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 operators. 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 operators, 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 of frequent item computation that does not suffer any of the overheads of existing parallel solutions when querying the results and offers significantly higher throughput.

1. INTRODUCTION

Modern hardware platforms are evolving at an increasing speed. In addition to the shift to multi-core architectures, even off-the-shelf machines have increasingly sophisticated parallelization techniques, from vector (SIMD) instructions to hardware threads, with additional hardware components being used more and more as additional processing units (*e.g.*, hardware encryption modules, GPUs, or FPGAs).

These developments result in considerable changes in the underlying assumptions made when designing data operators and data processing algorithms. For instance, recent work has revisited the trade-offs of processing joins using either hashing or sorting. While the accepted wisdom is that hashing performs better, it turns out that once SIMD instructions are wide enough, sorting is likely to be a better option [10]. Similarly, since hardware has become so efficient at hiding page faults and at prefetching data, it has been noted that hash partitioning might not be necessary when performing parallel joins [3].

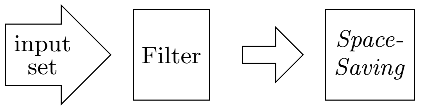


Figure 1: Input filtering for the frequent item problem. A filter absorbs much of the input data set and forwards only the remaining items to state-of-the-art *Space-Saving* instance.

In the same spirit, here we explore the processing of data in the presence of skew in the distribution. Skew is common in many real world data sets [6], but it is a problem in many algorithms where skew quickly leads to load imbalances. The two studies just mentioned [10, 3], for instance, report performance losses in the presence of skew. In these algorithms, like in many parallel implementations of common data operators, the parallelism available in the underlying hardware is exploited by partitioning the data and working in each partition in parallel. Afterwards, the results for each partition are merged. The main limitations to such an approach are that skew makes some partitions heavier than others and that the merging of results can itself be an expensive operation that often cannot be parallelized.

In this paper, we focus on the frequent item counting problem: given a data set, count the number of times each item appears in the set. 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]. Available sequential solutions are very efficient. For instance, *Space-Saving* reaches throughputs of around 20 million items per second [14].

Interestingly, it has been shown that the problem is difficult to parallelize using conventional data partitioning techniques [7]. On the one hand, frequent item counting is interesting when the data is really skewed (otherwise all items has similar counts and none has a higher frequency than others). On the other hand, processing parts of the input in parallel leads to an expensive merge operation when queries are posed. The overhead that this causes may dominate any gains obtained from the counting in parallel [7]. To date, there is no parallel implementation of frequent item counting that reaches the performance of the best sequential implementation of the algorithm.

The approach to processing skewed data we propose in this paper—and that we illustrate with the frequent item problem—is to take advantage of modern hardware characteristics not to speed up the underlying algorithm (*Space-Saving*, in this case) but to filter the data (Figure 1). In the presence of skew, the filtering can be done scanning a small number of registers and implemented using SIMD instructions. The corresponding code can be highly optimized for instruction and data locality (thereby fully benefiting from the existing instruction pipelines and speculative execution techniques); it does not involve any data traffic with the higher level caches (avoiding a common problem when parallelism is introduced that often requires clever memory arrangements [20]); and will only get faster as the width of SIMD instructions increases, since all the necessary comparisons for the data held in the filter are independent of each other and can be done in parallel.

Moreover, in the paper we show that our filtering approach can be easily parallelized using a small number of cores, reaching a throughput (200 million tuples per second for highly skewed data) that is close to one order of magnitude larger than the best published result. This performance is specially interesting because it can be maintained in the presence of concurrent queries while the counting is in process—an interesting property when operating on data streams.

The paper is organized as follows. In Section 2, we describe the frequent item problem, the state-of-the-art solutions available, and the difficulties in parallelizing these solutions. Section 3 introduces the basic filtering concept, which we build upon to leverage modern hardware characteristics in Section 4. We discuss the impact of queries in Section 5. We evaluate the performance of the filtering approach in Section 6 before reviewing related work in Section 7 and concluding in Section 8.

