popular data items causes significant delays, serializing execution and preventing the full utilization of the parallel hardware. A solution to this problem is to keep a private running aggregate for each heavy hitter on each core, to avoid coordination overheads. The final totals can be combined at the end of the pass.

When the number of grouping keys for an aggregate computation is limited, aggregation can be very fast. Under such conditions, Ye et al. was able to aggregate over one billion records per second on a commodity machine [19]. However, when the grouping cardinality increased beyond the CPU L1 cache capacity, performance dropped by an order of magnitude, even for distributions with heavy hitters that are likely to remain cache-resident. The latency of accesses to memory for non-heavy hitters dominated the performance.

In this work, instead of computing the aggregates for the whole table, we will only compute the aggregates of a few heavy hitter elements. By ignoring the non-heavy hitters, the entire aggregation is done in-cache, and the throughput is an order of magnitude higher. Further, by using branch-free SIMD implementations of various aggregation data structures, we are able to get additional speed improvements, significantly beyond the performance of Ye et al. [19] even for cache-resident aggregates. We utilize the same SIMD registers to hold multiple items (e.g.: counts & sums) and minimize the instruction footprint.

To identify the heavy hitters, we use a sampling step prior to aggregating the full data. In a billion-element data set, the cost of sampling even a million elements in advance is small relative to the cost of scanning the base data. The

*This work was supported by NSF grants IIS-0915956 and IIS-1049898.

DaMoN'13, June 24 2013, New York, NY, USA

basic idea is to count both matches and nonmatches for keys in the aggregation table. The biggest nonmatch count gives us an upper bound on the contribution of an item whose key was not explicitly inserted into the table. We can tune the number of keys and nonmatch counts to explore trade-offs between the number of heavy hitters explicitly counted, the bound on heavy hitters that might have been missed, and the overall performance of the algorithm.

Another trade-off that we explore is the design of the hash table used to perform the aggregation. A simple hash table without overflows or chaining is very efficient. However, the birthday paradox ensures that only a limited number of entries can be inserted before a collision is encountered. This collision bound can be extended by iterating through many random hash functions (using a fixed time budget for this process) and choosing the function with the highest occupancy before a collision. The bound can also be extended by having a smaller number of hash buckets that can each hold more than one element. Alternatively, schemes based on cuckoo hashing [16] offer higher occupancy guarantees, at the cost of using multiple hash functions and looking up more than one hash cell per key. We show how one might choose between the various alternatives.

Whichever method is used, our heavy hitter capacity will be limited by the size of the L2 cache memory. Even in a typical L1 cache with size of 32KB, we are able to store a few thousand keys along with counts and other data. A few thousand keys may be a small fraction of the keys in a dataset. Nevertheless, for data with Zipfian skew, the top thousand heavy hitters capture a large (and presumably interesting) fraction of the data. In the event that the user needs even more heavy hitters, we can fall back on standard aggregation methods. If the top thousand or so items satisfy the user most of the time, then it is a net win to use our specialized heavy-hitter methods because they are so much faster than standard aggregation.

In some cases, such as for uniformly distributed data, we may not identify any heavy hitters. Even in such cases, the nonmatch counts will allow us to bound the maximum frequency possible for all items. This behavior should not be considered a failure of the algorithm. The absence of heavy hitters beyond a certain threshold may be all that the user needs, such as when the task involves looking for outliers. In such cases, a fast heavy-hitter algorithm is better than a slower complete aggregate computation for all keys.

Much prior work on heavy hitters (discussed in Section 1.1) has focused on streaming applications, where memory is limited and one typically uses just one pass through the data. We emphasize that our target application is not streaming, but rather data analytics and decision support.

We present a novel approach that computes aggregates for the heavy hitter elements of a dataset. We design cache resident structures, copied between cores, based on perfect hashing and a novel hash probing method using SIMD. We show that this combination allows the algorithm to run at extremely high throughput rates. Using sampling and a number of different configurations for the hash table, we can exploit all tradeoffs between performance and the cardinality of the output (and quality of the extracted summary).

In the remainder of this section we present related work. In Section 2 we describe our approach in more detail, illuminating some key implementation details. In Section 3 we show our experimental evaluation and conclude in Section 4.

