Efficient Frequent Item Counting in Multi-Core Hardware

Pratanu Roy   Jens Teubner   Gustavo Alonso

Systems Group, Dept. of Computer Science, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

The increasing number of cores and the rich instruction sets of modern hardware are opening up new opportunities for optimizing many traditional data mining tasks. In this paper we demonstrate how to speed up the performance of the computation of frequent items by almost one order of magnitude over the best published results by matching the algorithm to the underlying hardware architecture.

We start with the observation that frequent item counting, like other data mining tasks, assumes certain amount of skew in the data. We exploit this skew to design a new algorithm that uses a pre-filtering stage that can be implemented in a highly efficient manner through SIMD instructions. Using pipelining, we then combine this pre-filtering stage with a conventional frequent item algorithm (*Space-Saving*) that will process the remainder of the data. The resulting operator can be parallelized with a small number of cores, leading to a parallel implementation that does not suffer any of the overheads of existing parallel solutions when querying the results and offers significantly higher throughput.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Data Mining

Keywords

frequent items, parallelism, data flow

1. INTRODUCTION

Concerned by the looming data deluge, users are looking more than ever at the properties of their core data mining algorithms. Algorithms that adapt well to the characteristics of modern hardware—such as parallelism or advanced instruction sets (*e.g.*, SIMD)—can benefit from technology advances, which ultimately is the only way to escape a drowning in data [7, 17].

Unfortunately, key characteristics of data mining algorithms are often at odds with classical approaches to parallelism. For instance, *skew in the input data* often lies at
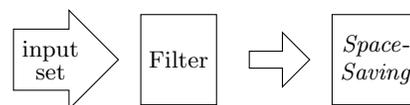
Figure 1: Input filtering for the frequent item problem. A filter absorbs much of the input data set and forwards only the remaining items to state-of-the-art *Space-Saving* instance.

the heart of the mining problem, but will lead to strong *load imbalances* in typical data partitioning schemes.

In this work we show how input skew can be turned into an advantage instead. Intuitively, we dynamically create an optimized code path for those input items that are frequent in the data set. An *input filter* routes each input item to its respective code path, making the common case fast without a noticeable slow-down of the remaining cases.

To illustrate the idea, we focus on the *frequent item counting problem*: given a data set, count the number of times each item appears in the set. In practice, this problem has to be answered with limited memory and with a single pass over the input data. Frequent item counting is used as a preliminary processing step to many data mining algorithms and it is a well-studied problem in the literature [5, 14]. Available sequential solutions are very efficient. For instance, *Space-Saving* can process close to 20 million items per second [15].

Besides being simple to describe and illustrate, the frequent item problem is interesting because it was shown to be difficult to parallelize by conventional means [6] and it is a good example of a data mining algorithm that is most meaningful when the input data is heavily skewed (otherwise all items has similar counts and none has a higher frequency than others). To date, there is no parallel implementation of frequent item counting that reaches the performance of the best sequential implementation of the algorithm.

Figure 1 sketches how we speed up frequent item counting through input filtering. A "Filter" stage at the front absorbs the most common cases with an optimized code path and routes only the remaining items to a conventional *Space-Saving* instance. Notice that, to speed up frequent item counting, we do not need to modify *Space-Saving* itself, but encapsulate all hardware awareness in the input filter. This makes our work complimentary to any optimizations that might be proposed for *Space-Saving* in the future.

As we show in the paper, input filtering can easily be parallelized using a small number of cores. Our actual im-

```
1  foreach stream item x ∈ S do
2  │  find bin b_x with b_x.item = x ;
3  │  if such a bin was found then
4  │  │   b_x.count ← b_x.count + 1 ;
5  │  else
6  │  │  b_min ← bin with minimum count value ;
7  │  │  b_min.count ← b_min.count + 1 ;
8  │  │  b_min.item ← x ;
```

Figure 2: Algorithm *Space-Saving*. A fixed number of bins monitors the most frequent items in the stream S [15].
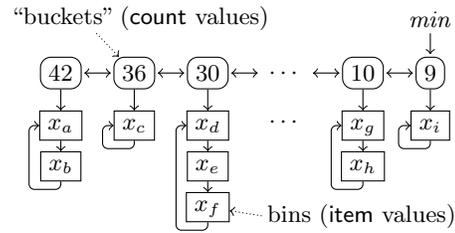

Figure 3: *Stream-Summary* data structure [15]. Bins with the same count value are grouped under one "bucket." Buckets form a double-linked list.

Zipf	data throughput	cycles/item
1.0	13.7×10^6 items/sec	135 cy/item
2.0	22.8×10^6 items/sec	82 cy/item
3.0	26.7×10^6 items/sec	70 cy/item

Table 1: Runtime characteristics of *Space-Saving*. Intel Nehalem-EX, 1.87 GHz; $k = 1,000$.

plementation reaches throughput rates (up to 200 million tuples per second) that are almost an order of magnitude faster than the best published results today. Moreover, we can sustain that throughput also in the presence of concurrent queries while the counting is in progress. Concurrent queries were shown to be a key performance blocker in earlier work [6].

2. COUNTING FREQUENT ITEMS

Several data mining techniques (such as top-k algorithms or a-priori [1]) start their data analysis by looking at the most frequently occurring items in the input data set. More formally, the *frequent item problem* is defined as

Given a data set S of size N and a threshold ϕ, return all items x that have a frequency f_x of at least ϕN in S.

To solve the problem accurately, the occurrences of *all* items have to be counted in memory, which requires an unrealistic $\Omega(N)$ space [5]. Fortunately, all common uses of frequent item counting can tolerate a small error ε in the counting result, allowing the result to include some additional items for which $(\phi - \varepsilon)N < f_x \leq \phi N$.

This *ε-approximate frequent item problem* can be solved very efficiently and with little space. ε-approximate solutions have been studied extensively (most recently by a survey article of Cormode and Hadjieleftheriou [5]), with the *Space-Saving* algorithm of Metwally *et al.* [15] (or variants thereof) generally seen as the most efficient solution.