2. USE CASE

We show in this paper how the counting of *frequent items* can benefit from filtering. The problem is highly relevant in practice and compact to specify. Yet, to date, there is not convincing solution that can leverage the parallelism inherent in modern hardware.

2.1 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.* [14] (or variants

```
1 foreach stream item $x \in S$ do
2     find bin $b_x$ with $b_x$.item $= x$ ;
3     if such a bin was found then
4         $b_x$.count $\leftarrow b_x$.count $+ 1$ ;
5     else
6         $b_{\mathsf{min}} \leftarrow$ bin with minimum count value ;
7         $b_{\mathsf{min}}$.count $\leftarrow b_{\mathsf{min}}$.count $+ 1$ ;
8         $b_{\mathsf{min}}$.item $\leftarrow x$ ;
```

Figure 2: Algorithm *Space-Saving*. A fixed number of bins monitors the most frequent items in the stream S [14].

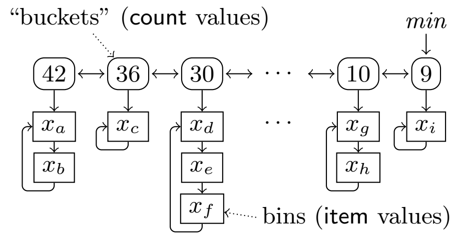


Figure 3: *Stream-Summary* data structure [14]. Bins with the same count value are grouped under one "bucket." Buckets form a double-linked list.

thereof) generally seen as the most efficient solution to the problem.

Observe that frequent item counting has many similarities with other database tasks, such as grouping/aggregation or star joins. In the following, we look at *Space-Saving*, the state-of-the-art solution to the frequent item problem, in more detail and illustrate why existing approaches are not suited for execution on modern hardware.

2.2 State of the Art: *Space-Saving*

The basic idea behind *Space-Saving* is to limit space consumption by counting ("monitoring" in [14]) only the occurrences of those items that are actually frequent. The algorithm allocates a configurable number k of $\langle \mathsf{item}, \mathsf{count} \rangle$ 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 item x read from the input 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 $\langle x, b_{\mathsf{min}}.\mathsf{count}{+}1 \rangle$. 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 [14, 5].

Implementation-wise, *Space-Saving* is typically backed up by two data structures. A *linked-list* variant (termed *Stream-Summary* in [14]; see Figure 3) keeps all bins ordered by their count values to make line 7 in Algorithm *Space-Saving* fast. Bin lookups by item value (line 2) are performed us-

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; $k = 1,000$.

ing 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.

2.3 *Space-Saving* Characteristics

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

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.

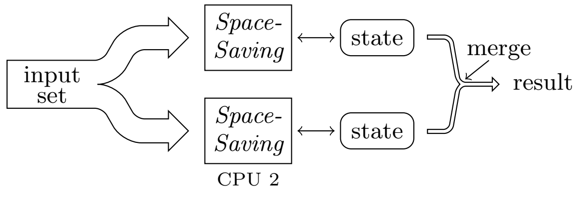
The individual iterations of *Space-Saving* update the item and *and* count fields. The resulting *data dependency* makes it difficult to apply local optimizations, such as *loop unrolling*, *vectorization*, or fine-grained *data parallelism* to *Space-Saving*. Without such optimizations, implementations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

2.4 It is Hard to Parallelize *Space-Saving*

The prevalence of multi-core hardware has spurred interest in parallel variants of *Space-Saving* [4, 7]. But it turns out the problem is hard to parallelize, even though the single-core solution is well understood and very efficient.

2.4.1 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 [7]) 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.

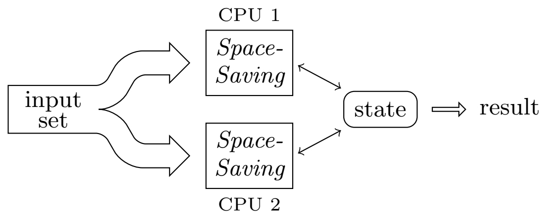
Result merging is a highly expensive operation that has to be performed each time an on-line query arrives. As Das *et al.* [7] have shown, even for low query rates merging costs quickly exceed any potential performance gain due to parallelization. The result of Das *et al.* is not really a surprise: merging has to be performed sequentially and the cost of merging increases with the number of parallel instances.

