High Throughput Heavy Hitter Aggregation

Orestis Polychroniou
Department of Computer Science
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross
Department of Computer Science
Columbia University
kar@cs.columbia.edu *

ABSTRACT

Heavy hitters are data items that occur at high frequency in a data set. Heavy hitters 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.

We design cache-resident, shared-nothing structures that hold only the most frequent elements from the table. Our approach 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 candidates. Finally, if necessary, a validation step identifies the true heavy hitters from among the candidates based on the query specification.

We identify trade-offs between the hash table capacity and performance. Capacity determines how many candidates can be aggregated. We optimize performance by the use of perfect hashing and SIMD instructions. SIMD instructions are utilized in novel ways to minimize cache accesses, beyond simple vectorized operations. We use bucketized and cuckoo hash tables to increase capacity, to adapt to different datasets and query constraints.

The performance of our method is an order of magnitude faster than in-memory aggregation over a complete set of items if those items cannot be cache resident. Even for item sets that are cache resident, our SIMD techniques enable significant performance improvements over previous work.

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.

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

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 terms of item counts, or in terms of other measures such as total sales. They may 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 many real-world datasets, skew in the data distribution 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 elements.

Now that systems with very large main memories are available, the performance bottleneck has shifted from I/O to CPU and memory [15]. 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 [2, 5]. 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 [6]. Contention for popular data items causes significant delays, serializing execution and prevents 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 in negligible time.

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 [23]. 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 novel 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. [23] even for cache-resident aggregates.

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. A large sample gives us strong statistical guarantees about the likelihood that items that are not quite heavy hitters in the sample, according to a specified frequency threshold, might turn out to be heavy hitters in the full data set. One can tune the sample size and/or the number of elements chosen to reduce the probability of missing a heavy hitter to a vanishingly small number.

If one is not satisfied with a probabilistic guarantee, or if one wants to find the top heavy hitters without specifying a frequency threshold, then we provide a second class of algorithm that dynamically determines a threshold. The 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 based on perfect hashing, 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 [20] 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 given one's application constraints.

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 2) 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.

Summarizing our contributions:

- We introduce a novel approach to aggregate for heavy hitter elements with cache resident structures. Focusing on the most important items, we preserve the quality of the resulting summary and benefit from fast in-cache processing.
- We accelerate our method by the use of perfect hashing and SIMD instructions to eliminate branching and minimize cache accesses. SIMD instructions are utilized in a novel non-uniform way, beyond simple vectorized operations.
- By utilizing bucketized and cuckoo hash tables, we can increase the capacity at the cost of speed. Increased capacity allows more heavy hitters. We adapt to each dataset and query constraints and pick the best option between the alternatives.

In the following section we present related work. In Section 3 we formulate the problem and present key concepts. In Section 4 we describe our approach in more detail, giving illustrative examples. In Section 5 we show our experimental evaluation. In Section 6 we describe refinements and conclude in Section 7.

2. 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 [13, 18], Lossy Counting [16], and Space Saving [17]. 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 [4] and Count Min Sketches [8]. Such algorithms compute summaries of the distribution that allow the approximate inference of heavy hitters and other queries. See [7] for an extensive analysis.

Aggregation on modern multi-core CPUs has been studied in [5, 6, 23]. 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.

Database algorithms that are sensitive to modern hardware have been studied in several contexts. MonetDB/X100 [3] has been designed with CPU cache performance in mind. HIQUE [14] and HyPer [19] provide query compilation for efficient execution. Staged databases [12], describe breaking up execution in stages and process a group of sub-requests

at each stage, thus exploiting data and work commonality. Apart from aggregation, hash joins have also been redesigned for modern multicore CPUs [2].

Single Instruction Multiple Data (SIMD) instructions have been used to speed up database algorithms [24], including hash probing in bucketized cuckoo hash tables [22]. The primary benefit is the reduction of cache accesses. SIMD execution has a secondary benefit of being able to avoid branches for many inner-loop computations [22, 24], an important benefit since branch mispredictions can be a significant performance overhead. Most past work on SIMD operations uses them in arrays of elements of the same type, performing many instances of the same operation using one instruction. We go beyond this kind of uniform processing, handling different kinds of work in each SIMD cell.

3. CONCEPTUAL DESIGN

3.1 Definitions

We use two different definitions for heavy hitter elements. The first is by specifying a minimum frequency. This frequency serves as a lower bound that aggregated groups must satisfy to be included in the output of the query. The second way is by keeping the top-K results, after sorting them in descending order of aggregated count.

The frequency-bound specification requires the user to define the frequency that distinguishes heavy hitters from the rest of the keys, as in the following SQL query.

```
select product_id, count(*)
from sales
group by product_id
having count(*) > 0.001 * (select count(*)
                           from sales);
```

With top-K queries the user does not need to specify a frequency threshold to distinguish the heavy hitters, as in the SQL query below.

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

Additional aggregates may be included in the `select` clause to generate more information. Heavy hitters defined by weighted counts are discussed in Section 6.1.

3.2 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.

3.2.1 Frequency Bound Queries

Suppose that the user specifies a threshold of p, so that any items occurring with a relative frequency above p are considered heavy hitters. Let n be the size of a random sample of the items. Consider a single item x that we hypothesize is a heavy hitter, i.e., we hypothesize that x's frequency in the full data set is at least p. We observe the number of times c that x occurs in the sample. How much less than np does c have to be, before we have high confidence that our hypothesis that x is a heavy hitter, is wrong?