2.1 State of the Art: *Space-Saving*

The basic idea behind *Space-Saving* is to limit space consumption by counting ("monitoring" in [15]) only the occurrences of those items that are actually frequent. The algorithm allocates a configurable number k of ⟨item, count⟩ pairs in main memory (we call them *bins*) that are supposed to monitor the most frequent items in S. Input items are then processed as listed in Figure 2. If an input item x is among those currently monitored, the corresponding count value is incremented. Otherwise, the bin with the lowest count value, b_{min} is replaced by the pair ⟨$x, b_{min}.count + 1$⟩. In the latter case, item x receives the benefit of doubt: it could have occurred as often as $b_{min}.count$ times before. *Space-Saving* never under-estimates frequencies and, hence, records $b_{min}.count$ as the frequency estimate for x.

The number of bins k reserved for monitoring is a configuration parameter that can be used to trade accuracy for space. As rule of thumb, $\lceil 1/\varepsilon \rceil$ counters are required to find frequent items with an accuracy of ε. For details we refer to [5, 15].

Implementation-wise, *Space-Saving* is typically based on two data structures. A *linked-list* variant (termed *Stream-Summary* in [15]; cf. Figure 3) keeps all bins ordered by their count values to make line 6 in Algorithm *Space-Saving* fast. Lookups by item value (line 2) are performed using a *hash table*. Both data structures allow updates with (approximated) constant-time complexity. That is, the time spent per input item is independent of the bin count parameter k.

2.2 *Space-Saving* Characteristics

Space-Saving is primarily characterized by its extremely tight loop within which it accesses a very compact data structure. The iterations of the loop execute remarkably fast (cf. Table 1): several ten million input items are processed per second, leaving less than a hundred CPU cycles for each loop iteration on current hardware. These CPU cycles per item counts are similar to those reported for the most efficient implementations of join operators [3, 10].

The primary reason for this is the cache efficiency of *Space-Saving*. For typical problem sizes, all data structures conveniently fit into modern CPU caches. Skewed input data—the typical case for frequent item computation—further increases data locality. In spite of its random access behavior, *Space-Saving* thus processes its data almost entirely within the L1 cache on typical computing hardware.

The individual iterations of *Space-Saving* update the item and count fields. The resulting *data dependency* makes it difficult to apply local optimizations, such as *loop unrolling*, *vectorization*, or fine-grained *data parallelism*. Without such optimizations, implementations leave much of the strength of modern computing hardware entirely unused.

2.3 It is Hard to Parallelize *Space-Saving*

The prevalence of multi-core hardware has spurred interest in parallel variants of *Space-Saving* [4, 6]. But it turns out the problem is hard to parallelize, even though the single-core solution is well understood and very efficient.

2.3.1 Data Partitioning

Intuitively, *Space-Saving* can be parallelized by running multiple independent executions of the original algorithm,

each of which gets one share of the input data stream. Depending on the application scenario, input data may already be provided as a set of disjoint streams (as in [6]) or additional logic performs, *e.g.*, round-robin partitioning:
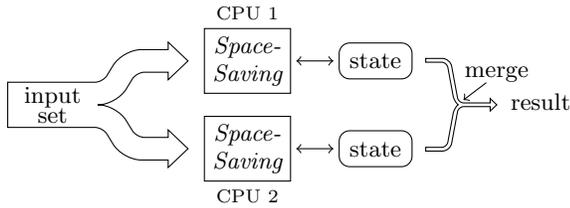
The overall counting result is then obtained by *merging* the data structures of the independent *Space-Saving* instances.

Result merging is a highly expensive operation that has to be performed each time an on-line query arrives. Das *et al.* [6] have shown that even in the presence of low query rates the overall execution time increases almost linearly with the number of parallel instances. Even for "hierarchical merging," which they showed to be the best strategy, merging is responsible for 65–98 % of the overall execution time.

Unfortunately, result merging cannot be avoided in partitioning-based schemes. Input sets to frequent item counting are usually heavily skewed (Manerikar and Palpanas report Zipfian skew in real-world data with at least $z \gtrsim 1.5$ [14]). Any value-based input partitioning (*e.g.*, by a hash on item values) will lead to strong load imbalances, leaving the setup bottlenecked in one of the *Space-Saving* instances.

Cafaro and Tempesta [4] have implemented the data partitioning strategy sketched above, using an MPI-based system. But their study ignores result merging and on-line querying and thus works around a key challenge in parallelizing the frequent item problem.

2.3.2 Shared Data Structures

Result merging *could* be avoided if all processors access the same *"shared"* data structure:
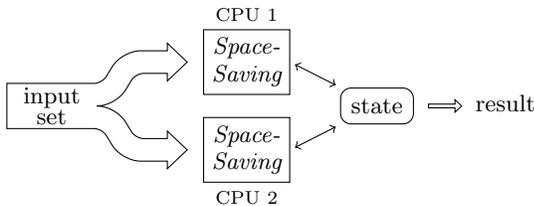
Queries would then get a consistent view on the counting state simply by reading out the shared data structure. This convenience, however, would come at a significant price. The shared state has to be protected with *locks* during every access. As mentioned before, the amount of time spent by *Space-Saving* on each input item is very small. Any additional lock maintenance code will immediately result in a noticeable performance degradation.

Cost of Cache Coherency. Given the very high cache locality of *Space-Saving*, participating processors will *not* actually share the "shared state" physically. Rather, the state ends up distributed over the respective L1 caches, depending on which piece was last accessed by which processor.

