NetSHa: In-network Acceleration of LSH-based Distributed Search

Penghao Zhang*, Heng Pan*, Zhenyu Li, Penglai Cui, Ru Jia, Peng He, Zhibin Zhang, Gareth Tyson, and Gaogang Xie

Abstract—Locality Sensitive Hashing (LSH) is widely adopted to index similar data in high-dimensional space for approximate nearest neighbor search. Demanding applications (*e.g.* web search) mean that LSH must exhibit low response times and high throughput. To achieve this, they tend to load balance between multiple machines. However, as the scale of concurrent queries and the volume of data grow, large numbers of index messages are required. Hence, the network is a key bottleneck. To address this gap, we propose NetSHa, which exploits the computational capacity of programmable switches. Specifically, we introduce a heuristic sort-reduce approach to drop potentially poor candidate answers while preserving search quality. Then, NetSHa aggregates good candidate answers from different index messages when transmitting them. Through this, it reduces the network communication cost. Furthermore, we introduce a best-effort replacement mechanism to improve its concurrency. We implement NetSHa on a Barefoot Tofino programmable switch and evaluate it using 7 real-world datasets. The experimental results show that NetSHa reduces the packet volume by $4 \sim 10$ times and improves the search efficiency by least $3\times$ in comparison with typical LSH-based distributed search frameworks.

Index Terms—Local sensitive hashing, distributed search, in-network computation.

1 Introduction

Nearest neighbor (*NN*) search, also known as *similarity search*, involves retrieving the most similar item(s) to a given query. It has played an important role in a variety of applications, such as recommendation systems [2], natural language processing [3] and sequence matching [4]. In such systems, low delay and high responsiveness are a must. Despite this, NN search is often applied to data that is processed as multi-dimensional vectors, leading to high computational costs. This has led to *approximate nearest neighbor* (*ANN*) algorithms [5], [6] that strive to improve the performance of similarity search in high dimensional space.

In this paper, we focus on one particularly well-known approach to ANN: *Locality Similarity Hashing (LSH)* [7], widely recognized as one of the most effective methods [8]. Using this approach, a family of hash functions are carefully designed such that similar data items (in high-dimensional space) can be allocated to the same buckets with a high probability. Thus, the system need only check a relatively small number of buckets to locate similar items (rather than querying the whole dataset).

LSH has garnered a wealth of attention from both industry and academia, particularly for data-intensive applications, *e.g.* web search [9], [10]. Thus, there are a number of implementations of LSH functions, such as MinHash [11] for Jaccard distance, E2LSH [12] for Euclidean distance and SimHash [13] for Angular distance. Researchers have also put forward variants of LSH such as Ternary Locality Sensitive Hashing (TLSH) [14] and Entropy LSH [15] to further improve efficiency. Regardless, as datasets have increased in size (*e.g.* billions of images for reverse lookups [16]) it has become infeasible to execute LSH on a single machine. Consequently, there have been recent attempts to implement high-performance LSH-based *distributed* search systems [17], [18], [19] within clusters of servers.

The distribution of LSH across a cluster poses a number of challenges though, most notably related to increased network communication costs [20]. For instance, let us consider query processing in an LSH-based distributed search system. When a query arrives, it must first interrogate a set of servers. In return, each server may generate multiple candidate answers. These candidate answers are then collected together (again via network communication) for further processing, such that the final top-K answers are compiled. As a result, it is necessary for the distributed answers to be transmitted to one centralized server. This will lead to network congestion (or "incast") which hurts performance. Our experimental results (see Table 2) confirm this significant overhead due to network communications. Although recent efforts aim at balancing load between different servers [17], [21], these methods reduce the network overhead only on end systems (a.k.a at the server side), while the network itself just blindly forwards candidate answers. However, we also find that the network, rather than the servers, can become the bottleneck for delivery high performance search.

