High-Throughput Subset Matching on Commodity GPU-Based Systems

Daniele Rogora
USI, Switzerland
daniele.rogora@usi.ch

Michele Papalini
Cisco Systems, France
michele.papalini@cisco.com

Koorosh Khazaei
USI, Switzerland
koorosh.khazaei@usi.ch

Alessandro Margara
Politecnico di Milano, Italy
alessandro.margara@polimi.it

Antonio Carzaniga
USI, Switzerland
antonio.carzaniga@usi.ch

Gianpaolo Cugola
Politecnico di Milano, Italy
gianpaolo.cugola@polimi.it

Abstract

Large-scale information processing often relies on subset matching for data classification and routing. Examples are publish/subscribe and stream processing systems, database systems, social media, and information-centric networking. For instance, an advanced Twitter-like messaging service where users might follow specific publishers as well as specific topics encoded as tag sets must join a stream of published messages with the users and their preferred tag sets so that the user tag set is a subset of the message tags.

Subset matching is an old but also notoriously difficult problem. We present TagMatch, a system that solves this problem by taking advantage of a hybrid CPU/GPU stream processing architecture. TagMatch targets large-scale applications with thousands of matching operations per seconds against hundreds of millions of tag sets. We evaluate TagMatch on an advanced message streaming application, with very positive results both in absolute terms and in comparison with existing systems. As a notable example, our experiments demonstrate that TagMatch running on a single, commodity machine with two GPUs can easily sustain the traffic throughput of Twitter even augmented with expressive tag-based selection.

CCS Concepts • **Information systems** → **Stream management**; *Data stream mining*

Keywords Subset matching, GPU-based processing, message selection and dissemination

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23–26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064190

1. Introduction

Subset matching is essential to global web applications. For example, within the Twitter messaging system, the first stage in ad selection for user queries finds a "match between user attributes and targeting criteria across the corpus of ads,"[1] which at a minimum amounts to checking that the attributes of the user query contain the targeting criteria of the ads. Even more important for Twitter is the selection of the messages themselves. Twitter allows users to "follow" a publisher for immediate delivery of published messages. In addition, Twitter provides a keyword or tag-based search over past and current messages. Combining these two features, a user might want to follow a publisher but only on a specific set of keywords or tags, or even just that set of tags without a specific publisher. This data selection based on subset matching is also essential in database [2, 9, 12] and data mining applications [3, 13, 18] where it is used as a selection or join operator, and in some publish-subscribe systems and some information-centric networking architectures [14] where it is used for message brokering and routing.

Perspective on an old problem. Basically, given a large corpus of sets $D = s_1, \ldots, s_n$ (the *database*) and a set q (a *query*) from a high-intensity input stream, subset matching means finding the sets s_i such that $s_i \subseteq q$. This is a basic combinatorial problem that is also notoriously difficult.

Roughly speaking, there are two kinds of solutions for subset matching. One checks sets s_i one by one against the query q. Typically, solutions of this kind use "signatures" $sig(s)$ that are compact representations of sets that admit to a fast comparison $sig(s_i) \subseteq sig(q)$ more or less exactly indicative of the relation $s_i \subseteq q$ between the sets [5, 12]. Another solution is to iterate over the elements of the query, $x \in q$, and to use an inverted index to find the list of sets s_i that contain x, and then to combine those lists to find which

[1] https://blog.twitter.com/2016/resilient-ad-serving-at-twitter-scale

sets are fully covered by q [7, 10]. This combination of lists can be seen as an exploration of the subsets of the query. In fact, a variant of this second solution looks for the subsets $q_j \subseteq q$ directly in the database (e.g., using a hash table).

Thus both types of algorithms reduce to an iteration over sets and neither one is ideal in all cases: one is a linear scan of the database; the other one iterates over the subsets $q_j \subseteq q$ and therefore is exponential in the size of the query q.

Approach and contribution. Given the long history of subset matching, we have little hope to achieve a high throughput with a purely algorithmic solution. We therefore take a multi-pronged approach whereby we leverage a hybrid stream processing system of CPUs and GPUs, combining techniques from state-of-the-art subset matching with novel algorithmic and technical improvements on both the CPU and GPU sides. We implement this approach in a subset matching engine called *TagMatch.*

	database size		
system	20M	40M	212M
GPU-only, plain	0.40	0.20	0.04
GPU-only, plain with batching	11.50	6.30	1.20
CPU-only, fast prefix tree	21.10	14.00	4.30
CPU-only, state-of-the-art ICN	27.60	17.40	—
CPU-only, TagMatch	3.90	3.40	0.68
TagMatch	268.80	144.40	35.30

(throughput: thousands queries per second)

Table 1: Summary evaluation: throughput of TagMatch vs. CPU-only and GPU-only systems.

Before introducing TagMatch, we emphasize that the synergistic use of CPUs and GPUs in TagMatch is essential, as demonstrated by the summary results of Table 1. Notice first that GPU parallelism alone is insufficient, and in fact it is inferior to state-of-the-art CPU-only solutions, over which TagMatch achieves an almost $10\times$ speedup. Notice further that TagMatch is also significantly faster than the best combination of CPU-only and GPU-only solutions, and that TagMatch itself achieves a $50\times$ speedup when running on a hybrid system. This is because TagMatch exploits the more versatile processing capability of CPUs *in combination with* the massively parallel processing capabilities of GPUs. TagMatch does that with an appropriate division of labor, specific algorithmic solutions, and an effective coordination between CPUs and GPUs. We further discuss an alternative, GPU-only architecture in Section 4.5.

To introduce TagMatch, we note that both types of existing algorithms—iterations over $s_i \in D$ or over $q_j \subseteq q$—can benefit from an index to take shortcuts. In the first case, if s_i is not a subset of q, then the iteration can skip all the supersets of s_i. Similarly, in the second case, if q_j is not in the database, then the iteration can skip all the subsets of q_j. So, both solutions can use an index in which sets are arranged according to their subset relations, and one such effective index is a prefix trie (or tree) like the one proposed originally by Rivest [19], which is in fact used in many subset-matching algorithms [7, 9, 15].

TagMatch takes a similar approach, although it indexes *signatures* rather than sets. In addition, TagMatch organizes the data and processing in a platform-specific way, with a coarse-grained index on the CPU side and a fine-grained selection on the GPU side. Specifically, TagMatch uses Bloom filters as set signatures, and partitions the database to divide and coordinate the work between CPUs and GPUs. Appropriately selected bit masks define the partitions and make up the CPU index, which is designed as a compact data structure to support a sequential but memory-efficient matching on CPUs. Correspondingly, signatures are grouped and sorted within partitions on GPUs to enable parallel processing and additional shortcuts. TagMatch then processes queries through a pipeline that alternates CPU and GPU stages. We design and engineer this pipeline so as to maximize the parallelism both between and within each stage.

We validate this design with an extensive experimental evaluation. We measure the performance of TagMatch in several relevant application scenarios both comparatively and in absolute terms. TagMatch outperforms all comparable systems we tested, namely a widely used database system (MongoDB), an existing message forwarding system, and a tightly optimized matcher based on a prefix tree that itself outperforms all subset matching algorithms we know of as reported in the literature. In absolute terms, as a highlight of our results, under a realistic Twitter workload with more than 212 million unique sets representing user preferences, TagMatch can process over 30,000 subset queries per second on a single, commodity machine with two GPUs.

In summary, we make the following contributions: (1) we develop a subset matching engine specifically designed for CPU/GPU systems, and (2) we demonstrate that a relatively inexpensive CPU/GPU system can support a messaging service capable of processing the entire traffic of Twitter[2] while providing a content-based filtering service that is significantly more expressive than that offered by Twitter today.

2. System Model

We design TagMatch as a general-purpose subset matching engine. To make this notion a bit more concrete, we consider sets of string *tags*, which is the most common use in applications and in any case provides a very general model. In essence, TagMatch implements a database of tag sets with a subset-match operation and the interface shown in Table 2.