1.1 Related Work

Heavy hitters have been extensively studied in data stream analysis. For some data stream scenarios, such as those motivated by network traffic analysis within a router, memory is limited and data is available only within a narrow time window. Under such conditions, many algorithms approximate heavy hitter counts because there is not sufficient space to maintain complete count information. As previously mentioned, our work does not assume limited memory or a single pass through the data. We also aim to compute exact counts and other aggregates, rather than an approximation, for items that are heavy hitters.

Counter based algorithms for heavy hitter identification in streams include Frequent [10, 14], Lossy Counting [12], and Space Saving [13]. Challenges include determining which elements to count, and how to approximate the counts particularly when new elements become frequent in the stream. Sketch based algorithms include Count Sketches [2] and Count-Min Sketches [5]. Such algorithms compute summaries of the distribution that allow the approximate inference of heavy hitters and other queries. See [6] for an extensive analysis and experimental evaluation of these methods.

Aggregation on modern multi-core CPUs has been studied in [3, 4, 19]. A small local table stores frequent keys to avoid contention between threads in shared data structures. While these methods work well for a small number of keys that stay cache resident, the throughput deteriorates rapidly in the presence of more distinct keys, even for heavy hitter distributions. By focusing on heavy hitter aggregates alone, our method runs more than an order of magnitude faster and retains the same performance for more distinct keys.

Single Instruction Multiple Data (SIMD) instructions have been used to speed up database algorithms [20, 21], including hash probing in bucketized cuckoo hash tables [17]. SIMD execution has a secondary benefit of being able to avoid branches for many inner-loop computations [17, 20, 21], an important benefit since branch mispredictions can be an important performance overhead.

Database algorithms sensitive to modern hardware have been studied in several contexts. MonetDB/X100 [1] is designed to operate primarily in the CPU cache. Recent work maximizes the time column values remain in registers based on compiled code for query execution [15]. Other recent work that focused on frequent item counting in streams [18], discusses filtering frequent data before applying Space Saving [13]. The approach is augmented by the use of SIMD for batch item comparisons and count increments.

Overall, most past work on SIMD uses arrays of elements of the same type, performing many instances of the same operation using one SIMD instruction. We go beyond this uniform processing, handling different kinds of work in each SIMD cell and reducing the total number of instructions.

1.2 Example Query

We show an example of a heavy hitter aggregation query.

```
select product_id, count(*) from sales
group by product_id order by count(*) desc
limit 1000;
```

Other variants, such as heavy hitters defined by a lower bound on frequency, are similar to our approach, but are not discussed in the scope of this paper. One can easily adjust our techniques to other heavy hitter definitions.

2. DESIGN & IMPLEMENTATION

2.1 Sampling

The first step of the process is sampling the data to extract heavy hitter candidates. Since the cost of sampling is known in advance, we can explicitly decide on the sample size so that it does not take more than a small fraction of the total time. In our target scenarios with billions of input records, we will be able to construct relatively large samples containing millions of elements. Such large samples will help us obtain good statistical bounds on the likelihood that we have sampled all true heavy hitters.

As described, we use sampling to identify the most likely heavy hitter candidates for the aggregation phase. We clearly need to include the top K items from the sample. We actually include more items (subject to capacity constraints) for two reasons. First, it could well be that items outside the top K in the sample are actually in the top K of the full data set, so all items with counts close to that of the Kth item in the sample should be included. Second, by counting additional items with high counts (even if not sufficiently high to be in the top K), we will be able to improve the accuracy of the validation step, described below.

2.2 Validation

In addition to aggregating a set of candidate heavy hitters, we can also simultaneously compute aggregates for non-candidates. Rather than aggregating non-candidates individually, we group them into hash buckets and compute an aggregate for each bucket. Similarly to the sketch-based techniques described in Section 1.1, the largest aggregate A among all hash buckets provides a conservative empirical bound on the heaviest hitter that is not among the candidates. We need K candidates to have aggregate above A.

The quality of the bounds derived by aggregating the non-candidates will depend on the heavy hitter distribution, as well as the number of buckets. The sample itself can provide an estimate of the fraction of the data set that is concentrated in the non-candidates, which will be useful when choosing the number of hash buckets for the non-candidates. Using more buckets gives a better accuracy bound, but may slow down computation because the hash table may need to reside in the L2 cache rather than the L1 cache.