Unfortunately, result merging cannot be avoided in partitioning-based schemes. Input sets to frequent item counting are usually heavily skewed. 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.

2.4.2 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 would end 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. The cost of this can be significant: Molka *et al.* [15] 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* 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

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

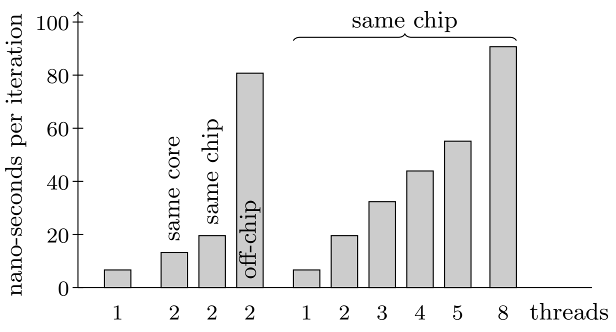


Figure 4: Cache coherency cost, depending on physical thread placement and degree of parallelism.

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 it was proposed by [7]).

In summary, neither partitioning nor sharing can adequately leverage the capabilities of current hardware. In fact, the current trends toward de-centralization and 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 illustrated 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 within X, we forward it to a *Space-Saving* instance that maintains the bin set Y.

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

```
1  foreach stream item $x \in S$ 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 $\leftarrow b_x$.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             $b_{X\mathsf{min}} \leftarrow$ bin in $X$ with min. count value ;
11             if $b_{X\mathsf{min}}$.count $\leq \frac{n'}{|Y|}$ then
                   // actual violation of consistency limit
12                 $b_{Y\mathsf{max}} \leftarrow$ bin in $Y$ with max. count value ;
13                 remove $b_{X\mathsf{min}}$ from $X$ and add it to $Y$ ;
14                 remove $b_{Y\mathsf{max}}$ from $Y$ and add it to $X$ ;
15                 $n' \leftarrow n' + b_{X\mathsf{min}}$.count $- b_{Y\mathsf{max}}$.count ;
16                 $b_{X\mathsf{min}} \leftarrow$ bin in $X$ with min. count value ;
17             consistency_limit $\leftarrow b_{X\mathsf{min}}$.count $- \frac{n'}{|Y|}$ ;
```

Figure 5: Algorithm *Filter*.

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. Otherwise, the forwarded item might overwrite a bin whose count value is not minimal.

The rationale behind filtering is that the corresponding code path (lines 2–4) executes more efficiently. Data structures for X that support this will increase the cost of count-based lookups in X. Searching X for b_{min} would thus be prohibitively expensive if done for every input item that fails the item search in Y. This is all the more true since we want to keep the heavy hitters in X, which makes it unlikely that b_{min} could even be found in X.

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 $b_{\mathsf{min}} \in Y$. 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 $b_{X\mathsf{min}}$.count (the minimum count value in X) and n', the total number of items forwarded to *Space-Saving* (lines 10/16 and 17). If $b_{\mathsf{min}} \in Y$ 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 fol-

```
 1 //load the input item into vector register
 2 const __m128i vec_x = _mm_set1_epi32(x);
 3
 4 //compare it to the bin contents
 5 __m128i cmp1 = _mm_cmpeq_epi32(itms.vec[0], vec_x);
 6 __m128i cmp2 = _mm_cmpeq_epi32(itms.vec[1], vec_x);
 7
 8 //update the count values
 9 cnts.vec[0] = _mm_sub_epi32(cnts.vec[0],cmp1);
10 cnts.vec[1] = _mm_sub_epi32(cnts.vec[1],cmp2);
11
12 //compute the overall result of the comparisons
13 cmp1 = _mm_or_si128(cmp2, cmp1);
14
15 //bring the result into  the scalar part
16 bool found = _mm_movemask_ps((__m128) cmp1);
```