The *add-set* and *remove-set* functions add and remove a set with an associated key, where the key is simply a link to application data. These changes are not immediately effective and instead are staged in a temporary index and become effective only after a call to the *consolidate* function.

[2] In 2015, Twitter registered an average rate of 6000 tweets per second.

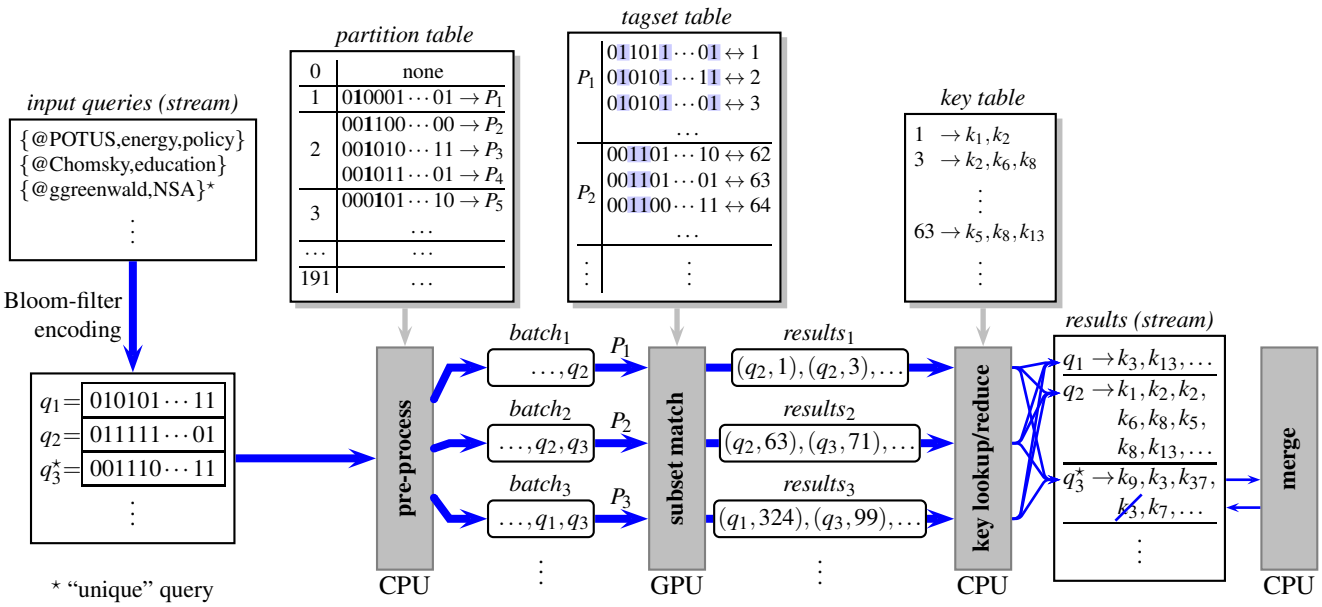


Figure 1: The architecture of TagMatch.

add-set(set, key) : *void*
remove-set(set, key) : *void*
consolidate$()$: *void*
match$(query\text{-}set)$: *multiset of keys*
match-unique$(query\text{-}set)$: *set of keys*

Table 2: TagMatch interface.

The *match* and *match-unique* find subsets of a given query set in the table. *match*(q) returns all the keys k associated with the indexed sets s such that $s \subseteq q$, possibly with multiple instances of the same key k if k is associated with multiple subsets of q. *match-unique* returns a *set* of keys k, such that at least one indexed set $s \subseteq q$ is associated with k (i.e., it avoids duplicate keys).

The *match-unique*(q) function is useful to implement the Twitter-like application discussed in Section 1. The application could store the preferences of users in a table $Users$ with two fields: $Users.prefs$ and $Users.id$. For each tweet in a stream $Tweets$, the application must find the ids of all the users interested in that tweet, effectively computing an inner join on $Users.prefs \subseteq Tweets.keywords$. Thus the application would use TagMatch to add each user preference u with *add-set*$(u.prefs, u.id)$, and then to find the matching users for each tweet t with *match-unique*$(t.keywords)$.

3. System Implementation

This section presents the general design of TagMatch, as well as the implementation details of its components. Figure 1 shows the high-level architecture of TagMatch, where the gray boxes represent the main computational steps of the *match* and *match-unique* functions while the white boxes represent the main data structures. TagMatch is built on a hybrid CPU/GPU system with one or more CPUs and one or more GPUs. Therefore, some computational steps run on CPUs while others run on GPUs.

At a high-level, TagMatch indexes partitions of related tag sets, and therefore finds the subset of an input query set in two steps: The first step finds the relevant partitions for a query set, and the second step matches the query against the individual sets within those relevant partitions.

Specifically, the matching algorithm consists of a four-stage pipeline: (i) The *pre-process* stage selects the relevant partitions for a query. (ii) The *subset match* stage finds the tag sets that match a query within each partition using a GPU. To maximize throughput and better use the processing capabilities of GPUs, this stage operates on batches of queries, evaluating them in parallel. (iii) The *key lookup/reduce* stage extracts the keys associated with each set of tags, and groups the results by query. (iv) Finally, the *merge* stage combines the results from multiple partitions into a single set of keys (for *match-unique*) or a multiset (for *match*).

TagMatch represents sets (database and query) as Bloom filters. Bloom filters are an ideal basis for subset matching, since they are compact, fixed-width bit vectors that admit to very simple membership and subset checks. However, those checks are only probabilistically correct and may result in false positives. For sets S_1 and S_2 represented with Bloom filters (bit vectors) B_1 and B_2, $S_1 \subseteq S_2$ implies $B_1 \subseteq B_2$ (bitwise), and $B_1 \subseteq B_2$ (bitwise) implies $S_1 \subseteq S_2$ with high probability, although $S_1 \not\subseteq S_2$ is possible (false positive). The width and the number of hash functions that define the Bloom-filter representation also determine its false-positive

probability, and therefore are high-level design parameters that can be optimized for various application domains. The specific implementation of TagMatch that we describe in this paper uses 192-bit Bloom filters with 7 hash functions, which provide very conservative bounds for false positives in all the application domains we considered.[3] In cases where false positives are absolutely unacceptable, the system or the application can perform an additional exact subset check. In the following, whenever we mention query or database sets, we refer to their representations as Bloom filters, and we implicitly refer to their bitwise inclusion relations.

TagMatch stores its index in three main tables in CPU or GPU memory (see Figure 1): the *partition table* (CPU) associates each partition with its defining bit mask; the *tagset table* (GPU) associates each tag set s in each partition with a unique id that points to an entry in the *key table* (CPU) that therefore associates tag sets with keys.

TagMatch is designed to exploit parallelism on both CPUs and GPUs. Thus TagMatch can assign any number of threads to the various stages in the processing pipeline. TagMatch may also replicate the tagset table on all available GPUs o match queries in parallel on multiple GPUs. Alternatively, TagMatch can also partially replicate or simply partition an extremely large tagset table on multiple GPUs.

TagMatch batches queries and results between some processing stages to amortize the cost of transferring information and control between CPUs and GPUs. However, batching may also introduce excessive latency when, depending on the application, some partitions would see a few matching queries over a significant period of time. In those cases, some batches would not fill up and therefore would hold back the queries contained in them. To limit this holding time, TagMatch uses a configurable timeout period after which it automatically processes batches even if they are not full.

We now detail the off-line partitioning of the database and then the on-line processing stages of the TagMatch pipeline.

3.1 Off-Line Partitioning

TagMatch indexes the database D in a number of partitions so that all tag sets in a partition share a chosen bit mask (in their bit-vector representation). Given a configuration parameter MAX_P, TagMatch computes a set of masks that define a set of partitions, each containing up to MAX_P tag sets. More specifically, to make the matching process more efficient, TagMatch computes *balanced* partitions using a recursive partitioning scheme implemented in Algorithm 1. This is done off-line within the *consolidate*() function.