To mimic a shared memory, the system's *cache coherency protocol* will actively ship cache lines between cores whenever a requested data item cannot be found in a local cache.
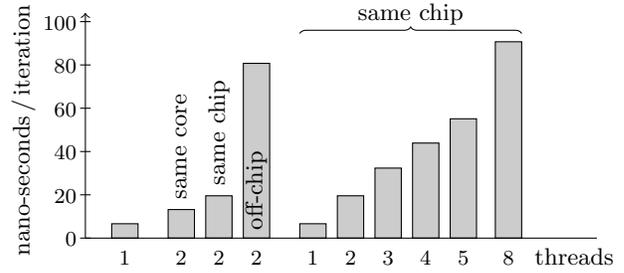

Figure 4: Cache coherency cost, depending on physical thread placement and degree of parallelism.

The cost of this can be significant: Molka *et al.* [16] measured a latency of 83 cycles for a single core-to-core cache line shipment in the Intel Nehalem architecture, which is similar to the processing time for a single item in *Space-Saving* (Table 1). The cost is intrinsic to shared data access and independent of any application-level locking strategy.[1]

A Micro-Benchmark for Cache Coherency Cost. To judge the impact that cache coherency cost can have on frequent item processing, we performed a micro-benchmark where a number of threads accesses a shared memory array. Each thread performs random counter increments in this array, following a Zipfian access pattern much like *Space-Saving* (no locking involved). On truly shared memory, thread performance should not depend on the number of threads in the system. The test platform was a 1.87 GHz Intel Nehalem EX (eight physical cores per CPU).

In the base case, exclusive data access by a single thread, each counter increment takes around 6.6 ns to complete. As Figure 4 shows, co-running threads significantly increase this access time to 20 ns (one co-runner) or even 84 ns (seven co-runners). In practice, cache coherency costs will offset any gain that parallel processing over shared data structures might bring. As mentioned before, the cost *cannot* be avoided by tuning the application's locking protocol (as proposed by [6]).

In summary, neither partitioning nor sharing can adequately leverage the capabilities of current hardware. In fact, the current trends toward de-centralization and lack of cache coherence protocols really work against using shared data structures.

3. INPUT FILTERING

The intuition behind input pre-filtering is simple: a *filter* is placed in front of an instance of *Space-Saving*, as shown in Figure 1. Out of a high-volume data stream, the filter takes out those input items that are particularly frequent and routes them through an optimized "shortcut" code path. If enough input items follow the shortcut, optimizing the filtering stage will lead to an improved overall throughput.

3.1 Algorithm

More formally, in Algorithm *Filter* we *partition* the in-memory bin set of *Space-Saving* into two groups X and Y. X consists of the *heavy hitters* for which we define a shortcut implementation. Only if an input item is not monitored

---

[1]In fact, locks are a shared data structure and thus will even exacerbate the problem.

```
1  foreach stream item x ∈ S do
2  │  look-up bin b_x in X with b_x.item = x ;
3  │  if such a bin was found then              /* shortcut */
4  │  │  └ b_x.count ← b_x.count + 1 ;
5  │  else                              /* forward to Space-Saving */
6  │  │  update Y according to Space-Saving ;
7  │  │  decrement consistency_limit ;
8  │  │  increment n' ;
9  │  │  if consistency_limit = 0 then
10 │  │  │  b_{Xmin} ← bin in X with min. count value ;
11 │  │  │  if b_{Xmin}.count ≤ n'/|Y| then
12 │  │  │  │  // actual violation of consistency limit
13 │  │  │  │  b_{Ymax} ← bin in Y with max. count value ;
13 │  │  │  │  remove b_{Xmin} from X and add it to Y ;
14 │  │  │  │  remove b_{Ymax} from Y and add it to X ;
15 │  │  │  │  n' ← n' + b_{Xmin}.count − b_{Ymax}.count ;
16 │  │  │  │  b_{Xmin} ← bin in X with min. count value ;
17 │  │  │  consistency_limit ← b_{Xmin}.count − ⌊n'/|Y|⌋ ;
```

Figure 5: Algorithm *Filter*.

within X, we forward it to a *Space-Saving* instance that maintains the set Y. To achieve overall consistency with *Space-Saving* semantics, *Filter* must ensure the following invariant: *the item with the overall minimum count b_{min} must be monitored by the bin set Y*. This is further explained later in this section.

Filtering becomes efficient whenever the size of X is small, yet catches a relevant share of the input data stream. These assumptions are reasonable since any meaningful input to the frequent item problem will be skewed. For data that follows a Zipf distribution ($z \gtrsim 1$), we find that $|X|$ values between 4 and 16 are enough to achieve effective filtering (we study the impact of this parameter in the experimental evaluation).

Figure 5 shows Algorithm *Filter* in pseudo code. Each input item is first compared to the bins in set X. If the item can be found there, the respective bin count is updated. Otherwise, bin set Y is updated according to the *Space-Saving* algorithm (line 6). In practice, bin sets X and Y are kept in separate data structures and accessed independently.

Consistency with *Space-Saving*. A partitioned execution over X and Y is consistent with the semantics of *Space-Saving* only if the processed item is either a heavy hitter (*i.e.*, it can be found in X) or if b_{min}, the bin with the minimum count value, is part of the bin set Y (*i.e.*, $b_{min} \notin X$). Otherwise, the forwarded item might overwrite a bin whose count value is not minimal. At the same time, we want to avoid checking $b_{min} \notin X$ too often, because X is part of the shortcut code path and not optimized for count-based searches.

Avoiding Expensive b_{min} Lookups. count-based searches on X can be mostly avoided if we *(a)* regularly push high-count bins into X and low-count bins into Y and *(b)* conservatively *estimate* whether b_{min} could possibly be found within X. This is what lines 7–17 do in Algorithm *Filter*. In this code, consistency_limit tells how many items can still safely be forwarded to the *Space-Saving* part without violating the consistency constraint $b_{min} \in Y$. This safety margin

is decremented for every input item that is forwarded to *Space-Saving* (line 7).

Once the safety margin hits zero (line 9), consistency_limit is re-estimated based on b_{Xmin}.count (the minimum count value in X) and n', the total number of items forwarded to *Space-Saving* (lines 10/16 and 17). If $b_{min} \in Y$ can no longer be guaranteed from statistics alone, lines 11–16 push one high-count bin from Y to X and one low-count bin from X to Y (as motivated before).