Even with good choices for the parameters the validation step may fail. Failure may happen for one of several reasons:

- The user is too ambitious, the specified K is too big.
- The user is not especially ambitious when specifying K. Nevertheless, the distribution is such that the Kth element reaches sufficiently far into a range where there are many items with similar counts.
- There is sufficient skew in the non-candidate counts that the maximum count among the non-candidate buckets is high. We can expand the number of candidates (subject to capacity constraints) to reduce both the mean and variance of non-candidate counts.

If a user truly wants information about many items, then a complete aggregation of the dataset may be necessary. For experimental guidance about what constitutes "too many," see Section 2.2. In general, for datasets with sufficient skew, we will be able to successfully and efficiently identify from hundreds up to thousands of heavy hitters.

2.3 Hashing

The basic structure we will build upon is the regular hash table, which we will heavily optimize for throughput. As our hash function, we will use multiplicative hashing. Multiplicative hashing is a very fast hashing method, and the class of multiplicative hash functions is universal [7]. For any given key size and table size, a hash function is determined by a single randomly chosen odd multiplier of size matching the key size. Once we have identified the candidate keys to aggregate from the sample, we decide on the hash multiplier. The goal is to map those candidate keys into the table *perfectly*, i.e., without collisions. A collision-free table will allow a simpler implementation and improve performance by eliminating branching and chaining.

While a random hash function may exhibit a few collisions (due to the birthday paradox), we have sufficient time to try a fairly large number of multipliers to find one that is collision-free on the candidates. On our experimental platform, we were able to try 10^5 multipliers in 50–60ms and were able to find a perfect function for roughly 250 keys out of 2048 slots. If we need to fit more keys, we will shift the design to a bucketized hash table. A 4-wide bucketized hash table would fit roughly 820 keys under the same conditions. We can compute the probability of overflowing any bucket of an m-wide n-sized table, using an $O(nm)$ algorithm [9] and then compute the expected fill rate after a number of tries. In order to further increase the fill rate up to 99%, we will use cuckoo hashing [16]. Cuckoo hashing uses two hash functions and perfectly hashes keys by moving colliding elements to their alternative hashed position. A flat cuckoo hash table will further increase the occupancy to 60–65%. Merging the two techniques results in the bucketized cuckoo table, which allows 90–92% occupancy in the 2-wide version, and 98–99% in the 4-wide version [17].

Multiplicative hashing can perform poorly in cuckoo hashing schemes [8], although the poor behavior is less noticeable in bucketized cuckoo hashing [17]. We overcome this problem, by repeating the process a few thousand times to get as many keys as possible in the table. Insertion time, normally a weaker point of cuckoo hashing for bigger tables (since we move items around multiple times), is not an issue here.

2.4 Updating with SIMD

A straightforward hash probe inner loop implementation for computing a count and sum for each group might be (in the C programming language):

```
if (key == table[hash].key) {
    table[hash].count++;
    table[hash].sum += value; }
```

The `if` test typically leads to a conditional branch in the inner loop. To avoid conditional branches, when we probe a key we have to execute an update to the aggregate of the table whether or not there was a match.

We start with two ideas previously used for branch and SIMD optimization. First, control dependencies can be converted into data dependencies by treating the result of the comparison as a variable. Second, comparison results can be used as masks for subsequent operations [20, 21]. We use the term *nullification* to describe this and rewrite the above:

```
equal = -(key == table[hash].key ? 1 : 0);
table[hash].count -= equal;
table[hash].sum += value & equal;
```

The binary representation of -1 is a word containing all 1 bits, making it suitable for masking. When there is no match, the mask is zero and the sum and count are unchanged. The compiler generated predicated `CMOV` instructions to do the updates. The improvement is 8%–10%.

The next step is to transform this implementation into one that uses SIMD. For the discussion below we assume a 128-bit SIMD register type as in x86 SSE instruction set, but the principles used also apply to other SIMD sizes. Figure 1 shows how data flows during probe/aggregation.