[3] Given two sets $S_1 \not\subseteq S_2$, an m-bit Bloom-filter encoding with k hash functions would result in $B_1 \subseteq B_2$ (a false positive) with probability $P(B_1 \subseteq B_2) = (1 - e^{-k|S_2|/m})^{k|S_1 \setminus S_2|}$, where $|S_1 \setminus S_2| > 0$ is the number of elements of S_1 that are not in S_2. In our case ($m = 192, k = 7$), with a set S_2 of $|S_2| = 10$ tags and another set S_1 that differs by $|S_1 \setminus S_2| = 3$ tags, the Bloom-filter encoding would indicate a false positive with probability 10^{-11}. Roughly the same 10^{-11} false-positive probability exists for a set S_2 of 5 tags and a set S_1 that differs by $|S_1 \setminus S_2| = 2$ tags.

Algorithm 1 Balanced Partitioning

Input: database sets D, max size MAX_P
Output: partition table $PT : Mask \to Partition$

```
PT ← ∅
Q ← {((mask = ∅) → (P = D), (used_bits = ∅))}
while Q is not empty do
  extract (mask → P, used_bits) from Q
  if |P| ≤ MAX_P and mask ≠ ∅ then
    PT ← PT ∪ (mask → P)
  else
    compute frequencies of one-bits in all sets B ∈ P
    pivot ← bit ∉ used_bits with freq. closest to 50%
    || pivot is a previously unused mask bit that splits P
    || as evenly as possible into two parts P_0 and P_1
    P_0 ← {B ∈ P | B[pivot] = 0}
    P_1 ← {B ∈ P | B[pivot] = 1}
    used_bits ← used_bits ∪ {pivot}
    add (mask → P_0, used_bits) to Q
    add ((mask ∪ {pivot}) → P_1, used_bits) to Q
  end if
end while
```

Notice that the MAX_P parameter (maximum partition size) can be used to balance the workload between the main processing stages in the TagMatch pipeline. Having a few large partitions would simplify the pre-processing on the CPU side but might overload the subset match on the GPU side. Conversely, small and therefore numerous partitions would reduce the cost of the subset match, but would also increase the cost of the pre-processing. We further discuss and evaluate this trade-off in Section 4.

3.2 Pre-Process

Given a query set q, the task of the pre-process stage is to forward q for further processing within all the partitions P_i that may contain matching tag sets for q (i.e., subsets of q). Since each partition P_i contains tag sets that share the same bits in $mask_i$, then the task of the pre-process stage is to find all $mask_i$ such that $mask_i$ is itself a subset of q (bitwise).

The pre-process stage uses a simple index for masks that is a kind of inverted index of bit positions (*partition table* in Figure 1). Concretely, the partition table is an array PT of 192 vectors of bit masks and the corresponding partition identifiers, where vector $PT[j]$ contains all the bit masks whose leftmost one-bit (bit set to 1) is at position j.

The pre-processing (Algorithm 2) then scans the one-bits of the query set q to classify q. The algorithm is not very sophisticated and yet it is quite efficient in practice because the partition table is very compact and therefore cache-efficient. Also, the concrete implementation of TagMatch uses bit vectors made of 64-bit blocks, so the subset checks in Algorithm 2 ($mask_i \subseteq q$) amount to three simple block opera-

tions.[4] It can also be shown that the efficiency of Algorithm 2 does not depend on the distribution of masks over the 192 bits in the partition table.

Algorithm 2 Pre-Process Stage

Input: partition table PT, query q
Output: forward q for processing within relevant partitions

```
for j ∈ all one-bit positions of q do
  for (mask_i → P_i) ∈ PT[j] do
    if mask_i ⊆ q then
      enqueue q for processing within partition P_i
    end if
  end for
end for
```

Whenever the pre-process stage fills up a batch for a given partition P_i, TagMatch extracts all the queries from the queue, copies them to the GPU memory, and invokes the *subset match* kernel for that batch of queries on partition P_i.

3.3 Subset Match

The *subset match* stage takes a batch of queries and a single partition of the tagset table, and returns the identifiers of the tag sets that match each query q in the batch.

We develop the subset match on a GPU following the Single Program Multiple Data (SPMD) model using the CUDA framework. With SPMD, one writes a "kernel" function designed to run on a single data item, then invokes that kernel on a set of data items, and the GPU schedules the execution of the kernel on all data items with as many parallel threads as its hardware resources allow.

Algorithm 3 High-level Subset Match Kernel

Input: batch of queries Q, table of tag sets P (partition)
Output: vector of pairs (query,set-id) *results*

```
// kernel code invoked on each individual entry of
// partition P = (s_1, id_1), (s_2, id_2), ..., (s_n, id_n);
// automatic variable thread_id identifies the entry
// assigned to this thread.
s ← P[thread_id].set
id ← P[thread_id].id
for q ∈ Q do
  if s ⊆ q then
    atomically append (q, id) to results
  end if
end for
```

At a high-level, the subset match kernel processes a single indexed tag set against a batch of queries (see Algorithm 3). The specific tag set is identified by an automatic $thread\_id$ variable.[5] Notice that, here too, the subset check amounts to a simple operation on each Bloom-filter block. Notice also that the output vector (*results*) is shared by all the threads of a kernel invocation, therefore the append operation uses an atomic increment on the size of the output vector.

3.3.1 Subset Match Optimizations

On the basis of the high-level design of Algorithm 3, we develop and implement several optimizations and performance improvements.

The first and most significant optimization is a pre-filtering step that takes place before the actual subset check. In CUDA, the threads in a kernel invocation are organized in *blocks*, such that all threads within a block run with consecutive thread ids on the same processor and can access a fast (but limited) block-level shared memory.

Algorithm 4 Pre-Filtering in Subset Match Kernel

Input: original queries Q', table of tag sets P (partition)
Output: batch of queries Q in shared memory

```
if thread_id = thread_block_first_id then
  first ← P[thread_id].set
  last ← P[thread_id + thread_block_size].set
  len ← leftmost_nonzero_bit(first ⊕ last)
  shared prefix ← first with all bit pos. ≥ len cleared
  shared Q ← ∅
end if
i ← thread_id − thread_block_first_id
while i ≤ |Q'| do
  if prefix ⊆ Q'[i] then
    atomically append Q'[i] to Q
  end if
  i ← i + thread_block_size
end while
```

The pre-filtering exploits the thread-block shared memory as shown in Algorithm 4. The first thread in the block computes the longest common prefix for all the tag sets assigned to the threads in the block, which requires only a simple bit-wise operation between the first and last tag sets in the block thanks to the fact that we store the sets in the tagset table in lexicographical order. Then, all threads in the block iterate through the original batch of queries (in parallel) to exclude the queries that do not match the common prefix.

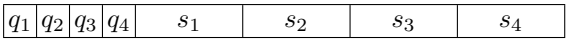
A second optimization affects the format of the output of the GPU kernel that needs to be copied to the CPU memory. Copying data from a GPU to the host memory is expensive because the bandwidth of the PCI-Express bus is limited and also because each call to the CUDA API has a fixed, non negligible cost. Therefore, one way to improve performance is to reduce the size of the output of the GPU kernel and to store that output in a single memory region, so as to minimize the number of copy operations.

The output consists of pairs (q, s) for a query q and a matching set s. In practice, we use 8-bit integers to identify

[4] In C, `((q[k] & ~mask_i[k]) == 0)`, for block k.

[5] The CUDA framework defines multiple variables to identify each thread and to control its behavior. For ease of exposition, we abstract from these implementation details and simply refer to a single $thread\_id$ variable.

a query within its batch, and a 32-bit integer to identify a tag set in the tagset table. However, because of alignment requirements, a simple structure to represent the (q, s) pair would require 64 bits, so a vector of pairs would result in a significant waste (38%) of memory and bus bandwidth. One way to avoid this waste is to store the query and set identifiers in two separate arrays. However, that would require two copy operations. We solve this problem by storing the output vector in groups of four (q, s) pairs, with four packed query identifiers preceding four packed set identifiers:



This layout yields a 100% or near-100% memory utilization, with a worst-case total loss of only three bytes.