The setting of consistency_limit can be motivated as follows. Let us assume we have forwarded n' items to *Space-Saving*. Let $|Y|$ denote the size of Y. Then, the following guarantee on the bin with minimum count in Y holds:

$$b_{Ymin}.\text{count} \le \left\lfloor \frac{n'}{|Y|} \right\rfloor \quad . \tag{1}$$

That is, the largest minimum count that is possible in Y can not be larger than this value (Lemma 3.3 in [15] identifies a similar bound for b_{min}.count in *Space-Saving*).

Observe that the value of $\lfloor n'/|Y| \rfloor$ changes only every $|Y|$ forwarded items. A well-engineered implementation[2] of *Filter* thus needs to perform consistency maintenance at most once every $|Y|$ forwarded items.

3.2 Hardware-Conscious Filtering

The complexity of the filtering stage (lines 2–4 in Figure 5) lies in the lookup of items x in the bin set X. A hash table was suggested for this task in *Space-Saving*, since it offers (approximate) $|X|$-independent lookup time. This asymptotic value becomes irrelevant, however, for the small bin counts $|X|$ we are considering here.

Thus, we can gain speed by implementing the lookup in line 2 as a *sequential scan* over X. Such a scan fits the pipelined execution model of modern processors, avoids random access and pointer chasing as it happens for a hash-based lookup.

Vectorized Search. Sequential scans can efficiently be implemented with *vectorized instructions*. By using the SIMD instruction sets of modern processors, four to eight[3] bin contents can be inspected in a single CPU cycle. As our evaluation in Section 6 shows, the use of SIMD dramatically improves the filtering throughput, particularly for real-world skew values.

The SIMD code that we use in our implementation is illustrated in Figure 6 for $|X| = 8$ and a SIMD width of 128 bits (*i.e.*, four items per SIMD register). The contents of X are held in four SIMD registers, two of which hold the eight item values (`itms.vec[0]` and `itms.vec[1]`) and two of which maintain the associated count values (`cnts.vec[0]` and `cnts.vec[1]`).

To prepare the vectorized search, the input item `x` is replicated into the four slots of the SIMD register `vec_x` (line 2). Comparison with the two item vectors (lines 4–5) yields vectors `cmp1` and `cmp2` that contain a '−1' where the item was found and '0' elsewhere. This means that count vectors can be updated by *subtracting* `cmp1` and `cmp2`, as done in lines 7 and 8. Finally, the comparison results are combined into a

---

[2]The value of b_{Xmin}.count can be cached and used as a lower bound before actually executing lines 10–16.
[3]The AVX instruction set of Intel's *Sandy Bridge* architecture offers a SIMD width of 256 bits. In previous processor generations, this number was 128.

```
1  // load the input item into vector register
2  const __m128i vec_x = _mm_set1_epi32(x);

3  // compare it to the bin contents
4  __m128i cmp1 = _mm_cmpeq_epi32(itms.vec[0],vec_x);
5  __m128i cmp2 = _mm_cmpeq_epi32(itms.vec[1],vec_x);

6  // update the count values
7  cnts.vec[0] = _mm_sub_epi32(cnts.vec[0],cmp1);
8  cnts.vec[1] = _mm_sub_epi32(cnts.vec[1],cmp2);

9  // compute the overall result of the comparisons
10 cmp1 = _mm_or_si128(cmp2, cmp1);

11 // bring the result into the scalar part
12 bool found = _mm_movemask_ps((__m128) cmp1);
```

Figure 6: Vectorized filtering using x86 intrinsics.
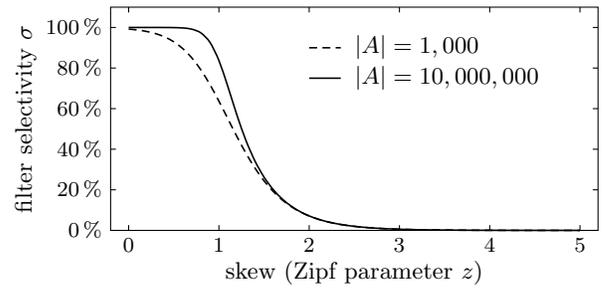

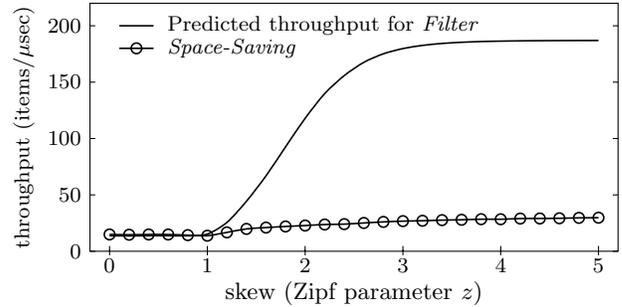**Figure 7: Filter selectivity ($|X| = 8$), depending on Zipf parameter z and alphabet size $|A|$.**

Figure 8: Predicted throughput for *Filter* assuming $k = 1000$, $|X| = 8$, *lookup cost in* $X = 10$ cycles, and an alphabet size $|A| = 5000000$.

single Boolean value (using a SIMD `or` and a `movemask` operation) to decide whether or not the item should be passed on to *Space-Saving*.

Observe that this code compares and updates bin contents eagerly. While this may lead to redundant instruction executions without effect (*e.g.*, we will often subtract '0' from the count registers), the absence of *branches* makes the code particularly efficient on actual hardware. We will evaluate the effectiveness of vectorization in Section 6.

3.3 *Filter* Characteristics

To understand the runtime characteristics of *Filter*, its per-item cost can be broken up into three terms:

$$lookup\ cost\ in\ X \tag{2}$$
$$+\ \sigma \times cost\ of\ Space\text{-}Saving\ on\ Y \tag{3}$$
$$+\ \sigma' \times cost\ of\ consistency\ maintenance\ , \tag{4}$$

where σ and σ' are the probabilities that the code branches in lines 6–17 and lines 10–16 are entered for the input item (respectively).