To quantify this probability, we assume conservatively that x has a true frequency of exactly p. Then, assuming uniform sampling, the distribution of c is binomial with parameters n and p. Then we can use Chernoff's inequality to obtain an upper bound on the cumulative probability $F(k; n, p)$ that at most k items were observed in a binomial sample of size n and probability p. Suppose that we reject x if $c/n < fp$, where f is a parameter with $0 < f \leq 1$. Thus $k = nfp$, but since there are at most $1/p$ true heavy hitters, we multiply by $1/p$ to conservatively bound the probability that *some* true heavy hitter has been rejected.

$$P_{miss} = \frac{1}{p} \cdot F(nfp; n, p) \leq \frac{1}{p} \cdot e^{-\frac{np(1-f)^2}{2}}$$

This bound is quite strong, and decays exponentially in n and p. For example, suppose $n = 10^6$, $p = 2 \times 10^{-4}$. Setting $f = 1/2$, we expect a count of 200 in the sample, and will reject x if its count is less than 100. The probability that a true heavy hitter has a count less than 100 is bounded by $0.5 \times 10^{-6} \cdot e^{-25} \approx 7 \times 10^{-8}$.

The f parameter will affect how many elements we must include in our table in the aggregation phase. In the example above, we would compute aggregates for all elements with sample counts of at least 100. We are trading space for accuracy, and need to make sure that our table structures have sufficient capacity. We are also implicitly trading time for accuracy, because faster hashing schemes are available when fewer items need to be stored (see Section 3.4). Alternatively, we can tune the sample size n to improve accuracy at the cost of a more time-consuming sampling phase. The query optimizer can evaluate the options by choosing the configuration that minimizes execution time under a given accuracy requirement.

If even a miniscule probability of missing a heavy hitter is unacceptable, then the validation method described in Section 3.3 can be used instead.

3.2.2 Top-K Queries

For top-K queries, we also 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 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 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 in Section 3.3.

3.3 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 2, the largest aggregate A among all hash buckets provides a conservative empirical bound on the heaviest hitter that is not among the candidates. If there are K candidates with aggregate above A, then we know we have the top-K. For frequency bound queries, if A is below the user-defined threshold, then we know that only candidates can be heavy hitters.

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. For frequency bound queries, the user's threshold is too fine. For top-K queries, the specified K is too big.
- The user is not especially ambitious when specifying K for a top-K query. 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. As previously mentioned, 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 5.5. In general, for datasets with sufficient skew, we will be able to successfully and efficiently identify hundreds of heavy hitters.

3.4 Perfect 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 [9]. 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 [11] 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 [20]. 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 [22].

Multiplicative hashing can perform poorly in cuckoo hashing schemes [10], although the poor behavior is less noticeable in bucketized cuckoo hashing [22]. Since we are repeating the process a few thousand times to get as many keys as possible in the table, we overcome this problem and achieve very high occupancy rates. Insertion time, normally a weaker point of cuckoo hashing for bigger hash tables, is not an issue here.

4. DESIGN FOR PERFORMANCE

To get high performance, we must implement our methods so that they run efficiently on modern architectures. In particular, we used all of the cores available on our hardware platform, and use SIMD instructions in novel ways to maximize processor utilization. At the same time, we use no conditional branches within the inner loop, avoiding performance pitfalls caused by branch mispredictions. The use of perfect hashing (Section 3.4) is essential to avoid the branching implicit in chaining. We also tune our methods for the required number of candidate heavy hitters. As we shall see experimentally, with fewer heavy hitters one can obtain better throughput.