Figure 6: Vectorized filtering using x86 intrinsics.

lows. Let us assume we have forwarded n' items to *Space-Saving*. Also, we know the size of Y is $|Y|$. Then, the following guarantee on the bin with minimum count in Y holds,

$$b_{Y\mathsf{min}}.\mathsf{count} \leq \frac{n'}{|Y|} \quad . \tag{1}$$

That is, the largest minimum count that is possible in Y can not be larger than this value (Lemma 3.3 in [14] identifies a similar bound for $b_{\mathsf{min}}.\mathsf{count}$ in *Space-Saving*). This comes from the fact that, if n' items are forwarded to *Space-Saving*, then the highest minimum count that is possible in Y is when every bin has the same count value. If one bin has higher count than $\frac{n'}{|Y|}$, then there must be another bin with a lower count than this. At any point, if we know the minimum count in X, i.e., $b_{X\mathsf{min}}.\mathsf{count}$, then we can forward ($b_{X\mathsf{min}}.\mathsf{count} - \frac{n'}{|Y|}$) number of items to *Space-Saving* instance without violating the overall consistency of the algorithm.

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[2] 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 5–6) 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 9 and 10. Finally, the comparison results are combined into a 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:

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

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 results in a noticeable cost, since it involves lookups an updates in X and Y that are not supported by any data structure (*e.g.*, line 13 in Algorithm *Filter* is essentially an insertion sort step). 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 $\sigma' \ll {}^{1}/_{|Y|}$.

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

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

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

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

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

[2] 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.

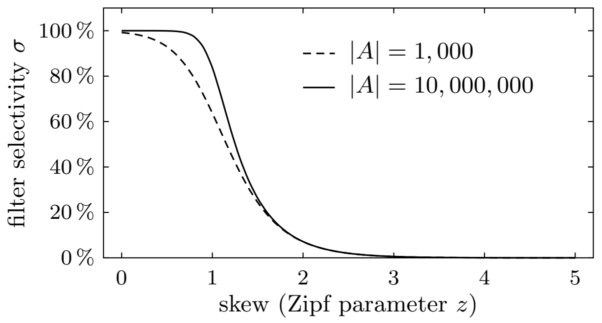


Figure 7: Filter selectivity ($|X| = 8$), depending on Zipf parameter z and alphabet size $|A|$.

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.

Filtering becomes ineffective if the input data is not skewed. For $z = 0$, almost all input items will pass the filter, *i.e.*, $\sigma \approx 1$, which means that the lookup cost in X is just additional overhead on top of *Space-Saving.* In Section 6, we determine this maximum overhead to be around 17 % with a SIMD-optimized implementation of *Filter.*

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 8. 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 9. 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, we expect a pipeline parallel execution to be most beneficial 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

```
1  foreach stream item $x \in S$ 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 $\leftarrow b_x$.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             $b_{X\mathsf{min}} \leftarrow$ bin in $X$ with min. count value ;
11             if $b_{X\mathsf{min}}$.count $\leq \frac{n'}{|Y|}$ then
                   // actual violation of consistency limit
12                 send $b_{X\mathsf{min}}$ to processor $P_1$ ;
13                 receive $b_{Y\mathsf{max}}$ from processor $P_1$ ;
14                 $n' = n' + b_{X\mathsf{min}}$.count $- b_{Y\mathsf{max}}$.count ;
15                 $b_{X\mathsf{min}}$.item $\leftarrow b_{Y\mathsf{max}}$.item ;
16                 $b_{X\mathsf{min}}$.count $\leftarrow b_{Y\mathsf{max}}$.count ;
17                 $b_{X\mathsf{min}} \leftarrow$ bin in $X$ with min. count value ;
18             consistency_limit $= b_{X\mathsf{min}}$.count $- \frac{n'}{|Y|}$ ;
```