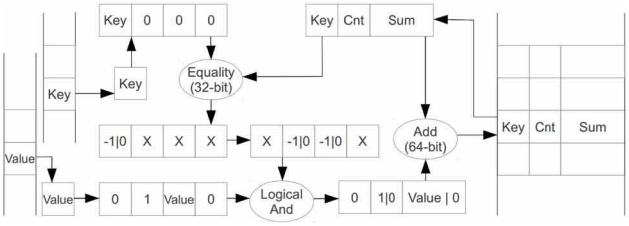

Figure 1: SIMD method for count, sum(value).

Cells labeled "X" are unimportant; we don't care what values they hold. Note that the `value` goes into the low-order (leftmost) bits of the 64-bit vector; the data representation is little-endian. The `1|0` goes into the high-order bits, so that it will increment the count but not the key. The hash table entry is loaded only once, and stored only once, unlike the scalar code. The speedup achieved using this method (which will be described in more detail in Section 3) is 76%.

SIMD techniques become even more useful for bucketized hash tables. Figure 2 shows SIMD probe implementation for buckets of size 2 for the same count and sum query.

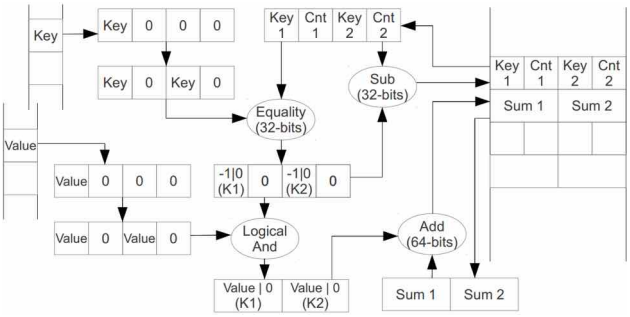

Figure 2: 2-wide table for count, sum(value).

To compute min and max values, we use specific max and min SIMD instructions that avoid branching. If the numbers are unsigned, we nullify the max update by turning the value to 0 by "and"-ing with the key comparison mask. For min update we turn the value to -1 by "or"-ing with the mask's inverse. In case we need signed numbers, the fastest way to do it is to store them in unsigned format and add or subtract the difference before displaying them. For simple sums (i.e., sum(value)), we subtract the count $\times 2^{31}$, offline at the end instead of doing conversions with each update.

When the aggregation operations required from the query are more complicated, the payloads are longer and flat tables become significantly faster than a wider bucketized table. They have less data to update and need fewer loads and stores. We use query 3 presented in Section 3, as our all-in-one case. In that query, the size of the candidate entry is 32 bytes and the bucket of the 4-wide hash table is 128 bytes. Thus, we need to load and store (most times with the same value) at least 8 times, with 128-bit SIMD registers.

To alleviate this problem, we divide payloads per key and access only the ones in the same offset as the matched key. For query 4, ideally we would like to access 16 bytes to read the 4 keys and 28 bytes for a single payload. We would have to compare with the keys, extract the offset (0-3) of the matched key, then load and update a single payload. In case no key actually matched, we must generate a mask to nullify it. In practice, we use a combination of selective payload update and partial nullification (i.e., when you rewrite 2 or 4 words but only one changes value). To extract the key offset, we invert the equality mask and get the minimum value and offset inside it (using `phminpusuw` from SSE4.1).

2.5 Non-Candidate Counters

As discussed in Section 2.2, we use non-candidate counts to validate our heavy hitter candidates. The most generic implementation is to maintain a table of counts that is separate from the table containing the candidates. We can fine tune the size of the candidate and non-candidate tables to match the dataset distribution. Since smaller tables will fit in faster memory, we aim to limit the size of the tables subject to our accuracy requirements.

Suppose that we have assembled our heavy hitter candidates and we know (or estimate) that they have a cumulative frequency of A. If $1 - A$ is much larger than A, then an equal number of non-candidate counters would not help at all. We need more non-candidate counters to dilute the $1 - A$ to below the desired threshold.