4.1 Update 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 [21]. Second, comparison results can be used as masks for subsequent operations [24]. We call this general approach *nullification* and illustrate it by rewriting the loop above as:

```
equal = (key != table[hash].key ? 1 : 0) - 1;
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. After this simple change, the (scalar) code runs 8% faster.

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 Intel's SSE instruction set, but the principles used would also apply to other SIMD sizes. A SIMD register can be interpreted as four 32-bit values, or as two 64-bit values. For the example above, we assume a 32-bit key, a 32-bit integer count, and a 64-bit integer sum.

A novel aspect of our approach is that we use SIMD data types containing different types of cell data, in contrast to typical vectorization optimizations that work on simple arrays of values of the same type. For example, in our hash cell for the sum/count example, we will pack a key, a count, and a sum into a single 128-bit SIMD unit. We assume SIMD operations are available for bitwise AND, for 32-bit vectorized comparisons, for 64-bit vectorized addition, and for 32-bit vector shuffling. SIMD vector comparisons give a zero cell when the comparison fails, and an "all 1" cell if the comparison succeeds. Figure 1 shows how data flows during the probe/aggregation process.

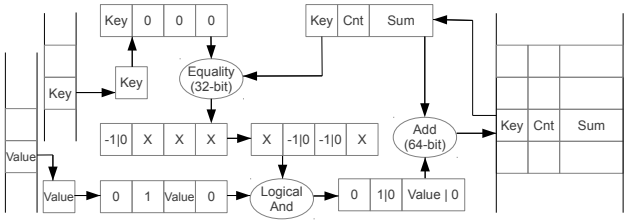

Figure 1: SIMD method for count, sum(value).

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 speed of this method (which will be described in more detail in Section 5) is 76% better than the original scalar code.

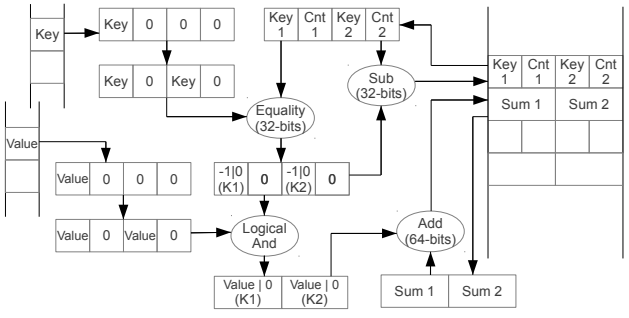

Figure 2: 2-wide table for count, sum(value).

SIMD techniques become even more useful for bucketized hash tables. Figures 2 and 3 show SIMD probe implementations for buckets of size 2 and 4 respectively. Note the subtraction of -1 to increment the count and the shuffle of 32-bit masks to nullify 64-bit sums.

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 bitwise inverse.[1]

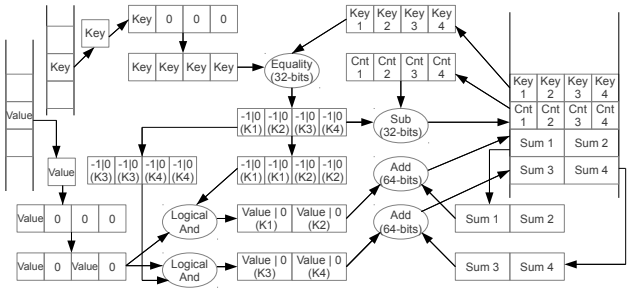

Figure 3: 4-wide table for count, sum(value).

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. To alleviate this problem with bucketized tables, we divide payloads per key and access only the ones in the same offset as the matched key. Figure 4 shows one example query. The new element in this example is the "min value & index" step that finds the index within the SIMD vector of the smallest value. There is an SSE4.1 instruction that provides this functionality. Figure 4 also illustrates the use of SIMD for max and min aggregates.

One last interesting case occurs when we want to use a single SIMD entry to derive both the candidate and non-candidate match counts for validation, as described in Section 3.3. In a hash cell entry we keep a "Yes" count that counts matches, and an "All" count that counts both matches and nonmatches that hash to this bucket. The non-candidate count can be computed at the end by subtracting the Yes-count from the All-count. We start counts with 1 instead of 0. (At the end of the computation we subtract 1.) We can then generate cells in SIMD registers that are guaranteed to be 0 by comparing the count with zero; we then shuffle and move it to the key offset. Figure 5 shows our inner loop implementation in this case for a simple count aggregation.

This way of computing non-candidate counts, which we call a "piggyback table," assumes that we use one non-candidate count per hash bucket. In Section 4.2 we will also consider alternative implementations where the non-candidate counts are stored separately and the number of counts can be chosen independent of the number of candidates. The methods will be compared experimentally in Section 5.

Versions for other combinations of aggregates are similar to those described above (e.g., use of floating point columns). For cuckoo versions, we perform the same basic operations on multiple hash entries. We read all table entries before writing any of them. In the event that multiple hash functions yield the same slot for a key, only one of the writes will be effective, so we avoid double counting.

[1]If we need signed numbers, we can store them in unsigned format and add or subtract the appropriate offset before displaying them. For simple sums (ie. sum(value)), we subtract the count $\times 2^{31}$, offline at the end instead of doing conversions with each update.

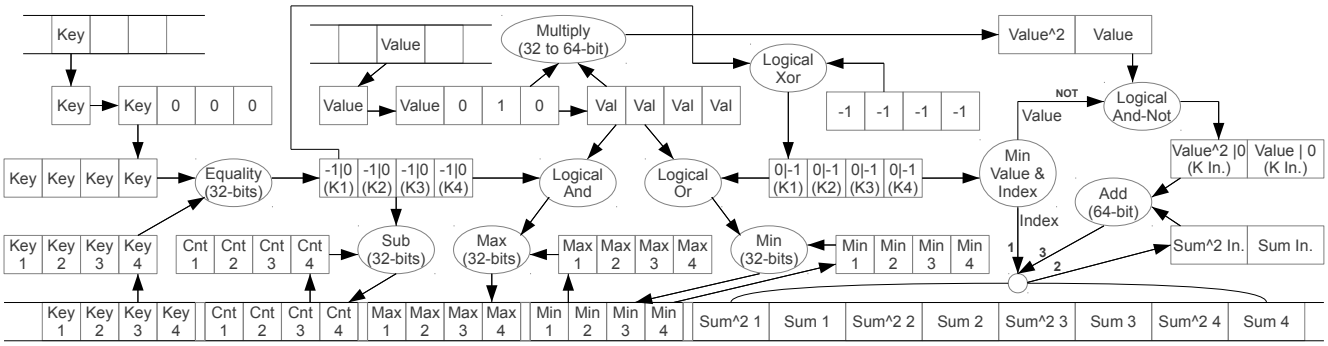

Figure 4: 4-wide table for count, max(value), min(value), sum(value), sum(value*value).

While implementing these SIMD methods, we sometimes observed that pure SIMD implementations were not always best. For example, it was better to leave the hash computation in scalar code rather than converting it to vectorized code. Since SIMD instructions are performed in different circuits from scalar instructions, the code is more efficiently parallelized by the out-of-order CPU if both kinds of instructions are being used. If the processor is already SIMD-limited, then adding more SIMD instructions will slow it down, while leaving the scalar pipeline underutilized.

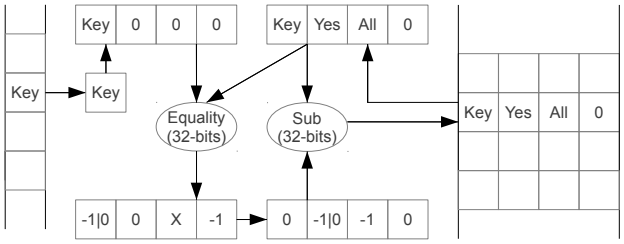

Figure 5: Piggyback table for count.

4.2 Non-Candidate Counters

As discussed in Section 3.3, 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 [11], 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 $b(n, m)$. As previously mentioned, 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.

For frequency based queries with threshold p, we need to choose m such that $b(n, m) < pN$ where N is the total data size. For top-K queries, 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 significantly hurt performance once a cache size threshold is crossed. One will typically make one of a small number of choices for m, based on the 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.

With 32-bit counters, it takes over 4 billion elements to overflow a counter. If there is a risk of overflow, we can pause the aggregation every 2 billion items, and look for items with counts above 2 billion. For each item found, we add 2 billion to a separate 64-bit counter for that item, and subtract 2-billion from the count in the hash table. The performance impact is minor since the hash table would be scanned so rarely. Threads can perform this process independently.

5. EXPERIMENTAL EVALUATION

5.1 Platform

The platform we used for our experiments has two Intel E5620 processors based on the Nehalem architecture, each with 4 cores. The L1 data cache is 32KB, the L2 cache is 256KB and is private per core, the L3 is 12MB and shared on each chip. The processors run at 2.4 GHz and support simultaneous multithreading of 2 threads per core and SIMD version SSE4.2. The total memory is 48GB.

Unless otherwise noted, keys and values are 4-bytes. Data is stored columnwise as arrays. That way, unneeded data for each row does not need to be read from RAM. For performance and capacity experiments, we ran the experiments 25 times and report the median numbers.

To calibrate our methods, we measured the performance of a simple parallel scan of a set of records, performing a simple bitwise SIMD `OR` of each column with either a register, successive elements of an L1-resident array (with wraparound), or successive elements of an L2-resident array. Using a register tests pure throughput, while the other cases test the additional cost of a SIMD cache load and store.[2] Table 1 shows the throughput achieved. These numbers represent upper bounds on the performance of any heavy hitter method on this platform.

Cache Updates	1 column	2 columns
None	8.05	4.05
L1 resident	7.28	3.67
L2 resident	4.55	2.29

Table 1: Upper bounds (billions of records / sec).

5.2 Queries and Table Structures

For our initial experiments we use four queries, chosen to reflect increasing complexity. We already used three of them in Section 4.1, in the figures describing the SIMD data flow. These queries will be evaluated in both a frequency based manner, and using a top-K formulation.