Finally, we apply various fine-grained optimizations to the kernel code. For instance, we manually unroll simple loops and we reduce the number of loop iterations required to read the queries in a batch by accessing two queries within each iteration.

3.3.2 Workflow Optimizations

While the *subset match* kernel exploits multiple GPU cores, running one kernel at a time still can not fully utilize a GPU. This inefficiency is due to the round-trip time incurred in the processing of a batch of queries. When a CPU thread fills a batch of queries within the *pre-process* stage, that thread then must invoke the *subset match* kernel on that batch. In particular, the CPU thread must (1) copy the batch of queries from CPU to GPU memory, (2) invoke the *subset match* kernel on that batch of queries and the corresponding partition, and (3) copy the results back from GPU to CPU memory. And running one such sequence at a time leaves a GPU unused during the copy operations.

To overcome this limitation, and also to parallelize individual kernel executions whenever possible, TagMatch uses CUDA *streams* to enable multiple CPU threads to submit tasks to a GPU concurrently. A stream is an abstraction of a queue of GPU operations. Operations within the same stream execute sequentially in FIFO order, while operations in different streams are executed in parallel as much as possible, depending on the available hardware resources. In TagMatch, each CPU thread that needs to invoke a kernel on a batch of queries acquires an available stream and then issues the sequence of commands for parameter copy, kernel invocation, and result copy, through that stream.

Notice, however, that streams alone do not solve all synchronization problems. Consider the two copy operations. It is immediately possible for an invoking thread to issue a command to copy the minimal amount of data to transfer the batch of queries from CPU to a GPU, because the size of the batch is known at the time of the invocation. However, the same thread can not know at that same time the size of the result. A straightforward solution would be to issue a command to transfer the size of the result, and then only later, when that information becomes available, issue the command to retrieve the results with a minimal transfer. However, this would introduce an additional round-trip time and an additional synchronization point.

In TagMatch we avoid this inefficiency by associating each GPU stream with *two* buffers for the results (call them *even* and *odd*), each containing a length and a set of results. We then alternate between the two buffers as follows. In an odd transfer cycle, we use the odd buffer to transfer the length of the next (even) set of results, as well as the set of results for the current (odd) cycle. And for this copy operation we can issue a command with minimal transfer size because the exact size of this (odd) set was transferred in the previous (even) cycle and is readable from the even buffer. Then, similarly in the following even cycle, we find the length of the current (even) set of results in the odd buffer, which we use to issue the copy command for the current (even) set of results, and so on.

In summary, TagMatch takes full advantage of streams, with the following key benefits for the pipeline architecture. First, GPUs can process multiple batches on multiple partitions in parallel. Second, communication between CPUs and GPUs achieves an optimal utilization of the bus in both directions. Third, CPU threads can invoke entire sequences of GPU operations asynchronously, which means that CPU threads are no longer responsible for the synchronization between copy and processing operations, which in turn allows them to continue with pre-processing, key lookup/reduce, and merge tasks. Finally, TagMatch splits the workload across all available GPUs, with maximal inter-GPU parallelism in the case of full replication of the tagset table.

3.4 Key Lookup/Reduce and Merge

As shown in the previous section, the *subset match* stage outputs results in the form of (q, s) pairs, where q is a query id and s is a unique identifier of a tag set in the tagset table.

When new results become available for a partition, a CPU thread picks up these results and performs the *key lookup/reduce* stage, which accesses the *key table* to retrieve the set of keys associated with each set-id s. The thread then groups these keys by query in a *results table* (see Figure 1), associating each query with a list of sets of keys. Additions to the *results table* also use the proper atomic operations to allow access from multiple threads.

For each query q going through the matching pipeline, TagMatch maintains a counter of all the batches (partitions) within which q is forwarded for processing. When q's preprocessing terminates, and the counter goes back to zero, signaling that all the results for all the batches returning from the GPUs have been accounted for, then TagMatch runs q through the last *merge* stage. In the case of a *match* query, that requires no additional processing. In the case of a *match-unique* query, the merge stage merges all sets of keys associated with q into a single set.

4. Evaluation

We now present the results of an experimental evaluation of TagMatch. The general objective of this evaluation is to assess the performance of TagMatch in terms of throughput. Most importantly, we are interested in (1) the effective throughput measured in queries processed per time unit under realistic workloads, (2) the scalability of TagMatch with respect to the size of the database and queries as well as to the capabilities of the platform (e.g., available CPU threads), and (3) the performance of TagMatch relative to other comparable state-of-the-art systems.

4.1 Subjects and Experimental Setup

The main subject of our experimental analysis is a C++ implementation of TagMatch as described in Section 3. In addition, we use the following subjects:

- *prefix tree:* a main-memory implementation of a subset matching algorithm that indexes database sets into a prefix tree. Specifically, this system uses a Patricia tree and solves the subset matching problem by navigating such tree. This implementation is representative of most state-of-the-art approaches based on trees (see Section 5).
- *ICN matcher:* an implementation of a state-of-the-art algorithm specifically designed to perform packet forwarding in Information Centric Networks (ICN) [15]. In this context, the database encodes forwarding information represented as sets of tags, and the queries are the packets to dispatch. This algorithm is also based on a prefix tree and, similar to TagMatch, it is designed for high throughput.
- *MongoDB:* the *MongoDB* Database Management System (version 3.2.10), which offers an explicit subset operator.

Our testbed is a general-purpose machine equipped with two Intel Xeon E5-2670 v3 processors, each with 12 cores running at a clock frequency of 2.30GHz, and 64GB of RAM. The machine also has two NVidia TITAN X graphic cards each with 12GB of GDDR5 RAM.

To make the comparison as fair as possible, we configure the prefix tree and the ICN matcher to use Bloom filters with the same size as TagMatch (192-bit), and we try to feed the same input and allocate the same system resources to all subject systems. Thus for all the comparative experiments, with give each system the same number of threads.

Still, we could not perform exactly the same experiments with all systems. In particular, since the ICN matcher uses a significant amount of memory to build its index, we could only test the ICN matcher with a reduced portion of the largest workloads. We discuss this case in Section 4.3.2. We encounter analogous and even more extreme difficulties with MongoDB. In fact, the performance of MongoDB is limited to the point of making larger experiments impossible or pointless. We therefore test MongoDB with specially crafted and relatively small workloads. We discuss the case of MongoDB in Section 4.4.

4.2 Workloads

We evaluate the absolute performance of TagMatch using a workload representing a Twitter-like messaging system. In this workload, the database entries are tag sets that represent the interests of users, the keys associated with each tag set are the identifiers of the users interested in that tag set, the queries are the tweets published by the users, and the tags in the queries are the hash-tags (keywords) of the tweets.

4.2.1 Database Sets

The workload includes 300 million users (keys), which is roughly the number of users that are active on Twitter every month,[6] and contains over 212 million unique sets representing user interests.

We generate the set of interests based on a real dataset of tweets provided by the TREC conference (2011-2012), containing 16 million tweets recorded during two weeks in 2011.[7] To derive realistic relations between users, we use a graph of 41.7 million Twitter users and 1.47 billion follower relations [8]. To amplify the data set and to prevent a bias toward the English language in the workload generation, we artificially create multiple languages in our data set: given a tag, we "translate" it by adding a prefix that indicates the new language. For example, the original tag *cat* becomes *fr_cat* in French or *it_cat* in Italian. In our workload we assume that 40% of the users speak only one language while the remaining 60% speak two languages[8]. To select the language spoken by each user we use two different distributions: the first one is the language distribution on Twitter [6], while the second one is the distribution of the most frequent second languages used in the world[9].