Let m denote the number of non-candidate counters, and n the total count among all non-candidates as estimated from the sample. It is overly optimistic to simply divide n by m to derive a threshold, since the true threshold will be the count in the largest hash bucket, not the average bucket count. We would like to obtain an estimate $b(m, n)$ of the total count in the largest bucket. We can use the algorithm of [9], which is somewhat optimistic because it assumes n independent choices; in our case duplicate keys map to the same slot. Nevertheless, it is likely to be a reasonable estimate if the individual item frequencies are small among the non-candidates. Alternatively, one could try several different m values on the sample to get an empirical estimate for $b(n, m)$. The inclusion of additional keys in the candidates table can reduce individual item frequencies among the non-candidates, and thus the $b(n, m)$ estimate.

If C_K is the relative frequency of the Kth item in the sample, we need $b(n, m) < C_K N$. Larger values of m will help the accuracy thresholds, and will only hurt performance once a cache size threshold is crossed. One will therefore typically make one of a small number of choices for m based on the maximum capacity of each cache level.

Ideally, we would like to update only one of the two tables each time. Branching code could achieve that. However, it would make the throughput dependent on the dataset distribution, because of mispredictions that will occur. Instead, we always update both tables, nullifying only the update to the candidates table. The non-candidates table is updated every time. At the end of the probing loop, we subtract each candidate's count from its respective non-candidate table location to obtain the true non-candidate counts.

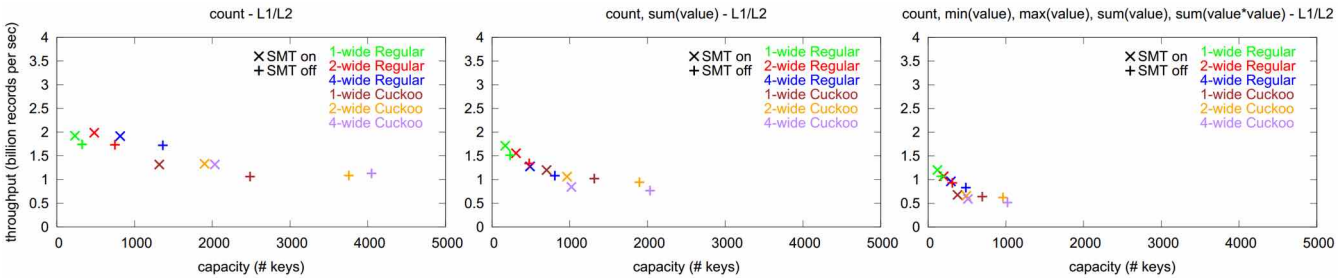

Figure 3: Throughput vs. capacity skyline for hybrid L1/L2 table (all queries).

3. EXPERIMENTAL EVALUATION

The platform we used for our experiments has 2 Intel E5620 processors (Nehalem) @ 2.4GHz with 4 cores each. The size of L1 cache is 32KB for data and 32KB for instructions, L2 is 256KB and both are private per core. The L3 is 12MB and shared throughout each chip. The processors include all SSE instructions and support simultaneous multithreading of 2 threads per core. The total RAM capacity is 48GB. We use 3 queries that reflect increasing complexity:

```
Q1:  select count(*) from table group by key ...
Q2:  select count(*), sum(value)
     from table group by key ...
Q3:  select count(*), min(value), max(value),
           sum(value), sum(value * value)
     from table group by key ...
```

Figure 3 shows the throughput (in billions of records per second) of various configurations. The candidate table is 32KB (L1) and the non-candidate table is 192KB (3/4 L2). The number of slots depends on the size of the aggregates. When SMT is enabled, table sizes are halved. We used 10^6 distinct keys and a randomly generated Zipf with $\theta = 1$.

Not all methods are suitable for all queries. We are interested in the configurations that are on the skyline, i.e., there is no other configuration greater in both throughput and capacity. The capacity of methods decreases with the complexity of the query, since we use space for aggregate calculation. For example, `Q2` needs double the space of `Q1`, so it fits almost half as many elements. `Q3` needs double the space of `Q2`. The non-candidate counters are unaffected.

For simple aggregation workloads, such as count, the 4-wide table is slightly faster than the flat table: The data is already vectorized, so fewer instructions are needed in the inner loop. The number of instructions becomes less important when we access more than one column or use cuckoo hash tables, as there is more overlap with cache accesses.