```
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)
     from table group by key ...

Q4:  select count(*), min(value), max(value),
            sum(value), sum(value * value)
     from table group by key ...
```

We consider distributions for the `key` column based of the Zipf distribution that allows the degree of skew to be varied according to a parameter θ. In these experiments we used one billion rows and one million distinct keys. In the sampling phase, using 16 threads, we sample and assemble counts for 1,000,000 entries in 20 ms. (This is a very small fraction of the total processing time, and is not included in the measurements below.) We uniformly pick elements from the dataset, ignoring possible duplicates. We do not use the optimization of counting all elements inside each cache line we fetch, which would improve sampling speed but damage the quality of the sample due to possible correlations of neighboring elements.

We will use a variety of configurations determined by: (a) the hash table bucket size (1, 2, or 4 elements); (b) whether conventional hashing or cuckoo hashing is used; (c) whether 8 threads ("SMT off") or 16 threads ("SMT on") are used; and (d) the hash table footprint (either L1 resident or L2 resident). When SMT is on, there are two threads sharing the same L1/L2 cache, and so each is allocated half the space. When we also compute non-candidate counts, we consider (i) using the same cache level for both tables; (ii) a "hybrid" with an L1 resident candidate table and an L2 resident non-candidate table; and (iii) a piggyback table.

To determine the capacity of a hashing scheme, we set a time budget of 50ms (not included in the measurements), which is small relative to the cost of processing one billion rows. The number of multipliers we can try in this time budget depends on the scheme. For cuckoo tables, we can try a few thousands. Otherwise, they can be from a few millions down to a few hundred thousands. For example, an L1-sized 4-wide table on `Q1` can try half a million multipliers. To see how sensitive the capacity is to the time spent choosing multipliers, we tried a 5 sec. budget, and found that the capacity increased by no more than 20%. Thus, investing more time choosing multipliers is not advisable.

When the hash table entry size is a power of 2, the table size is usually a power of 2 as well. Hashing against a power of 2 buckets is faster, as it uses shifting instead of a second multiplication (for multiplative hashing). If we generalize our methods to support any query, we will come across cases where the hash table entries are not powers of 2. We can choose to either pad this space or use it. If we pad it, we keep the same throughput; if we use it, we lose some performance but gain extra capacity and/or validation accuracy.

We observed that seemingly minor changes to the inner loop code sometimes resulted in a noticeable (5–10%) performance change, particularly for L1-resident tables that are operating close to the hardware limit. While we tried hard to optimize the code, it is possible that an alternative instruction order, for example, could perform slightly better. This phenomenon is less visible in L2 resident tables where the latencies of slower cache accesses mask instruction latencies. When there are multiple implementations that answer the same query using different SIMD methods, we show the performance of the best one for the given setting.

5.3 Throughput vs. Capacity

Figure 6 shows the throughput (in billions of records per second) of various configurations as a function of the capacity of the candidate table. The four rows of charts correspond to the four queries of Section 5.2. The first column of Figure 6 displays L1 resident tables and the second column displays L2 resident tables. The third column presents the hybrid scheme that fits the candidate table in the L1 cache and has a table of non-candidate counts in the L2 cache. We used 10^6 distinct keys and a randomly generated Zipfian key distribution with $\theta = 1$.[3]