Figure 8: Algorithm *Parallel-Filter* for Processor P_0.

```
1  foreach msg received from processor $P_0$ do
2      if msg = bin $b_{X\mathsf{min}}$ then
3          $b_{Y\mathsf{max}} \leftarrow$ bin with max. count value ;
4          send $b_{Y\mathsf{max}}$ back to processor $P_0$ ;
5          add $b_{X\mathsf{min}}$ to the bin set $Y$;
6      else if msg = stream item $x$ then
7          find bin $b_x$ with $b_x$.item $= x$ ;
8          if such a bin was found then
9              $b_x$.count $\leftarrow b_x$.count $+ 1$ ;
10         else
11             $b_{\mathsf{min}} \leftarrow$ bin with min. count value ;
12             $b_{\mathsf{min}}$.count $\leftarrow b_{\mathsf{min}}$.count $+ 1$ ;
13             $b_{\mathsf{min}}$.item $\leftarrow x$ ;
```

Figure 9: Algorithm *Parallel-Filter* for processor P_1.

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. Only accesses to in- and output data require access to memory.

Except for explicit messages between the threads, all remaining data accesses stay entirely local to the respective core. The internal state of *Space-Saving* is usually small enough to stay in the L1 cache of P_1. Such locality is an important ingredient to achieve high cycles-per-instruction (CPI) values on modern hardware.

Message Passing. Observe in Figure 8 that the bulk of the messages between P_0 and P_1 is sent fully asynchronously. P_0 explicitly waits for a reply only when the consistency guarantees have to be re-established between cores (lines 12–13). As we discussed earlier, the chances that this happens, σ', are low.

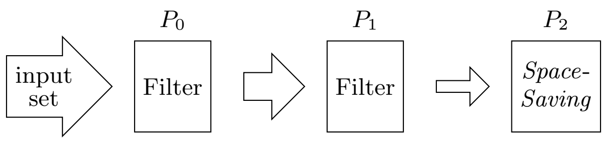


Figure 10: *Parallel-Filter* on three cores.

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].

More Than Two Cores. This idea can easily be extended to multiple cores running the filtering. Let us assume that there are n processors $P_0, P_1, ..., P_{n-1}$, arranged in a pipelined fashion (cf. Figure 10). Each processor P_i where $i < n-1$ and $i \neq 0$, can send and receive messages from processor P_{i-1} and processor P_{i+1}. Processor P_0 is connected to P_1 and P_{n-1} is connected to P_{n-2}. Each of the processors from P_0 to P_{n-2} monitors a small disjoint subset of the bins. However, the processor P_{n-1} monitors a larger disjoint set of bins by running an instance of *Space-Saving*.

In such a setting, processor P_0 runs the algorithm presented in Figure 8 and processor P_{n-1} runs the algorithm presented in Figure 9. Any other processor P_i receives messages from processor P_{i-1} and if it contains an input item x, it behaves similarly as processor P_0. On the other hand, if it receives the bin content, then it behaves similarly as processor P_1 shown in line 2–5 in Figure 9.

5. QUERIES

Both algorithms, *Filter* and *Parallel-Filter*, can support queries very efficiently as there is no result merging overhead. An additional query thread will issue the queries. In case of *Filter*, once a query for the top most k items is received, the thread emits the content of X and then emits top k items from the *Stream-Summary* data structure to the query thread. Since the emitted contents from the *Stream-Summary* are already sorted, with the help of insertion sort the query thread can efficiently sort these results and answer the query.

The query answering process is even more efficient for the *Parallel-Filter* algorithm. Once the first processor in the pipeline P_0 receives the query, it emits the bin content to the query thread and sends a query request to the processor P_1. Similarly, when a processor P_i receives a query requests, it also emits the bin content and passes the query request to the next processor. The query request propagates until the last processor P_{n-1} which is running the *Space-Saving* instance. When the processor P_{n-1} receives the query message, it emits the top k items from the *Stream-Summary* data structure. The query thread then sorts all these results to produce the final top k items. This approach is extremely efficient because once the processor emits its content it can start processing the input items again, in contrast to the *Filter*, where for each query the processing thread must suspend processing items and answer the query by emitting the bin content of X and top most k items from the *Stream-Summary* data structure.