For each user in our workload we generate a set of interests as follows. First we select the languages spoken by the user according to the distributions mentioned above. Then we pick the number of followed publishers according to the follower distribution that we derive from the Twitter graph. Then we randomly select the publishers from the list of users available in our data set and collect their tweets. We generate one interest for each publisher by randomly selecting one of their tweets and using the hash-tags in that tweet. In addition, we "translate" the hash-tags using one of the two languages assigned to the user, since we assume that a user follows only publishers that write in one of the user's languages.

If the publisher of the tweet is a frequent writer, we also add the id of the publisher as a tag in the interest. We consider a publisher to be a frequent writer if he or she is ranked in the top 30% based on the number of published

[6] Twitter stats: https://about.twitter.com/company

[7] https://github.com/lintool/twitter-tools/wiki/Tweets2011-Collection

[8] http://ilanguages.org/bilingual.php

[9] https://www.ethnologue.com/statistics/size

tweets. An interest with only hash-tags describes the set of information that the user wants to collect, while an interest that includes a user id selects only the information of interest that are generated by the user with that id. This procedure results in interests containing an average of five tags.

4.2.2 Queries

One method to generate queries is to select the tags in each query (that is, the hash-tags in each tweet) more or less uniformly at random. However, that would most often result in queries that are immediately and very efficiently discarded in the initial pre-filtering stage. So, to be conservative, we instead create a workload in which each query matches at least one tag set in the database. To do that, we generate each query by selecting a tag set from the database to which we then add between two and four extra tags selected at random. (We also experiment with a broader range of additional tags; see Section 4.3.1.) The rationale for this generation algorithm is that the selected set from the database would perhaps represent a generic topic while the additional tags would characterize the specificity of the tweet. Beyond that, as we said, the intended effect of this method is to obtain conservative results for the matching throughput, since the method essentially forces every query to go through the subset match phase on the GPU stage and then the key lookup/reduce and merge phase on the CPU (in the case of *match-unique*).

4.3 Performance and Scalability

We now present three series of experiments intended to test the performance and scalability of TagMatch under a variety of workloads and configurations. Thus in these experiments we first measure the throughput in terms of number of processed queries per seconds, and later we measure the matching latency. We analyze the performance of TagMatch in absolute terms and also in comparison with the state-of-the-art prefix tree.

4.3.1 Size of the Query Set

In the first series of experiments we test the performance of TagMatch and the prefix tree with queries of increasing sizes. Figure 2 shows the results of these experiments. As explained in Section 4.2.2, the primary workload we use consists of queries with between two and four additional tags, which corresponds to the histograms at positions 2–4 in Figure 2.

It is clear from the figure that the number of tags in each query has a very significant impact on performance (notice the log scale). This is intuitive from an algorithmic perspective, since query sets of higher cardinality are likely to lead to more one-bits in the Bloom filters, which are likely to match more prefixes and therefore require more data transfer between CPUs and GPUs, and also more work on both sides.

However, notice that for the same intuitive reasons, the decline in *input* throughput, which is what we measure in

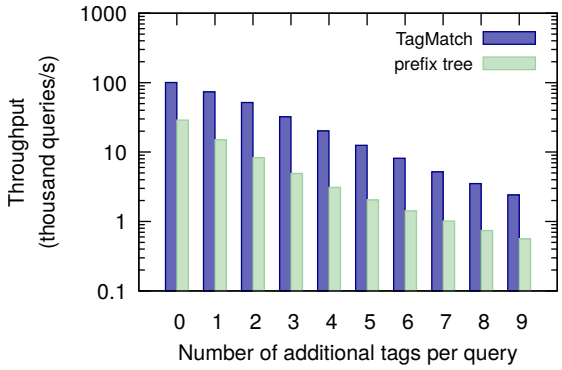


Figure 2: Average throughput for *match-unique* with queries of different sizes.

Figure 2, does not result in a corresponding decrease in *output* throughput. In fact, as it turns out, and as we demonstrate with the measurements of Figure 3, the output throughput for the same experiments increases significantly with the query size. We argue, intuitively, that for selective queries, meaning queries that have a few matching sets and that represent tweets in the long tail of popularity, the most important performance metric is the input throughput. Conversely, for queries with a high fan-out, which represent highly popular Twitter traffic, the limiting factor and therefore the most interesting performance metric is the *output* throughput. And in this respect, the experiments show that TagMatch performs quite well.

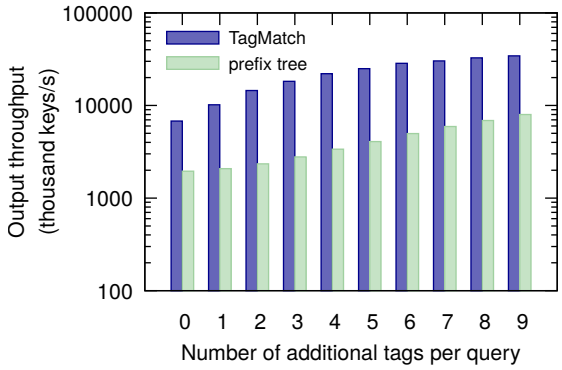


Figure 3: Average output rate for *match-unique* with queries of different sizes.

In comparative terms, TagMatch is consistently faster than the prefix tree system by almost one order or magnitude for both the input and output throughput. In all these experiments, the results for *match* (not shown in the graphs) are very close to the results for *match-unique*.

4.3.2 Size of the Database

In a second series of experiments, we test the scalability of TagMatch with respect to the size of the database. We

report the results of this analysis in Figure 4. Once again we measure the throughput (input) as we vary the size of the database from 20% to 100% of the entire Twitter database of 212 million tag sets.

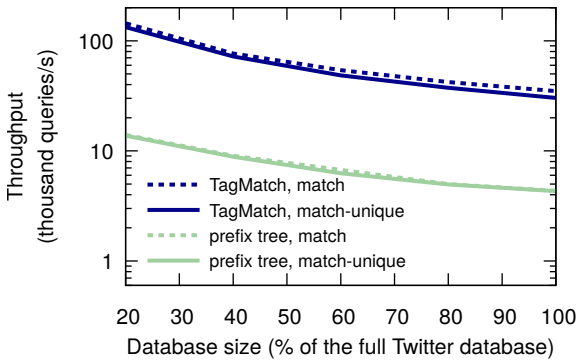


Figure 4: Average throughput for *match* (left) and *match-unique* (right) with different database sizes.

The salient result of this analysis is that TagMatch can process more than 30 thousand queries per second in the case of *match-unique*, and more than 35 thousand queries per second in the case of *match*, with the full database of 212 million unique sets. This is well above the entire traffic of Twitter, which was on average 6000 tweets per second as of 2015—on a single commodity machine, with the added capability of filtering tweets based on their content. In contrast, the state-of-the art CPU implementation based on a prefix tree can process about 4400 queries per second both in the case of *match* and in the case of *match-unique*.

As Figure 4 shows, and as one would also expect intuitively, the size of the database significantly affects performance: with a database with 20% of the entries of the full Twitter workload, TagMatch can achieve a throughput of over 130K queries per second in the case of *match-unique* and more than 140K queries per second in the case of *match*, compared to the CPU implementation that achieves a throughput of less than 14K queries both in the case of *match-unique* and in the case of *match*.

	database size			
	10%	20%	10%	20%
system	*match*		*match-unique*	
TagMatch	268.8	144.4	249.3	133.0
Prefix tree	21.1	14.0	21.0	13.8
ICN matcher	27.6	17.4	27.5	16.8

(thousand queries per second)

Table 3: Comparison with the CPU prefix tree, CPU algorithm for ICN. Average throughput for *match* and *match-unique* with 10% and 20% of the full Twitter database.

Table 3 compares the throughput of TagMatch with the ICN matcher [15]. In this case we could only consider up to 20% of the full Twitter database because the implementation of the ICN matcher requires a lot of memory during the construction phase to generate the final index that is actually used for the matching. Creating the index for databases larger than 20% of the full workload would require more than the 64GB of main memory available on our machine. The ICN algorithm reaches a higher throughput than the CPU prefix tree algorithm, but remains about an order of magnitude slower than TagMatch.