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

[2]In our cache update we `OR` four elements at once using SIMD. If we use scalar code to update one element at a time, the performance decreases by about 40%.

[3]The throughput rates we see in this experiment are slightly biased by the cumulative frequency of heavy hitters in the column. If there is enough of a concentration of frequency in a small number of heavy hitters, then the most frequently accessed items will be L1-resident even when the table is sized for the L2 cache. When we spread the distribution more evenly throughout an L2 resident table, we observed a modest 10–15% performance decrease.

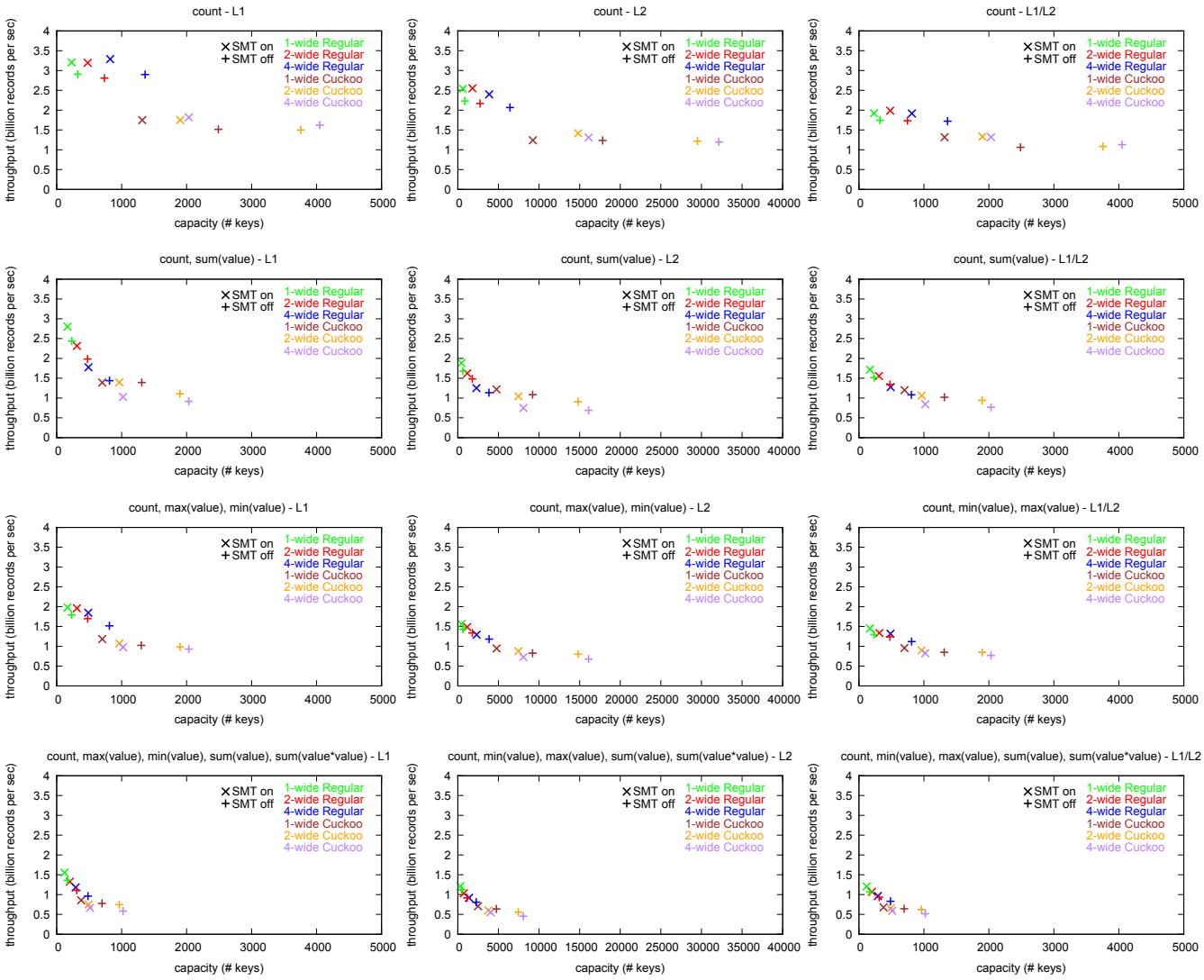

Figure 6: Throughput vs. capacity skyline for L1 & L2 tables and hybrid L1/L2 table.

calculation. For example, `Q2` and `Q3` need double the space of `Q1`, so they fit almost half as many elements. `Q4` needs double the space of `Q2` and `Q3`. These restrictions do not apply to the separate table of non-candidate counters.

For the simple count(*) aggregation, all regular hash tables have a throughput of about 3.3 billion records per second. 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.

Comparing the first two columns of Figure 6, there is a clear trade-off between capacity and throughput. The L2-resident table is about 25% slower, but the key capacity is much larger, by up to a factor of 8.

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.

SMT is mostly important for faster configurations. The SMT enabled versions give a 10% increase in the throughput for regular tables. For a regular 4-wide table running `Q1`, SMT increases throughput from 2.9 to 3.3 billion records per second. On the 4-wide cuckoo table that runs `Q4`, the increase due to SMT is marginal, while the number of keys supported is halved. Thus SMT should not be used when we need many candidates or larger non-candidate tables.

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 showed (Figure 4) a way to bypass long rewrites by extracting the offset of the matched key to work on a single payload. Still, this technique is not fast enough to always be our best choice; for 2-wide tables, it is more efficient to process both payloads, even if at most one aggregate may change its values.

Comparing the first and third columns of Figure 6 shows

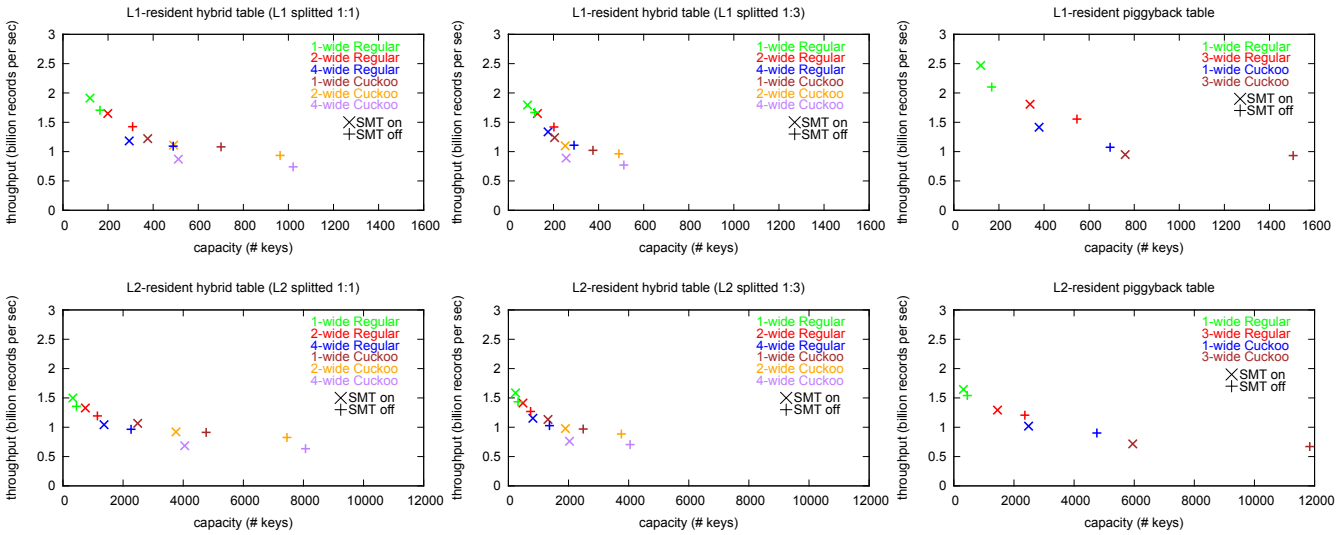

Figure 7: Throughput / capacity skyline for L1 & L2 tables with miss counters (Q2).

that the overhead of updating an L2-resident non-candidate count table is about 35% less throughput. We examine other non-candidate configurations in Figure 7, limited just to Q2. The first row of Figure 7 contains L1 configurations, while the second row contains L2 configurations. The first column divides the space in the cache in a 1:1 ratio between the candidate and non-candidate tables. The second column uses the ratio 1:3. The third column implements the piggyback table, where the two previously separate tables are merged.

Comparing the L1 and L2 charts of Figure 7, the L2 regular hash tables are faster than L1 cuckoo tables but usually have less capacity. The intuition that L1-resident tables should always perform better than L2-resident tables is wrong. The cuckoo tables access two locations and execute almost twice as many instructions.

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.

Piggyback tables are faster than hybrid configurations, where the heavy hitter candidates table and non-candidate counters table are held separate. When both tables of the separate versions are in L1, the piggyback is 25% faster.

5.4 Aggregation

In order to demonstrate the efficiency of our SIMD implementation, we compare it against a state of the art conventional aggregation implementation by Ye et al. [23] using the same experimental platform. In particular, we ask whether for grouping sets small enough to fit in cache, do we compute aggregates faster than Ye et al. [23]? In Ye et al.'s PLAT method, data is partitioned across different threads and while partitioning, a small local table is created from the first keys and remains private to each core; collisions are solved by chaining. With a small number of keys, all data is hashed using the local tables and there is no need to go to the global table. Thus, we test the efficiency of our perfect hashing scheme coupled with SIMD, against an L1-resident chaining hash table with almost no mispredictions.