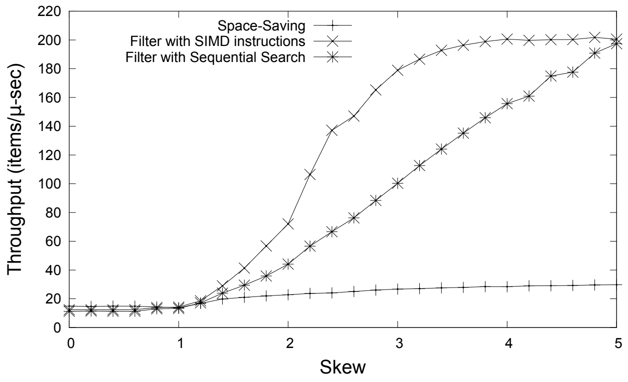


Figure 11: *Filter* performance, $k = 1000$, $|X| = 8$.

6. EVALUATION

6.1 Experimental Setup

We have implemented all the above algorithms in C. The code was compiled with both *gcc* (version 4.5.2) and Intel's *icc* (version 12.1.0). We used the O2 optimization option while compiling the code. With these two versions of the executable, we got slight quantitative differences, but no qualitative differences in the results. Therefore, here we present all the result on the code compiled using *gcc*. We used Ubuntu 11.04 for running all the experiments. We ran the experiments on an Intel machine with Nehalem CPU, Intel(R) Xeon(R) CPU L7555 with a clock speed of 1.87 GHz. The CPU has 24 MB L3 cache shared between 8 cores. We also disabled *hyper-threading* and *turbo-boost* while running the experiments.

For evaluating the results, we used data generated using *Zipf* distribution. We generated several different data sets with different seeds and ran the experiments on all of these data points. 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 have implemented two variants of the *Filter* algorithm. One with sequential search and another using SIMD instructions. 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 $k = 1000$. We got similar graphs for $k = 100$ and 10000. In this experiment, we fixed the size of X to 8.

As shown in the figure, for a skew of 0.0 to 0.8, both implementations of *Filter* run slower than *Space-Saving*. Depending on skew, the performance loss for the sequential scan is up to 28 %. For the SIMD version it is 2 % to 17 %. 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

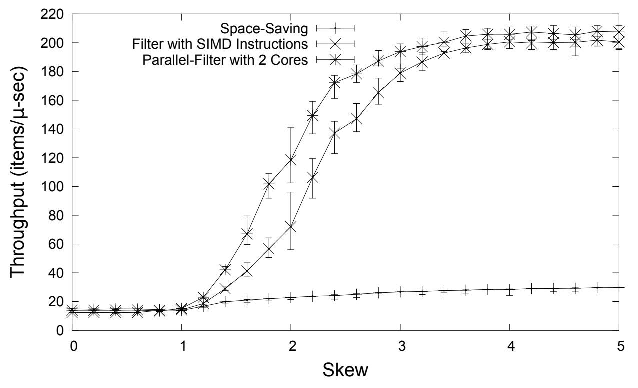


Figure 12: *Filter* vs *Parallel-Filter* without querying, $k = 1000$, $|X| = 8$.

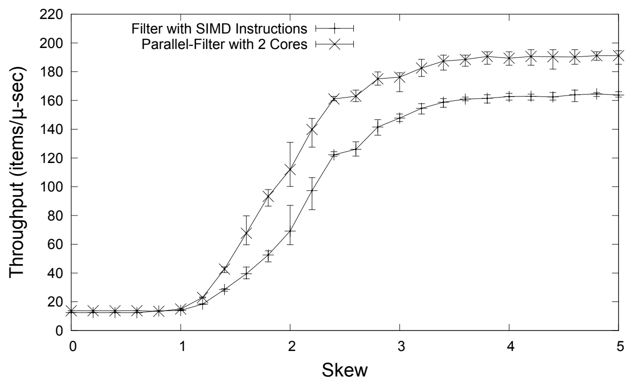


Figure 13: *Filter* vs *Parallel-Filter* with querying, $k = 1000$, $|X| = 8$, query rate: 2000 queries per second.

than 0.8, both implementations of *Filter* become much faster than *Space-Saving* by almost an order of magnitude. This is consistent with the analytic assessment in Section 3.3, where the filtering rate σ sharply changed as skew values exceeded values slightly below 1.