4.3.3 Number of Threads

We now test the ability of TagMatch to distribute its work load over multiple threads. In particular, we measure the throughput as we allocate an increasing number of threads to the CPU stages.

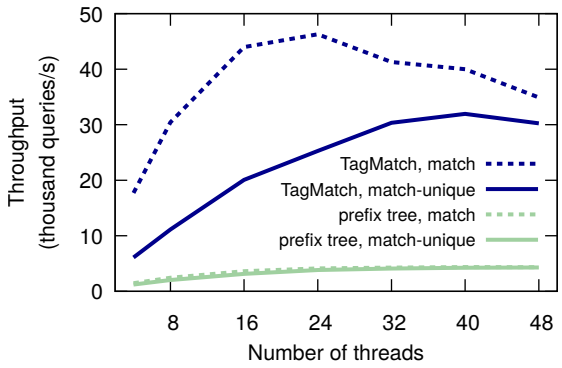


Figure 5: Average throughput for TagMatch and the CPU prefix tree with different numbers of CPU threads.

As shown in Figure 5, for both *match* and *match-unique* queries, TagMatch achieves an almost linear scalability in the number of threads, with a speedup of more than 1.8× from 4 to 8 threads, and 3.3× from 4 to 16 threads. With more than 24 threads, the throughput for *match* decreases, while the throughput for *match-unique* keeps growing up to more than 40 threads. The difference between the two algorithms is simply due to the higher CPU load of the merge stage for *match-unique*.

The decrease in parallelism speedup over a certain number of threads is instead due to a limitation of the GPU architecture. Basically, beyond a certain number of threads, the GPU stages become the bottleneck for the whole pipeline. However, there are also other factors that limit the pipeline to an overall throughput that is lower than the maximal throughput of the GPU. In particular, CPU threads and the GPUs interact through a set of GPU "streams" (see Section 3.3.2), and on our platform we can allocate a maximum of 20 streams (10 per GPU), primarily due to memory limitations. Having more streams would allow for more parallelism. Furthermore, our test machine has 24 real cores, so when we allocate 32, 40, and 48 threads, those run using Intel's Hyper-Threading technology.

4.3.4 Latency

The batching of queries in the TagMatch pipeline is essential to achieving high throughput but also induces a latency overhead. To limit latency, an application can set a timeout after which a batch of queries will be pushed through the GPU (see Section 3). In Figure 6 we characterize the distribution of the matching latency for different timeout settings, including the case with no timeout.

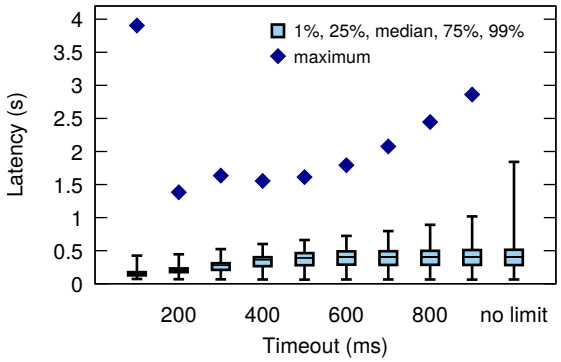


Figure 6: Distribution of the end-to-end latency for *match-unique* with the full Twitter database.

These experiments show that the timeout mechanism is indeed effective in reducing latency. Even without a timeout limit, the vast majority of queries (99%) incur a latency of less than 2 seconds, with a median latency of under 400ms. However, the maximal latency values, also shown in Figure 6, can be significantly higher.

The case in which the timeout is set to 100ms is particularly interesting. Excluding the case with no timeout limit, the 100ms setting is the one with the highest maximal latency at nearly 4 seconds. This is due to the fact that a very short timeout leads to inefficiencies in the use of the CPU/GPU pipeline. In particular, a short timeout triggers too many invocations of the GPU matching kernels with batches of only a few queries, and since the matching kernel requires the same amount of GPU resources even for small batches, this increases the load on the GPU without a corresponding increase in throughput. In fact, with a timeout setting of 100ms, TagMatch suffers a loss of overall throughput of about 20% (24 thousand *match-unique* queries per second).

However, this inefficiency disappears very quickly with a slightly higher timeout setting. A timeout as short as 200ms already enables TagMatch to process more than 28 thousand queries per second, and a timeout of 300ms further increases the throughput to 30 thousand queries per second, which is close to the maximum achievable with no timeout limit at all.

4.3.5 Balance Between CPU and GPU Load

We now study another algorithmic aspect of the TagMatch pipeline that balances the load between CPUs and GPUs. As discussed in Section 3.1, TagMatch uses a configuration parameter MAX_P to define the maximum number of tag sets in each partition. Therefore, for a given set of database sets, MAX_P controls the balance between the number and size of the partitions, which in turn can balance the load between the pre-processing phase (on CPUs) and the subset match phase (on GPUs). Large partitions simplify the pre-processing phase, but might overload the subset match, while several small partitions reduce the complexity of the subset match but increase the cost of pre-processing, as well as the duplication of queries into multiple batches.

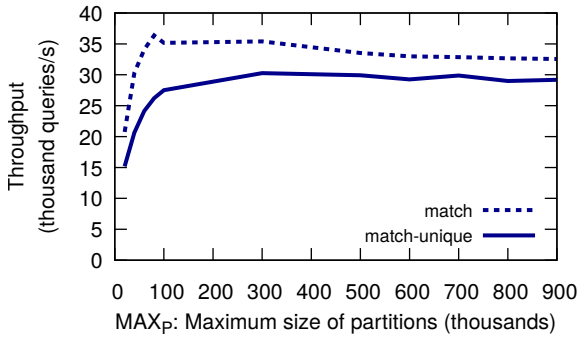


Figure 7: Average throughput of TagMatch for *match* and *match-unique* with different size of partitions.

Figure 7 shows the results of an experiment intended to analyze this trade-off. The chart shows how the throughput of TagMatch changes with different values of MAX_P for the same database. We observe that TagMatch achieves the best performance with around 200K tag sets per partition, and that the throughput remains stable after this threshold. The results do not differ significantly in the cases of *match-unique* and *match*.

4.3.6 Off-Line Partitioning Costs and Memory Usage

TagMatch provides two functions, *add-set* and *remove-set*, to add or remove sets from the database. However, these changes become effective only after a call to the *consolidate* function, in which TagMatch builds its partition and tagset tables using the balanced partitioning of Algorithm 1. We now evaluate the performance of the partitioning algorithm as well as the memory usage on both the CPU and GPU side.

Figure 8 shows the running time of the partitioning algorithm as a function of the size of the database. The experiments confirm that the algorithm has a linear complexity, and they also demonstrate that the actual performance is reasonable in absolute terms, with a maximum off-line running time of about 50 seconds for the full workload of 200 million tag sets. As a rough comparison, consider that MongoDB requires about 33 seconds for a table of only 5 million sets, for which our partitioning algorithm runs in about 2 seconds.

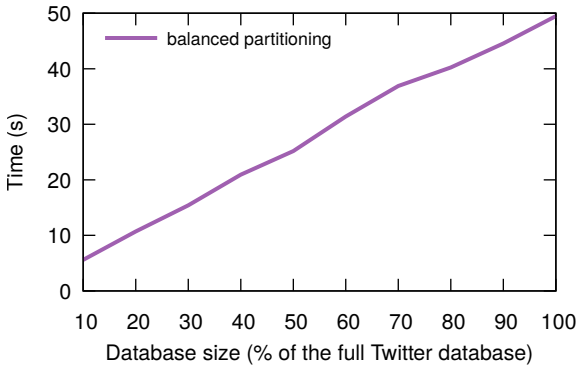


Figure 8: TagMatch partitioning time, $MAX_P = 200$K.