We modify our method slightly to conform to the configuration described in [23]. In particular, we use 8-byte keys and values, stored row-wise rather than column-wise. These modifications have only minor impact on our methods, the most significant of which is the need for a 64-bit multiplication for hashing rather than a 32-bit multiplication. We use a query from [23] that computes the count, sum, and sum-of-squares from an array of (key,value) pairs. We use two SIMD vectors to store the key, count, sum, and sum-of-squares in 64-bit cells. We use the same hash function, and the same degree of parallelism (16 threads) as [23].

For 100 distinct keys, Ye et al. report a throughput of approximately 1.1 billion records per second. We use the L1 resident flat table, using half of the L1 cache per core to use 2 threads. This configuration can support approximately 120 keys. It runs at 1.9 billion records per second, a speedup of 72% over Ye et al. For 1000 distinct keys, Ye et al. report a throughput of 0.7 billion records for heavy hitter input distributions. To fit 1000 keys we use an L2 resident flat cuckoo table for half L2, enable SMT and use 1/4 of L2 per hardware thread. This configuration fits approximately 1300 keys. This experiment runs at 1.2 billion records per second, a 71% speedup. Our implementation of this particular query is suboptimal, due to the absence of a matching SIMD instruction for 64-bit multiplication. Still, we prove that even without SIMD support for every operation, our method runs significantly faster than the state of the art conventional aggregation, for the same task.

5.5 Accuracy

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 8 we show the number of heavy hitters achieved under Zipfian distributions for different θ using 100,000 distinct keys. Skew decreases as one moves to the right in Figure 8; $\theta = 0$ is uniform. We use the same configurations as Figure 7 for the same query Q2.

As expected, from near-uniform distributions we cannot extract any heavy hitters. For high θ values, we extract the

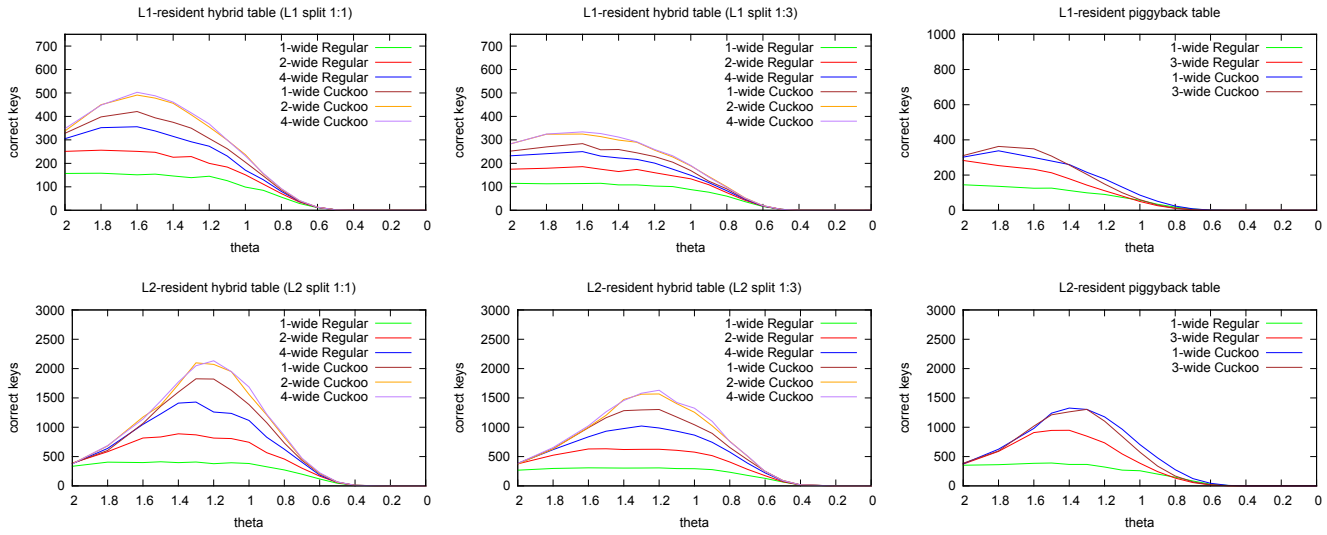

Figure 8: Correct keys under Zipfian distribution for L1 & L2 tables with miss counters (Q2).

maximum number of distinct keys. 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. If we need that many heavy hitters, the L2-resident option is the right choice. If not, the L1-resident option is faster to compute.

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. Admitting 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 of the cache. Thus, on Zipfian distributions the space is better utilized for extra heavy hitter candidates rather than for extra non-candidate counters.

5.6 Wikipedia Data Set

In our final experiment, we will test our method on a realistic dataset. For this purpose, we used Wikipedia access data, provided freely by Wikipedia in aggregated form.[4] We assembled 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, one representing the URL coded as a 32-bit unique id, and another coding the time of each access at hourly granularity. A row is generated for each Wikipedia page 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). We use a sample size of 10^7 for this dataset, which translates to about 200ms for sampling (a small overhead relative to the processing time).

The Wikipedia dataset is not as skewed as we initially expected. The heaviest hitters, which are the main page and two special index pages, each have a frequency of about 1.5%. The first 100 keys have a cumulative frequency of 6.65%, while the first 10,000 account for 25.3%.