Bucketized tables increase capacity by allowing more keys to be perfectly hashed. For the 2-wide table, we can hash twice as many candidates as the basic table, and for the 4-wide table we almost quadruple the number of candidates. Since the probability of fewer collisions does not scale linearly, these ratios drop as we go from L1-resident to L2.

As the select clause of the query becomes more complicated, hash tables need to hold and update a longer payload. The performance impact is more noticeable in bucketized hash tables. We mentioned in Section 2.4 a way to bypass long re-writes, by extracting the offset of the matched key to work on a single payload. This technique is not fast enough to always be our best choice; computing few aggregate functions on 2-wide tables, it may be faster to process both payloads, even if at most one may be updated.

For most queries, the 4-wide cuckoo configurations are the slowest. For `Q2` they run 20% slower than 2-wide cuckoo tables. Since the capacity gap between these two methods is less than 10%, the trade-off favors the 2-wide cuckoo table, unless the query really needs that extra 10% capacity.

The most important attribute of configurations with non-candidate counters is the accuracy they can achieve, i.e., how many heavy hitters can be validated based on the maximum non-candidate count. In Figure 4 we show the number of heavy hitters achieved under Zipf for different θ using 100,000 distinct keys. Skew decreases as one moves to the right in Figure 4; $\theta = 0$ is uniform. As expected, from near-uniform distributions we cannot extract any heavy hitters. For very high θ values, the distribution has relatively few distinct keys and we can extract all of them.

The cuckoo tables validate many more candidates than the regular tables, and bigger buckets also help, even when the methods use the same number of miss counters. The L2-resident tables allow us to validate 3–8X more heavy hitters than the L1-resident tables, because the non-candidate distribution is spread over many more counters. Admitting

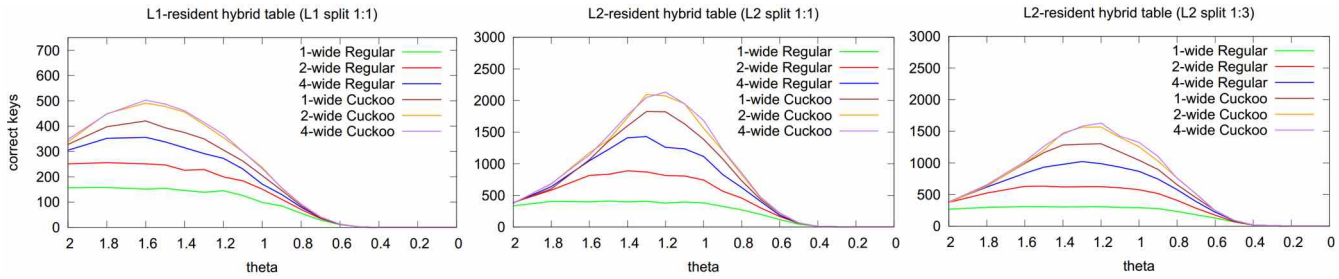

Figure 4: Correct keys under Zipf distribution for L1 & L2 tables (Q2, SMT off).

Candidates	Non-Cand.	Scheme	1-wide			2-wide			4-wide		
			Time	HH.	Freq.	Time	HH.	Freq.	Time	HH.	Freq.
L1 ×1/2	L1 ×1/2	Regular	2.32	9	3.62	3.07	10	3.62	3.47	10	3.62
		Cuckoo	3.41	12	3.62	3.93	12	3.39	4.78	12	3.45
L1 ×1/4	L1 ×3/4	Regular	2.15	14	3.39	2.55	14	2.95	3.28	16	2.72
		Cuckoo	3.47	15	2.78	3.73	16	2.78	4.59	16	2.70
L2 ×1/2	L2 ×1/2	Regular	3.59	92	1.00	3.67	145	0.75	4.11	187	0.69
		Cuckoo	4.49	217	0.63	4.67	260	0.61	5.72	273	0.57
L2 ×1/4	L2 ×3/4	Regular	2.77	103	0.95	2.98	146	0.78	3.68	187	0.67
		Cuckoo	3.92	215	0.62	4.28	260	0.59	5.38	268	0.57
L1	L2 ×3/4	Regular	2.59	84	0.89	2.83	121	0.88	3.55	141	0.80
		Cuckoo	3.74	162	0.73	4.11	179	0.72	5.24	179	0.71