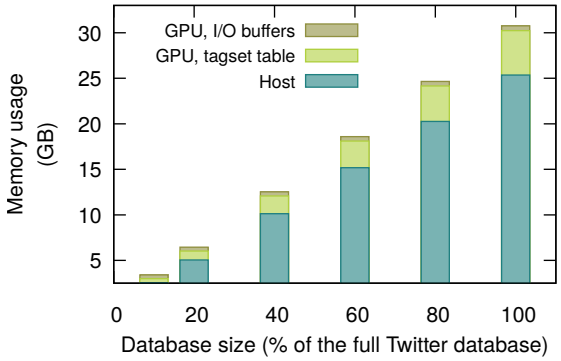


Figure 9: TagMatch memory usage (GB).

Figure 9 shows the memory usage on the CPU (Host) and GPU sides. The Host memory is used almost exclusively for the key table, with only a small portion for the partition table and the buffers used for communication between the CPUs and the GPUs. The memory of the GPUs is used primarily for the tagset table, with a small fraction used for communication buffers.

4.4 Comparison with MongoDB

This section evaluates the performance of subset query processing in MongoDB version 3.2.10. We treat MongoDB as a special case due to the significant difference in performance with TagMatch. In particular, since we could not use the full Twitter workload used in Section 4.3 due to the higher processing time and memory consumption of MongoDB, we construct and experiment with a scaled-down workload with a similar selectivity and with the same number of additional tags per query.

We configure MongoDB to store a database of sets of tags on a RAM disk in main memory. We also force MongoDB to index the entries of the database to improve the performance of the query process. We run MongoDB in two configurations, first as single server and then by sharding the database over multiple instances. In both settings, we use the Java API to connect and submit queries through a TCP socket (on localhost). We then use a single thread to submit asynchronous queries. We also experimented with multiple client connections. However, even though MongoDB can process queries from different connections in parallel, we did not observe any performance improvement.

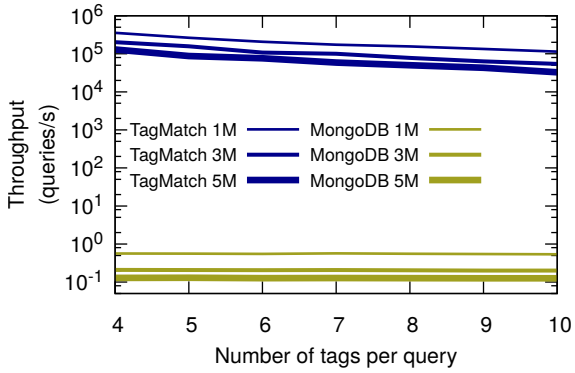


Figure 10: Comparison with MongoDB. Average throughput for *match* with different number of tags per query.

Figure 10 shows the results of an experiments in which we compare TagMatch and MongoDB in the single-server setting with different database sizes, different numbers of tags per database set, and varying numbers of additional tags per query. Even with a small database of one million sets, MongoDB takes more than two seconds to process a single query, and the performance decreases significantly with the size of the database (notice the log scale), down to more than 10 seconds per query in the case of 5 million sets. Conversely, neither the number of tags in the database sets nor the number of additional tags in each query influence the overall performance of MongoDB, despite the fact that they both have a significant impact on the selectivity of the workload. In comparison, TagMatch can process more than 32,000 queries per second even in the most challenging scenario of 2-tags database-sets and 10-tags queries.

We also test a distributed deployment of MongoDB with a database sharded over multiple servers. In this setting, MongoDB sends each query to all the instances for processing on each individual database shard. We perform an experiment in this setting to evaluate the benefits and scalability of distribution and sharding. To minimize the network overhead, we run all MongoDB instances on the same physical machine, and only consider a relatively small deployment of up to 24 instances (the machine has CPU 24 cores and sufficient memory). We show the results for a database of 3 million entries, each containing 3 tags, and for queries of 6 tags. The results of this experiment, shown in Figure 11, demonstrate that sharding and distribution are clearly beneficial, and specifically that the throughput of MongoDB increases linearly up to 8 instances and overall by a factor of 3 with 24 instances. However, even assuming a perfectly linear scala-

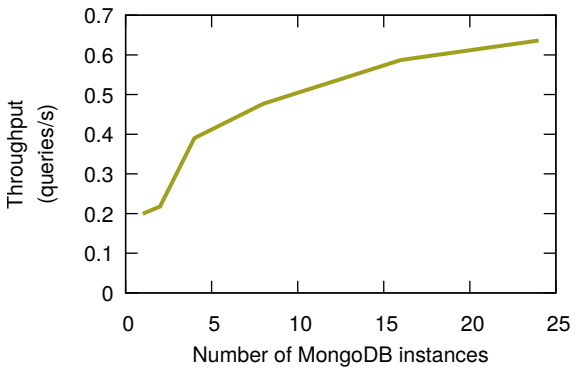


Figure 11: Scalability of MongoDB with sharding. The database contains 3 million entries, with 3 tags each. Queries contain 6 tags.

bility, MongoDB would require tens of thousands of server instances to reach the level of performance of TagMatch.

4.5 Experience with an Alternative Design

In the early stages of the development of TagMatch, we experimented with a design in which both the pre-process and subset-match phases would run entirely on GPUs. This architecture is technically feasible thanks to the "dynamic parallelism" of the newer NVidia GPUs.

With dynamic parallelism, it is possible to launch a new kernel from within a running kernel on the GPU. Thus, one can launch a kernel that performs the pre-process algorithm on a batch of input queries and that uses partition queues in the global memory of the GPU. When one of those queues fills up, the pre-process kernel can then invoke a new subset-match kernel on that queue directly from within the GPU.

This architecture is potentially advantageous, since it allows for numerous parallel subset-match kernels for different partitions. This approach also overcomes one of the factors that in part limits the performance of TagMatch, namely the need to transfer a single packet to the GPU potentially many times (one for each matching partition). Furthermore, the pre-process algorithm is also inherently parallelizable, and should therefore work well on a GPU.

And yet, the prototype we built was not that efficient. As it turns out, the GPU-only design works well when the vast majority of packets are filtered out in the pre-process phase, but not when many packets reach the subset-match phase. In this latter case, the pre-process algorithm must copy many queries into potentially many partition queues, which induces many atomic operations and an almost random access pattern into the global (slow) memory of the GPU. Also, the GPU-only design still requires synchronization, as well as the transfer of some partial results between CPUs and GPUs, which further limits parallelism.

5. Related Work

Determining subset relations on large collections of sets is a fundamental problem in data management and retrieval. We review the main results developed in the past, in particular emphasizing the ideas that influenced our TagMatch architecture the most. We start with general solutions for the subset matching problem and then focus on subset matching in data management, streaming, and networking.

Recall that, given a database D of sets $s_1, s_2, \ldots, s_n$ and a query set q, *subset matching* amounts to finding all sets $s_i \in D$ such that $s_i \subseteq q$. When the universe of elements (tags) from which we draw the database sets and the query is finite, then the problem is also known as the partial matching problem. Notice that in most applications it is possible to compress a finite or even an infinite universe (such as the set of string tags) into a finite and often small one. This compression is particularly effective when the database sets are small, as is the case in our target application domain, in which case Bloom filters are an ideal form of compression.

It is easy to see that the subset matching problem has two trivial solutions: (1) scan the entire database for each query, which requires $O(n)$ space and $O(nm)$ total time, or (2) precompute and store all answers to all queries, which requires $O(2^m)$ space and $O(m)$ query time; where n is the size of the database and m is the total number of unique tags (or bits in their Bloom-filter representation).

Rivest proposes the first non-trivial results [19]. Rivest develops a solution based on a hash table that amounts to a partitioning for sets, and a solution that organizes the sets in the database as a prefix tree and that requires linear storage and, on average, sublinear query time. Rivest also reviews other methods based on inverted lists. For each element x, an inverted index stores the list $list(x)$ of all sets s_i that contain element x. Using inverted indexes, *superset* matching (finding all s_i such that $s_i \supseteq q$) amounts to computing the intersection between the lists $list(x_j)$ for every element $x_j \in q$, while subset matching amounts to counting how many times each set appears in all the lists [23].