Terms (2) and (4) are overhead that is not present in the plain *Space-Saving* algorithm. In return, the cost of *Space-Saving* is reduced by the factor σ. Vectorized execution will make the lookup cost in X very small. The code in Figure 6, *e.g.*, requires only seven assembly instructions to perform the filtering work.

Entry into the consistency maintenance code path might result in a noticeable cost, since it involves accesses to X and Y that are not supported by any data structure (*e.g.*, line 13 in Algorithm *Filter* is essentially an insertion step in a sorted list). But the factor σ' is very small in practice. To illustrate, if all bins in Y contain the same `count` value (which is the worst case that defines the bound in (1)), $|Y|$ items can be forwarded to *Space-Saving* before the minimum `count` value increases. In practice, we find that $\sigma' \ll \sigma \times \frac{1}{|Y|}$.

This indicates that the filter selectivity σ is the dominant factor in determining the benefit from filtering.

For Zipf-distributed data, σ can be described in closed form. The probability of finding an item x among the k most frequent elements of a Zipf distribution is

$$P(k, z, |A|) = \sum_{i=1}^{k} \frac{1}{i^z} \times \frac{1}{\sum_{j=1}^{|A|} \frac{1}{j^z}}\ . \tag{5}$$

That is, we sum up the first k probabilities of a Zipf distribution and normalize using the sum of all item probabilities.

The probability that an item passes the filter is thus

$$\sigma(|X|, z, |A|) = 1 - P(|X|, z, |A|)\ .$$

Assuming a filter stage with eight bins ($|X| = 8$, as reflected also in Figure 6), the resulting filter selectivity is illustrated in Figure 7. As can be seen in the figure, the effectiveness of filtering increases sharply as soon as the input data skew exceeds a Zipf parameter of around 1. For instance, for a skew of $z = 1$ and an alphabet size $|A| = 1000$, filtering already avoids more than 35 % of the *Space-Saving* work. Since real-world data sets exhibit Zipf values much larger than 1 ([14] provides a detailed study), we can expect a very high effectiveness of filtering.

3.4 Overall Cost

We are now ready to model the overall cost of an execution of *Filter*. Figure 6 lists the actual SIMD code that implements the lookup in X. With a loop body added, this code will require around 10 cycles to execute for each input item. Since we know σ and the cost of *Space-Saving* (from experiments, cf. Table 1), we can compute cost terms (2) and (3) and predict the throughput of *Filter* depending on data skew.

Such a prediction is illustrated in Figure 8. In this figure, we complemented experimental results for *Space-Saving* with a prediction on the throughput of *Filter*. For real-world skews (starting from around 1.5), we can expect several factors of throughput improvement. Filtering causes a slight cost increase for very low skew values ($z \lesssim 0.5$). We will verify and evaluate both characteristics in Section 6.

```
1  foreach stream item x ∈ S do
2  │  look-up bin b_x in X with b_x.item = x ;
3  │  if such a bin was found then
4  │  │  └ b_x.count ← b_x.count + 1 ;
5  │  else
6  │  │  send x to processor P_1 ;
7  │  │  decrement consistency_limit ;
8  │  │  increment n' ;
9  │  │  if consistency_limit = 0 then
10 │  │  │  b_{Xmin} ← bin in X with min. count value ;
11 │  │  │  if b_{Xmin}.count ≤ n'/|Y| then
   │  │  │  │  // actual violation of consistency limit
12 │  │  │  │  send b_{Xmin} to processor P_1 ;
13 │  │  │  │  receive b_{Ymax} from processor P_1 ;
14 │  │  │  │  n' = n' + b_{Xmin}.count − b_{Ymax}.count ;
15 │  │  │  │  b_{Xmin}.item ← b_{Ymax}.item ;
16 │  │  │  │  b_{Xmin}.count ← b_{Ymax}.count ;
17 │  │  │  │  b_{Xmin} ← bin in X with min. count value ;
18 │  │  │  consistency_limit = b_{Xmin}.count − n'/|Y| ;
```

Figure 9: Algorithm *Parallel-Filter* for Processor P_0.

```
1  foreach msg received from processor P_0 do
2  │  if msg = bin b_{Xmin} then
3  │  │  b_{Ymax} ← bin with max. count value ;
4  │  │  send b_{Ymax} back to processor P_0 ;
5  │  │  add b_{Xmin} to the bin set Y;
6  │  else if msg = stream item x then
7  │  │  └ run regular Space-Saving ;
```

Figure 10: Alg. *Parallel-Filter* for Processor P_1.

The above model does not include the cost of consistency maintenance (cost term (4)). In practice, it will have only marginal impact. This is because the operation is necessary at most once every $|Y|$ forwarded items, *i.e.*, with a probability of at most $\sigma \times {}^1/_{|Y|}$. If, say, we choose Y to contain 1,000 bins, this means that a consistency maintenance run would have to be at least a thousand times more expensive than *Space-Saving* to have noticeable impact.

4. PARALLELIZATION OF FILTER

The key benefit of Algorithm *Filter* comes from reducing the amount of data reaching the *Space-Saving* routine. On modern multi-core machines, this effect can be amplified by dividing the filtering and *Space-Saving* work over two or more processor cores. The structure of *Filter* describes a *data flow system* (cf. Figure 1), which can be mapped to *pipeline parallelism* on multi-core hardware.

4.1 *Parallel-Filter*

Intuitively, the task of input filtering is associated with one (possibly also more; see below) CPU core P_0, while a different core P_1 runs the *Space-Saving* part. The two cores interact through *asynchronous message channels* installed between the threads on P_0 and P_1.

The algorithm *Parallel-Filter* is illustrated in Figure 9 and 10. Essentially, all interaction with the bin set Y (*i.e.*, the *Space-Saving* part) is replaced by explicit *messaging instructions*, as shown in lines 6, 12, and 13.