We use the same configurations that we used in Figure 7 and in the hybrid configuration of Figure 6. We are mostly interested in three results per case: the number of heavy hitters validated, the number of heavy hitters we estimated we'd extract based on just the sample, and the frequency of the minimum-frequency element we extracted. We summarize these results in Table 2 for separate miss counter table configurations and in Table 3 for piggyback tables. The results are given in the form *extracted/estimated* and the minimum frequency is scaled by 10^{-4}. The numbers reported are the medians of 25 runs.

In practice, the heavy-hitter count estimates based on the sample were within 20% of the actual counts 97% of the time, and within 10% of the actual counts 89% of the time. The reliability of this estimate is important for the DBMS to be able to choose the best method for a query.

To extract the most heavy hitters, the best configurations were the L2-resident ones, as expected. We extract up to 273 heavy hitters using the L2-resident bucketized cuckoo methods. If fewer heavy hitters are needed, alternative methods could do the job in less time. The minimum frequency for the bucketized cuckoo methods is close to the limit imposed by the size of the non-candidate table. If the non-candidate table uses 3/4 of the L2 cache, we store 49,152 miss counts. The best possible frequency bound if candidate counts are ignored[5] is thus about 2×10^{-5}. The minimum frequency of the last heavy hitter in the L2-resident cuckoo tables is less than 6×10^{-5}, a small multiple of the ideal.

We also varied the sample size. With a sample size of 10^6 rather than 10^7, the best cuckoo-based scheme identified 235 true heavy hitters rather than 273. With 10^8 samples (taking 2 seconds to sample, which is no longer negligible) the best scheme identifies roughly the same number of heavy hitters as with 10^7 samples.

The actual query answers provided some potential insights. The "SOPA" page ranked 29th, and its mean access date was close to January 14th. The "2012" page ranked 85th, and its mean access date was close to January 1.

[4] http://dumps.wikimedia.org/other/pagecounts-raw/

[5] For the Wikipedia data, about 82% of the data remains when the candidate counts are subtracted; the impact on this bound is therefore small.

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

Table 2: time (sec), # heavy hitters, min frequency ($\times 10^{-4}$) on Wikipedia with separate tables (Q2).

Size	1-wide			3-wide			1-wide			3-wide		
	Regular						Cuckoo					
	Time	HH.	Freq.	Time	HH.	Freq.	Time	HH.	Freq.	Time	HH.	Freq.
L1	1.95	3	152.16	2.62	3	152.16	3.75	3	152.16	4.45	3	152.16
L2	2.57	26	1.92	3.14	16	2.78	4.10	44	1.54	5.36	20	2.47

Table 3: time (sec), # heavy hitters, min freq. ($\times 10^{-4}$) on Wikipedia with piggybacked counters (Q2).

Comparing the throughput rates with those of Figure 6, we notice that for this dataset, they are about 10% lower. The decrease is due to the lower cumulative frequency of heavy hitters in the Wikipedia data set compared with the synthetic Zipf data set. More than 80% of the entries are dispersed in the non-candidate table, making accesses to the L2 sized table unlikely to be L1 hits.

Despite the absence of extreme skew, our methods were effective on the Wikipedia data set. In just a few seconds, we could compute precise information on hundreds of the most popular Wikipedia URLs from 3 billion rows of raw page visit data. The minimum frequency of these heavy hitters was less than 6×10^{-5}.

6. REFINEMENTS

6.1 Weighted Frequencies

While `count` is a commonly used aggregate for defining heavy hitters, we can also support other kinds of aggregates. We need estimates on the sample to give a good estimate of the final total. Unbounded maxima would not qualify because a large outlier may be missed in the sample. The following SQL query uses counts weighted by the product price, i.e., total sales by product. If the prices are sufficiently bounded (and positive), the sample will be representative of the final aggregate.