Figure 5: Time (sec), # correct HH, min freq ($\times 10^{-4}$) on wikipedia with separate tables (Q2, SMT off).

more candidates is lessening the cumulative frequency of the rest of the dataset, dispersing a smaller load across non-candidate counters and decreasing the validation threshold. Similarly, it pays to use a 1:1 split rather than a 1:3 split. Thus, on Zipf distributions the space is better utilized for extra candidates rather than extra non-candidate counters.

In our final experiment, we test our method on a realistic dataset. We used Wikipedia access data, provided freely in aggregated form. We assembed files storing URLs and access counts for the period from the 1st to the 14th of January 2012 for English URLs. We generate two columns, the URL coded as a 32-bit unique id, and the time of each access at hourly granularity. A row is generated for each visit, and the rows are randomly shuffled. The total number of rows is 3,463,321,585 and there are 102,216,378 distinct keys (URLs). In the dataset, the three heaviest hitters have a frequency of about 1.6%. The first 100 keys have a total frequency of 6.65% while the first 10,000 account for 25.3%. We use a sample size of 10,000,000, which requires 100ms.

We are interested in three results per case: the number of heavy hitters validated, the execution time and the frequency of the minimum-frequency element we extracted. We summarize these results in Figure 5. We report an average of 100 runs. The "Hit" and "Miss" column describe the location of the candidate and non-candidate table respectively.

We studied the sample-based estimations for how many heavy hitters we will extract in each case. The estimations are never more than 10–15% from the final result. Thus, to choose the best fitting method, adding a 15% "fudge factor" to the sample-based estimate seems empirically justified.

To extract the most heavy hitters, the best configurations were the L2-resident ones, as expected. We extract more than 250 heavy hitters using the bucketized cuckoo methods, on average. If fewer heavy hitters are needed, alternative methods could do the job in less time. The ideal minimum frequency is the cumulative frequency of non-candidates ($\approx$ 0.85 here) divided by $49,152$ (3/4 of the L2), i.e., 1.7×10^{-5}. We differ from it by a small factor, reaching $< 6 \times 10^{-5}$.

4. CONCLUSIONS

We presented a method to quickly aggregate the heavy hitters of a table, by first sampling them, computing their exact aggregates and then validating. By not aggregating all distinct groups, we can focus on a small working set composed of the most important elements. We build cache-resident structures to hold them and rapidly compute their aggregates. Through careful use of SIMD operations, we can boost performance and be faster than prior aggregation methods when the number of groups is small. Finally, we created a menu of options to exploit tradeoffs, and described a way to pick the most appropriate.

5. REFERENCES

[1] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[2] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, 2002.

[3] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, 2007.

[4] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, 2010.

[5] G. Cormode et al. An improved data stream summary: the count-min sketch and its applications. *J. Algo.*, 55(1), 2005.

[6] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *VLDB*, 2008.

[7] M. Dietzfelbinger et al. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1), 1997.

[8] M. Dietzfelbinger and U. Schellbach. Weaknesses of cuckoo hashing with a simple universal hash class: The case of large universes. In *SOFSEM*, 2009.

[9] W. J. Ewens and H. S. Wilf. Computing the distribution of the maximum in balls-and-boxes problems with application to clusters of disease cases. *PNAS*, 104(27), 2007.

[10] R. M. Karp et al. A simple algorithm for finding frequent elements in streams and bags. *ACM T. Dat. S.*, 28(1), 2003.

[11] S. Manegold et al. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3), 2000.

[12] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[13] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3), 2006.

[14] J. Misra and D. Gries. Finding repeating elements. Technical report, Cornell University, 1982.

[15] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *VLDB*, 4(9), 2011.

[16] R. Pagh et al. Cuckoo hashing. *J. Algorithms*, 51(2), 2004.

[17] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.

[18] P. Roy, J. Teubner, and G. Alonso. Efficient frequent item counting in multi-core hardware. In *KDD*, 2012.

[19] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

[20] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, 2002.

[21] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN*, 2006.