The other end of these messages is a modified *Space-Saving* instance that runs on P_1. We listed its pseudo code in Figure 10. Input to this code are the messages received from P_0. The message is then dispatched either to run a regular *Space-Saving* update or perform bin pushing as discussed earlier in Section 3.1.

The most apparent advantage of *Parallel-Filter* is that operations on X and Y can now be executed in parallel. In particular, cost terms (2) and (3) in Section 3.3 now arise

on separate cores, such that only the larger term determines the overall runtime.

This is particularly valuable if both of the terms are non-negligible. That is, a pipeline parallel execution could offer performance advantages especially for mid-range skew values $1 \lesssim z \lesssim 2$, where both algorithm parts take a considerable share of the input data (cf. Figure 7). This skew range matches what can be found in many real-world use cases.

***Parallel-Filter* and Modern Hardware.** Partitioning *Filter* over independent CPU cores also leads to routines with an extremely small code and data footprint. Most importantly, the filter task is simple enough to keep all bin contents within the (SIMD) registers of P_0. Such locality is an important ingredient to achieve high cycles-per-instruction (CPI) values on modern hardware.

Message Passing. Observe in Figure 9 that the bulk of the messages between P_0 and P_1 is sent fully asynchronously. Asynchronous message passing of this type is known to be a good fit for current hardware. In fact, hardware makers are developing new hardware architectures entirely based on message passing [9] and the operating systems community is working on full OS re-designs built around message passing primitives [2]. In our implementation, we adopt the message passing protocol of [2].

More Than Two Cores. Intuitively, the idea of *Parallel-Filter* can be extended in a pipelined mode, where multiple cores are running the filtering (cf. Figure 1). The performance advantage of such an approach will heavily depend on the data distribution, since each core has to take out enough share of the data to improve the overall throughput.

5. QUERIES

The key challenge in frequent item counting is to achieve high throughput, yet retain the ability to support *on-line queries* (*e.g.*, top-n queries), while the algorithm processes input. As analyzed in detail by Das *et al.* [6] and motivated briefly by us in Section 2.3, the cost of *result merging* may easily ruin any throughput advantages gained from parallelism or work distribution. For instance, Das *et al.* account more than 90 % of all CPU resources to query processing and result merging even for low query rates.

This problem does not arise in *Filter*, because all algorithm state is already partitioned by item values. All it takes to answer a top-n query is to return the bins in X and as many of the (count-sorted) bins from Y as needed to answer the query.

The advantages of *Filter* in the presence of on-line queries become even more pronounced when *Filter* is run in a parallel setting (cf. Section 4). The query can now be forwarded
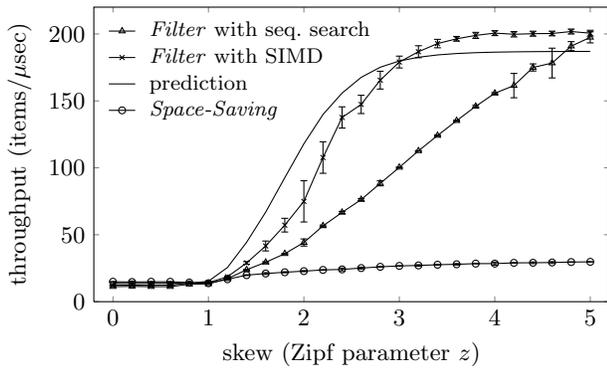
Figure 11: *Filter* performance, $k = 1000$, $|X| = 8$.

Figure 12: *Filter* vs. *Parallel-Filter* without querying, $k = 1000$, $|X| = 8$.

within the processing pipeline much like input items. Unlike in classical parallelization schemes, there is no need to suspend parallel processing because of an on-line query.

6. EVALUATION

6.1 Experimental Setup

To evaluate the potential of input filtering, we implemented the above algorithms using C (compiled with gcc version 4.5.2) on an Intel Nehalem CPU with a clock speed of 1.87 GHz (Intel model number L7555) and running Ubuntu 11.04. To ensure deterministic performance, we disabled the system's *hyper threading* and *turbo boost* features (which could lead to further throughput improvements in practice).

For evaluating the results, we generated synthetic data that follows a Zipf distribution. We ran the experiments on a skew range of 0.0 to 5.0. All the generated data sets contained 50 million items with an alphabet size of 5 million.

To keep the comparison meaningful, we did not implement the *Space-Saving* with linked list ourselves, instead we used the implementation from [5], which is publicly available. We recompiled that code using gcc and got similar results as presented in [5]. Therefore, we treat those results as the baseline for comparison of the performance.

6.2 Different Implementations of *Filter*

We implemented two variants of the *Filter* algorithm. One that uses a naïve sequential search and one based on the SIMD code shown in Figure 6. The performance of these two implementations along with the throughput of *Space-Saving* are shown in Figure 11. The figure shows the skew vs. throughput (items/μ-second) of the implementations for $k = 1000$, $|X| = 8$. We got similar graphs for $k = 100$ and 10000.

The results shown in the figure are consistent with the analytic assessment in Section 3.4 (repeated as a solid line in Figure 11). For very low skew (up to $z \lesssim 0.8$), *Filter* runs slightly slower than the baseline *Space-Saving* (up to $\approx 28\%$ loss for the sequential implementation, 2–17% for the SIMD code). In this low-skew range, the overhead of filtering and consistency maintenance cannot be compensated by a reduced workload on *Space-Saving*.