```
select product_id, sum(price) from sales, prices
where sales.product_id = prices.product_id
group by product_id
having sum(price) > threshold;
```

We can support these kinds of queries if the weights are limited by known bounds. We make a simple modification in our algorithm to support them: While assembling the candidatates, we multiply each key count in the sample by the element's weight. We must also make sure that if there are highly weighted items, that we always include them in the candidate set of heavy hitters, even if their cardinality in the dataset or the sample is low.

6.2 Compound Queries

For queries that perform some other computation before calling the heavy hitter operator, we assume that the intermediate data is written to RAM. Since this intermediate memory traffic is sequential, it can proceed at high bandwidth and is unlikely to be a performance bottleneck. The heavy hitter calculation can then proceed without competition for cache resources. If the intermediate data is too large, our approach can run on partial data, if we have some guarantees about the sample quality.

6.3 Two Passes

Suppose the aggregation is complicated and requires a lot of space in each hash cell. This space reduces the effectiveness of the validation process by limiting both the candidate and non-candidate table cardinalities. In such a case it might be viable to do the heavy hitter aggregation in two passes instead of one. In the first pass, we compute only counts to identify and validate the true heavy hitters. In the second pass, we compute aggregates only for them.

6.4 Query Compilation

We have generated efficient SIMD configurations for particular queries. In general, it will be the job of the query compiler to generate appropriate configurations for a set of aggregates and data types. On architectures with a different mix of SIMD capabilities, alternative implementations may be needed. While we have not explored query compilation in detail here, we expect that the compilation complexity is manageable using techniques like those described in [14, 19]. Compilation time is negligible since the size of the inner loop code is small [19].

6.5 Longer SIMD

Our approach could benefit from longer SIMD registers. If we use 256-bit registers, as provided in the recent Intel AVX architecture, we can store four 64-bit keys or eight 32-bit keys efficiently. These instructions will allow us to process larger buckets efficiently, giving more trade-off options.

6.6 Applications

Imagine a user interface that allows a user to build a visualization of a data set based on an aggregation. If the full aggregation takes too long to be interactive, the system could instead compute just the heavy hitter aggregates and display those immediately. The full aggregate could proceed in the background, or could be deferred until the user explicitly indicates a need to look beyond the heavy hitters. By employing a specialized heavy hitter algorithm, such a system gives results to the user with small response time.

Data mining is another application where heavy hitters may be directly used. For example, the a-priori algorithm [1] needs to quickly identify the most frequent itemsets in a collection. We have highlighted the differences between our methods and the typical limitations imposed on streaming methods. Nevertheless, if a streaming application is able to hold a meaningful time window of data in RAM, our methods are fast enough to be able to scan that data multiple times to compute various heavy-hitter metrics.

7. CONCLUSIONS

We have demonstrated a method to quickly aggregate the heavy hitters of a table, by first sampling them, computing their exact aggregates and then validating, if required. By not aggregating all distinct groups, we can focus on a small working set composed of the most important elements and build cache-resident structures to hold them, computing their aggregates very fast. Through perfect hashing and careful use of SIMD operations, we minimize cache accesses and can further increase performance close to the hardware limit, significantly faster than the state of the art conventional aggregation. Using bucketized and cuckoo hashing, we increased cache utilization and created a menu of options. We pick the best fit for a given dataset and query.

Our experimental evaluation on both synthetic and real data shows that we can process up to several billion records per second for memory-resident data. Our approach leads to a fuller and more efficient use of modern CPU capabilities.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994.

[2] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proc. of the 2010 Int. Conf. on Management of Data*, pages 37–48, 2011.

[3] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proc. of the 2nd Conf. of Innovative Data Systems Research*, 2005.

[4] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the 29th Int. Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[5] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *Proc. of the 33rd Int. Conf. on Very Large Data Bases*, pages 339–350, 2007.

[6] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. Automatic contention detection and amelioration for data-intensive operations. In *Proc. of the 2010 Int. Conf. on Management of Data*, pages 483–494, 2010.

[7] G. Cormode and M. Hadjieleftheriou. Finding the frequent items in streams of data. *Communications of the ACM*, 52:97–105, Oct. 2009.

[8] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. of Algorithms*, 55:58–75, April 2005.

[9] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25:19–51, October 1997.

[10] M. Dietzfelbinger and U. Schellbach. Weaknesses of cuckoo hashing with a simple universal hash class: The case of large universes. In *Proc. of the 35th Conf. on Current Trends in Theory and Practice of Comp. Science*, pages 217–228, Berlin, Heidelberg, 2009.

[11] 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. *Proc. of the Nat. Academy of Sc.*, 104(27), July 2007.

[12] S. Harizopoulos and A. Ailamaki. StagedDB: Designing Database Servers for Modern Hardware. *IEEE Data Eng. Bull.*, 28(2):11–16, 2005.

[13] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. on Database Systems*, 28:51–55, March 2003.

[14] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *26th Int. Conf. on Data Engineering*, pages 613–624, March 2010.

[15] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9:231–246, December 2000.

[16] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of the 28th Int. Conf. on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[17] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. on Database Systems*, 31:1095–1133, Sept. 2006.

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

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. of the VLDB Endowment*, 4:539–550, June 2011.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, May 2004.

[21] K. A. Ross. Selection conditions in main memory. *ACM Trans. on Database Systems*, 29:132–161, 2004.

[22] K. A. Ross. Efficient hash probes on modern processors. In *23rd Int. Conf. on Data Engineering*, pages 1297 –1301, April 2007.

[23] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proc. of the 7th Int. Workshop on Data Management on New Hardware*, pages 1–9, New York, NY, USA, 2011.

[24] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proc. of the 2002 Int. Conf. on Management of Data*, pages 145–156, New York, NY, USA, 2002.