Filtering improves the overall throughput almost linearly with respect to the skew with a sequential search. Although the synthetic nature of the data plays some role in this case, the result clearly shows the impact of a small sequential search on modern hardware. Compilers are intelligent enough to unroll such a small loop and due to instruction pipeline, out of order execution, and improved branch prediction logic on modern processor this executes very fast.

With the SIMD implementation, we get further performance improvements due to the much faster processing times on the bin set X. For a skew higher than 3.6–3.8, we do not see much improvement; the throughput graph flattens out. This happens because, at such a high skew, only few items go to *Space-Saving* and with SIMD instructions a fixed number of comparisons are needed to find the corresponding bin.

All these results emphasize the fact that the filtering approach significantly improves the performance of frequent item.

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

Figure 12 shows the throughput (items/μ-second) of *Parallel-Filter*, *Filter* and *Space-Saving* with increasing skew. The result is shown for $k = 1000$ and $|X| = 8$. We also found similar graphs for $k = 100$ and 10000.

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*.

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 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*).

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.

6.3.2 With Querying

Existing parallel approaches to the frequent item problem suffer from very high overhead whenever queries have to be answered during input processing. Das *et al.* [7] report that the cost of *merging* of partial results in a data-partitioned execution of *Space-Saving* far exceeds the actual input processing cost even for low degrees of parallelism.

To show how *Filter* and *Parallel-Filter* react to online queries, we re-ran the experiments of Figure 12 but issued queries while the input was processed. We issued queries at a rate of 2,000 queries per second.

With queries applied, we observe the throughput rates shown in Figure 13. It is clear that both approaches can sustain a very high query rate without significant deterioration of the overall performance. In comparison to *Filter*, *Parallel-Filter* performs better in the presence of queries. For a very high skew of 5.0, the throughput drop for *Parallel-Filter* is only 8 % in comparison to the case of without queries (Figure 12).

In contrast, *Filter* looses performance in the range of 18 % in comparison to the no query scenario. Inevitably, producing output causes additional work. In *Filter*, this work has to be performed by a single core, thus directly adding to the algorithm's processing time. In a parallel execution, the core that runs *Space-Saving* is only lightly loaded. The overhead of traversing the *Stream-Summary* data structure to answer a query can be hidden by the overlapped execution of the filtering stage and *Space-Saving*.

6.4 Multi-Stage *Parallel-Filter*

In Section 4, we mentioned that multiple-cores can be arranged in a pipelined fashion to form a multi-stage *Parallel-*

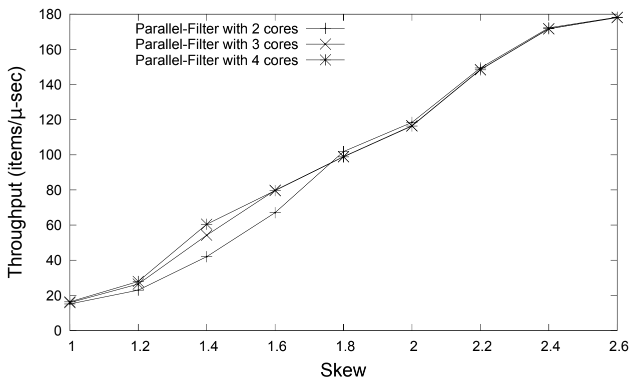


Figure 14: Impact of increasing cores on *Parallel Filter*, $k = 1000$, $|X| = 8$.