As soon as the skew goes higher than 0.8, *Filter* becomes much faster than *Space-Saving*, whereby the SIMD implementation quite closely follows our analytical model. For very high skew we even see better throughput as predicted,
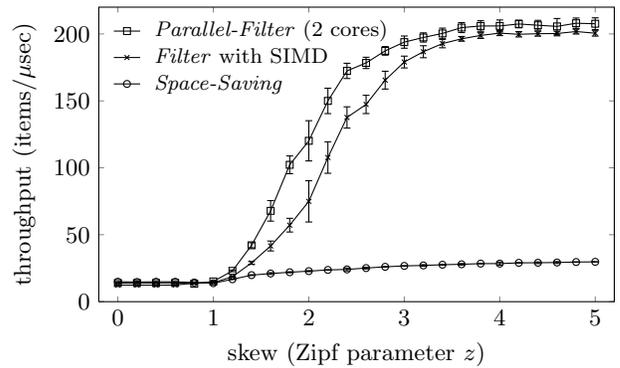
which indicates that we over-estimated the cycle count needed per item in the SIMD code.

For mid-range skew values ($1.5 \lesssim z \lesssim 2.5$), the measured throughput stays behind our model. This is mainly because our model ignored a statistical effect. When filtering takes away the $|X|$ most frequent items from the input stream, the remaining data set is actually *not* Zipf-distributed any more. Rather, the skew perceived by *Space-Saving* is appreciably smaller than the one seen at the input. This leads to the under-estimation of the *Space-Saving* cost in our model seen in Figure 11.

The implementation based on sequential search clearly does not follow our cost model. This is because we assumed the eager execution model of our SIMD code, which leads to data-independent runtimes (we had assumed 10 cycles/item). This assumption does not hold for a non-hardware-optimized search that aborts the sequential search lazily. Yet, note that also this naïve implementation shows significant throughput advantages over an off-the-shelf *Space-Saving* implementation.

6.3 *Filter* vs. *Parallel-Filter*

In this section, we explore how *Filter* and *Parallel-Filter* perform with respect to each other. We analyze the performance both in the presence and in the absence of queries. We used the SIMD implementations for comparing performance.

6.3.1 Without Querying

In *Parallel-Filter*, the *Filter* and *Space-Saving* processing overlap. For very low skew values, this reduces the overhead of filtering. Since item lookups in X are now performed on a separate core, only consistency maintenance and a small communication overhead add to the cost of the baseline *Space-Saving*. More specifically, *Parallel-Filter* runs only 5% slower than *Space-Saving* (cf. Figure 12).

As discussed already in Section 4, parallel execution becomes most effective if the contributions of terms (2) and (3) in our cost model are of comparable size, which is the case for mid-range skew values. Our throughput results in Figure 12 confirm this expectation. Skew ranges above 1.0 are most relevant in practice, and we see an improvement of *Parallel-Filter* over *Filter* of up to 80% in this range (and a factor of up to six compared to plain *Space-Saving*).
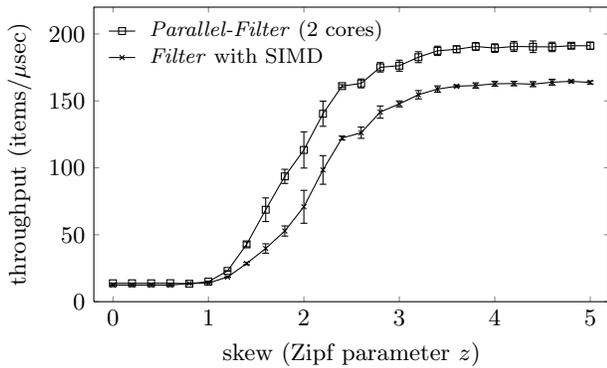
Figure 13: *Filter* vs. *Parallel-Filter* with querying, $k = 1000$, $|X| = 8$, query rate: **2000 queries/second.**

Figure 14: Impact of increasing sizes of X on *Parallel Filter* (2 cores), $k = 1000$.

For very high skew values, the advantage of parallel execution becomes less pronounced. In this range, the *Space-Saving* part of *Filter* is called very infrequently. Parallel execution of filtering and *Space-Saving* thus yields little benefit over the single-threaded case. Likewise, in additional experiments we found that parallelism beyond two cores will not significantly increase overall throughput any further.

6.3.2 With Querying

A key benefit that we expect from *Filter* is a better robustness against concurrent queries. To verify that property, we re-ran the experiments of Figure 12, but issued queries while the input was processed. To illustrate high robustness, we deliberately chose a very high query rate of 2,000 queries per second.

As shown in Figure 13, even such high query rates cause only little performance impact compared to the numbers we saw for input processing without queries (Figure 12). Most realistic workloads will use lower query rates, which means that on-line queries will cause negligible overhead in practice. This is in sharp contrast to the merging overhead reported by Das *et al.* [6], where on-line queries would immediately reduce input processing rates by several factors.

Also observe in Figure 13 how parallel processing is even more robust to on-line queries than our single-threaded implementation. This is because queries propagate from one CPU core to another. And while, *e.g.*, the last core in the pipeline collects bin contents from its *Stream-Summary* data structure, all other cores can already proceed with input processing.

6.4 Impact of the size of X

To see the impact of different sizes of X, we analyzed the performance of *Parallel-Filter* with $|X| = 4$, 8, and 16 (Figure 14). As expected, for a very low skew, they perform similarly because the added overhead by 4, 8 or 16 bins is too small in comparison to the *Space-Saving* cost. But for a very high skew, the performance will drop due to the increase in constant-time look up cost of X. As shown in the figure, the performance with 4 bins supersedes that with 8 and 16 bins, and reaches up to 250 million items per second. On the contrary, for a skew range of 1.0 to 1.6, $|X| = 16$ bins perform best, since it reduces the number of items that get forwarded to *Space-Saving*. Overall, $|X| = 8$ is an intermediate value that offers good performance balance for all skew ranges.
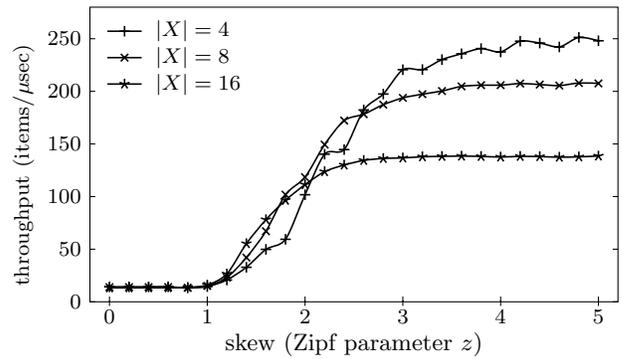
7. RELATED WORK

To leverage the advances in hardware technology, it has become unavoidable to re-design software systems such that they match the strengths of the underlying hardware. As Kim *et al.* [10] have shown, the changing hardware characteristics may even shift the balance between algorithmic approaches. More specifically, they conclude that sort-based join methods will soon overtake hash-based alternatives because sorting can better exploit modern instruction sets, most notably SIMD extensions.