Data Management Helmer and Moerkotte [5] study subset matching as a join operator for set-valued attributes. Helmer and Moerkotte introduce signatures to avoid expensive set comparisons. A signature is a hash value over the content of the set that preserves the partial order on sets induced by the subset relation. In TagMatch we use Bloom filters as a space-efficient signature.

Ramasamy et al. [17] proposed partitioning as a divide-and-conquer strategy. This is also the essence of Rivest's first solution, and an idea we use in TagMatch. Specifically, our algorithm partitions the database of tag sets and preprocesses the query to discard all the tag sets that belong to partitions that cannot satisfy the query. Melkin and Garcia-Molina [12] propose two algorithms intended to improve the effectiveness of partitioning by dynamically adapting the partitioning criteria to the features of the workload.

More recent proposals from the database community build on the early development of the ideas of signatures, partitions, and inverted lists, and also Rivest's idea of organizing the database as a trie [7]. Perhaps the most current and advanced algorithms are the ones developed and analyzed by Luo et al. [9] and by Bouros et al. [2]. Luo et al. propose two algorithms. The first one, called PATRICIA trie-based signature join (PTSJ), uses hash-based signatures to encode a set as a shorter bit string, and builds a trie out of all signatures. The second algorithm works on the actual elements of the sets and builds what amounts to an inverted index. Bouros et al. propose an adaptive methodology to reduce the space requirements and the cost of traversing the trie.

The trie-based algorithm we developed and used in our comparative evaluation is a representative of these state-of-the-art algorithms. Our algorithm is in fact conceptually very similar to the PTSJ algorithm by Luo et al., which seems to be the best-performing algorithm in most cases according to the evaluation of Luo et al. Furthermore, based on the results Luo et al., our trie-based algorithm seems to outperform PTSJ on all comparable scenarios.

Streaming and Networking While in this paper we consider subset matching as a data selection problem, a conceptually similar problem is at the root of a new category of forwarding algorithms for packet-switching and similar messaging systems. In particular, the forwarding information base (or FIB) in routers corresponds to our database entries, and incoming packets corresponds to our stream of queries. For IP forwarding, a router must find the longest prefix in its FIB that matches the destination address of each incoming packet. This is a fundamentally easier problem than subset matching, also because of the fixed and small size of IP addresses. However, a recently developed notion of Information Centric Networking (ICN) [1] extends the idea of IP networks by considering richer forms of addressing.

One of the first algorithms for high-throughput ICN forwarding was developed by Wang et al. [21] for hierarchical name-based addressing. Thus in this context, forwarding amounts to a longest name-prefix matching over path names, which is still fundamentally different from subset matching. However, the work of Wang et al. is particularly relevant to this work because it also exploits the parallelism of GPUs. Wang et al. implement longest-prefix matching on a character trie compressed in a data structure called multi-aligned transition array, and report a throughput of 63.52 million packets per second with a database of 10 million entries.

Most ICN forwarding systems implement longest name prefix matching with a hash-table: the algorithm stores the database in a hash table and, for an input name of n components, proceeds by first looking up the whole name, then the prefix of length $n-1$, and so on. On this basis, authors have built a number of variants [16, 22, 24].

These systems report throughput values that are orders of magnitude higher than what we report for TagMatch. However, once again, notice that these systems solve *prefix* matching, a problem that is significantly simpler than *subset* matching. A more relevant work in the context of ICN is the algorithm developed by Papalini et al. for an ICN addressing scheme based on subset matching [15]. Papalini et al. also build a prefix trie, but then use a number of heuristics to rearrange and compress the trie. We use the algorithm by Papalini et al. in our comparative evaluation. However, since the heuristics require additional space to restructure the trie, we could apply the algorithm only to a few reduced-size scenarios. In these scenarios the algorithm by Papalini et al. is competitive with the basic trie-based matching algorithm but also significantly slower than TagMatch.

Some studies on content-based publish-subscribe systems explore the idea of exploiting parallel architectures for matching. Farroukh et al. [4] exploit multi-core CPUs both to reduce the latency for individual messages and to increase the overall throughput of the system. Similarly, Margara and Cugola [11] implement a counting-based algorithm on GPUs that exploits the parallelism to both decrease the latency and increase the throughput. Finally, Tsoi et al. [20] focused on ad-hoc (FPGA) hardware. While the problem of content-based forwarding prove to be amenable to a parallel implementation, the usage of richer matching semantics than in our work penalizes the performance, which are order of magnitude lower than those we achieve.

6. Conclusion

We presented the design and implementation of TagMatch, an efficient subset matching engine that exploits a hybrid system of CPUs and GPUs. TagMatch targets applications that perform subset matching between a high-rate stream of queries, each consisting of a relatively small set of tags, with a large database of hundreds of millions of tag sets.

We presented an extensive evaluation of TagMatch in which we test its absolute performance with various workloads and we compare it with the MongoDB database system, with a message forwarding system, and with a system based on a prefix tree that is representative of the most efficient solutions for subset matching we know of. TagMatch outperforms these systems, in most cases with at least an order of magnitude higher throughput. Remarkably, TagMatch can process about five times the average message traffic of Twitter on a single commodity machine, while offering a refined service that dispatches tweets based on the interests of the users rather than only on the publisher of the tweets.

Our plans for future work include the integration of TagMatch within a full fledged data processing or messaging systems, to measure the benefits that it can bring to such application domains.

Acknowledgments

This work was supported in part by the Swiss National Foundation under grant number 200021-157164 ("TagMatch").

References

[1] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[2] P. Bouros, N. Mamoulis, S. Ge, and M. Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, pages 1–28, 2015.

[3] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the International Conference on Management of Data*, SIGMOD '98, pages 355–366, 1998.

[4] A. Farroukh, E. Ferzli, N. Tajuddin, and H.-A. Jacobsen. Parallel event processing for content-based publish/subscribe systems. In *Proceedings of the International Conference on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, 2009.

[5] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '97, pages 386–395, 1997.

[6] L. Hong, G. Convertino, and E. H. Chi. Language matters in twitter: A large scale study. In *ICWSM*, 2011.

[7] R. Jampani and V. Pudi. Using prefix-trees for efficiently computing set joins. In *Proceedings of theInternational Conference on Database Systems for Advanced Applications*, DASFAA '05, pages 761–772, 2005.

[8] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[9] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. D. Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *Proceedings of the International Conference on Data Engineering*, ICDE '15, pages 303–314, 2015.

[10] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 157–168, 2003.

[11] A. Margara and G. Cugola. High-performance publish-subscribe matching using parallel hardware. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):126–135, 2014.

[12] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Transactions on Database Systems*, 28 (1):56–99, 2003.

[13] T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, KDD '98, pages 284–288, 1998.

[14] M. Papalini, A. Carzaniga, K. Khazaei, and A. L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN'14, pages 17–26, 2014.

[15] M. Papalini, K. Khazaei, A. Carzaniga, and D. Rogora. High throughput forwarding for ICN with descriptors and locators. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, pages 43–54, 2016.

[16] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 137–148, 2014.

[17] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB '00, pages 351–362, 2000.

[18] R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *Proceedings of the SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 20–27, 2003.

[19] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[20] K. H. Tsoi, I. Papagiannis, M. Migliavacca, W. Luk, and P. Pietzuch. Accelerating publish/subscribe matching on reconfigurable supercomputing platforms. In *Many-Core and Reconfigurable Supercomputing Conference*, MRSC '10, 2010.

[21] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: A gpu-based approach. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 199–212, 2013.

[22] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for named data networking. In *Proceedings of the International Symposium of Quality of Service*, IWQoS '14, pages 198–207, 2014.

[23] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, 1994.

[24] H. Yuan and P. Crowley. Reliably scalable name prefix lookup. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 111–121, 2015.