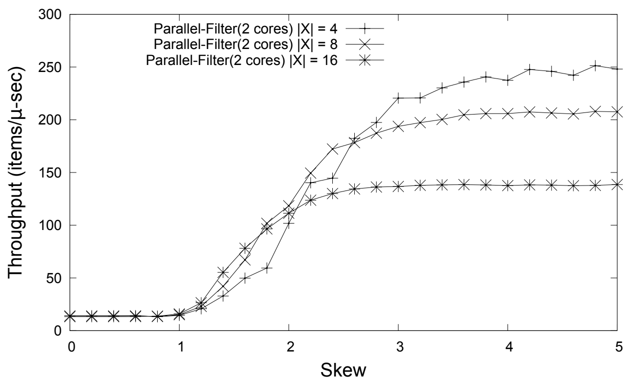


Figure 15: Impact of increasing sizes of X on *Parallel Filter*, $k = 1000$.

Filter, where multiple cores run the filtering step. In Figure 14, we show the performance of *Parallel-Filter* with an increasing number of cores. The graphs shows the throughput (items/μ-second) if the input skew is in the range $1.0 \leq z \leq 2.6$ for $k = 1000$ and $|X| = 8$. To improve the throughput with a multi-stage filter, subsequent processors doing the filtering need to get enough items. In case of a Zipf distribution, for a skew higher than 1.8, the additional processors running the filtering step do not receive enough input items to improve performance. For a skew lower than 0.8, the performance drops slightly with increasing cores due to additional consistency limit violations and the corresponding synchronizations. But for a skew range of 1.0 to 1.8, where all the filtering stages have some items to filter out, multi-stage filtering can in fact improve performance.

6.5 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 15). For a very low skew, they perform similarly. 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. 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 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, the authors of [10] 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 [19] and sorted set intersection [16]. 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.* [17] 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.* [21] showed how lightweight data compression can better utilize real-world memory subsystems.

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 [18]. 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].

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.

9. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of the 20th Int'l Conference on Very Large Data Bases (VLDB)*, pages 487–499, September 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. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, 2009.

[3] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proc. of the 2011 ACM SIGMOD Conference on Management of Data*, pages 37–48, 2011.

[4] Massimo Cafaro and Piergiulio Tempesta. Finding Frequent Items in Parallel. *Concurrency and Computation: Practice and Experience*, 2011. (online preprint).

[5] Graham Cormode and Marios Hadjieleftheriou. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1530–1541, 2008.

[6] Graham Cormode and S. Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SDM*, 2005.

[7] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams. *Proc. of the VLDB Endowment (PVLDB)*, 2(1), August 2009.

[8] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, February 2008.

[9] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *2010 IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, February 2010.

[10] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1378–1389, 2009.

[11] H. T. Kung and Charles E. Leiserson. Systolic Arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282, Knoxville, TN, USA, November 1978.

[12] H. T. Kung and Philip L. Lohman. Systolic (VLSI) Arrays for Relational Database Operations. In *Proc. of the 1980 ACM SIGMOD Conference on Management of Data*, Santa Monica, CA, USA, May 1980.

[13] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, April 2002.

[14] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems (TODS)*, 31(3):1095–1133, September 2006.

[15] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proc. of the 18th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009.

[16] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast Sorted-Set Intersection using SIMD Instructions. In *2nd Int'l Workshop on Accelerating Data Management Using Modern Processor and Storage Architectures (ADMS)*, Seattle, WA, USA, September 2011.

[17] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the 20th Int'l Conference on Very Large Data Bases (VLDB)*, pages 510–521, September 1994.

[18] Wouter B. Teeuw and Henk M. Blanken. Control Versus Data Flow in Parallel Database Machines. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 4(11):1265–1279, November 1993.

[19] Thomas Willhalm, Nicolae Popvici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast In-Memory Table Scan Using On-Chip Vector Processing Units. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2), 2009.

[20] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN Workshop*, pages 1–9, 2011.

[21] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. of the 22nd Int'l Conference on Data Engineering (ICDE)*, Atlanta, GA, USA, April 2006.