The use of SIMD extensions proved effective also in our work, but only after we designed an algorithmic structure that would permit the use of SIMD. The necessity of careful algorithm designs to enable SIMD, but also the benefits that can be gained, were shown previously for a number of database tasks, including in-memory decompression [23] and sorted set intersection [18]. As in our case, the performance advantages obtained far exceed what would be expected from a classical cost analysis alone.

A number of research groups have re-designed database algorithms to better match the characteristics of caches and main-memory subsystems in modern hardware. Shatdal *et al.* [19] proposed adaptations to database join algorithms as early as 1994. Later their results were extended to match the cache characteristics of modern hardware, most notably in the MonetDB project [13]. More recently, Zukowski *et al.* [25] showed how lightweight data compression can better utilize real-world memory subsystems.

Probably most related to the frequent item counting problem is a recent paper by Ye *et al.* [24] that studies the performance of aggregation algorithms on modern hardware. Technically, the full aggregate of a data set contains all information needed to determine the most frequent item. But large memory requirements and necessary post-processing render the approach impractical for frequent item counting. Aside from that, Ye *et al.* achieve comparable throughput for the alphabet sizes we considered here, but require 8-way parallelism and much stronger CPUs to do so.

The resulting algorithm structure of our work resembles a *data flow system*, and there are numerous examples of how data flow-oriented designs can be mapped very efficiently to a given piece of hardware. In fact, most hardware designers first analyze the data flow of any given problem before they try to solve it.

A seminal work in stressing the importance of data flow-driven designs were the *systolic arrays* of Kung *et al.* [11, 12]. In the database world, one of the most notable studies has been done by Teeuw and Blanken [20]. Many of these results carry over to modern multi-core environments in the form of *pipeline parallelism*, as was emphasized recently, *e.g.*, by Giacomoni *et al.* [8].

Data flow-oriented processing proved effective also in some of our own recent work. In [21], we showed how it helps to parallelize the processing of *sliding-window joins* in many-core systems; similarly, we used a data flow-based design to solve the frequent item problem in (FPGA) hardware [22].

8. CONCLUSIONS

We have demonstrated the effectiveness of *pre-filtering* on modern hardware when applied to the *frequent item problem.* Filtering opens the door to using vectorized instructions on modern processor architectures. Depending on the input data skew, our proposed algorithm *Filter* improves performance over *Space-Saving* (the existing state of the art) by one order of magnitude.

We also show that *Filter* can easily be parallelized and thus make use of additional computing resources in multi-core systems. The proposed *Parallel-Filter* algorithm improves the performance of *Filter* significantly on the data with medium skew—the most common case in real-world applications. Both algorithms, *Filter* and *Parallel-Filter*, sustain their throughput properties even under high query load, since they avoid the merging overhead of earlier parallel frequent item solutions.

The improvements due to filtering are orthogonal to possible advances in the back-end algorithm (*Space-Saving* in our case). In fact, since filtering turns data skew into a throughput advantage, it could also complement other algorithms where skew often causes major performance problems.

Acknowledgements

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, 1994.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM SOSP*, Big Sky, MT, USA, 2009.

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proc. ACM SIGMOD*, 2011.

[4] M. Cafaro and P. Tempesta. Finding frequent items in parallel. *Concurrency and Computation: Practice and Experience*, 2011. (online preprint).

[5] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endowment (PVLDB)*, 1(2), 2008.

[6] S. Das, S. Antony, D. Agrawal, and A. E. Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proc. VLDB Endowment (PVLDB)*, 2(1), 2009.

[7] P. Dubey. Teraflops for the masses: Killer apps of tomorrow. In *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[8] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proc. ACM SIGPLAN*, Salt Lake City, UT, USA, 2008.

[9] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *2010 IEEE Int'l Solid-State Circuits Conf.*, San Francisco, CA, USA, 2010.

[10] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endowment (PVLDB)*, 2(2), 2009.

[11] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, Knoxville, TN, USA, 1978.

[12] H. T. Kung and P. L. Lohman. Systolic (VLSI) arrays for relational database operations. In *Proc. ACM SIGMOD*, Santa Monica, CA, USA, 1980.

[13] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4), 2002.

[14] N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowl. Eng.*, 68(4), 2009.

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

[16] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proc. PACT*, 2009.

[17] X. Qiu, G. Fox, H. Yuan, S.-H. Bae, G. Chrysanthakopoulos, and H. Nielsen. Parallel data mining on multicore clusters. In *Proc. GCC*, Bloomington, IN, USA, 2008.

[18] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using SIMD instructions. In *ADMS Workshop*, Seattle, WA, USA, 2011.

[19] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proc. VLDB*, 1994.

[20] W. B. Teeuw and H. M. Blanken. Control versus data flow in parallel database machines. *IEEE TPDS*, 4(11), 1993.

[21] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proc. ACM SIGMOD*, Athens, Greece, 2011.

[22] J. Teubner, R. Mueller, and G. Alonso. FPGA acceleration for the frequent item problem. In *Proc. ICDE*, Long Beach, CA, USA, 2010.

[23] T. Willhalm, N. Popvici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endowment (PVLDB)*, 2(2), 2009.

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

[25] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, Atlanta, GA, USA, 2006.