To address this challenge, we propose **NetSHa**, a system that exploits the computational capacity of programmable switches (*i.e.* P4 [22]) to reduce the network costs of LSH

- *Penghao Zhang, Heng Pan, Zhenyu Li, Penglai Cui, Ru Jia, Zhibi Zhang are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zhangpenghao@ict.ac.cn, panheng@ict.ac.cn, zyli@ict.ac.cn, cuipenglai20b@ict.ac.cn, jiaru@ict.ac.cn, zhangzhibin@ict.ac.cn). Corresponding author: Zhenyu Li.*
- *Peng He is with ByteDance Inc. (e-mail: hepeng.0320@bytedance.com).*
- *Gareth Tyson is with the Queen Mary University of London (e-mail: g.tyson@qmul.ac.uk).*
- *Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences (e-mail: xie@cnic.cn).*

**Co-first authors.*

The preliminary shorter version [1] of this paper appeared at IEEE INFOCOM 2021.

and improve performance. NetSHa utilizes switches to reduce and aggregate candidate answer packets during their transmission. As a result, through multiple network hops (switches), the volume of packets can be significantly reduced. This alleviates the network bottleneck, which in turn reduces the network latency and enables the system to support increased concurrent queries. It is also noteworthy that NetSHa (at the network side) is complementary with prior optimizations on server side [17], [21], such that they can cooperatively improve the system efficiency.

To enable this, we propose a simple extension to the Internet Protocol (IP) and design novel scheduling algorithms in programmable switches to efficiently aggregate index messages. Furthermore, NetSHa proposes a heuristic sort-reduce approach that can filter out poor candidate answers on programmable switches while preserving the search quality. Due to the limited on-chip memory of programmable switches, we further introduce a best-effort replacement mechanism so that NetSHa can accelerate more concurrent query tasks. Note that NetSHa also works on a hierarchy of programmable switches. NetSHa has been implemented on Barefoot Tofino switches [23]. We evaluate NetSHa across 7 real-world datasets and show that NetSHa reduces the packet volume by about $4 \sim 10$ and improves the search efficiency by over $3\times$, in comparison with software-based TLSH [14] and Layer-LSH [17] distributed search frameworks. In addition, multi-stage switches can cooperatively achieve better performance improvements, showing NetSHa's high scalability.

To the best of our knowledge, we are among the first to exploit in-network computation to accelerate LSH-based distributed search. To sum up, our contributions are threefold:

- We propose an in-network computing paradigm for distributed search, and develop NetSHa which utilizes programmable switches to improve search efficiency.
- We design a heuristic sort-reduce approach to reduce poor candidate answers, while preserving search quality. We also develop a scheduling algorithm for packets aggregation on programmable switches. In addition, we propose a best-effort replacement mechanism to improve the concurrency of NetSHa.
- We implement NetSHa on programmable switches and extensively evaluate it with different datasets. The results demonstrate NetSHa's efficiency.

This paper builds on our prior work [1], where we presented the basic in-network answer reduction and aggregation scheme. Here, we expand our analysis of the scalability of NetSHa (relevant to multi-stage switches), the adopted "best-effort" concurrency mechanism, as well performing more experiments using larger real-world datasets.

The rest of the paper is structured as follows. Section 2 presents the background and motivation, whereas Section 3 describes the design of NetSHa. We discuss the practicality of NetSHa in Section 4 and evaluate it in Section 5. Section 6 surveys related work, and we conclude the paper in Section 7.

2 Background and Motivation

In this section, we first briefly review LSH, and then present a typical LSH-based distributed search system and its query processing, highlighting potential performance problems. We finally introduce in-network computation that underlies our approach.

2.1 Overview of LSH

LSH is widely recognized as one of the most effective methods to index similar items in high dimensional space. It is described as follows.

Definition 1. *Let D(p, q) denote the distance of point p and q. A hash function family* $H = \{h : R^d \rightarrow U\}$ *is (*P_1*,* P_2*,* r_1*,* r_2*)-sensitive, if for any* $p, q \in R^d$*, they satisfy two conditions:*

$$(1)\, if\ D(p,q) \leq r_1,\ then\ Pr[h(p) = h(q)] \geq P_1;$$

$$(2)\, if\ D(p,q) \geq r_2,\ then\ Pr[h(p) = h(q)] \leq P_2.$$

For any one d-dimensional point $p \in R^d$, it can be hashed to a value $u \in U$ through a LSH function $h \in H$. This hash function holds two properties. First, two similar points can be hashed to the same value with a high probability of at least P_1 if their distance in the d-dimensional space is less than or equal to r_1. Second, two dissimilar points have a low probability of at most P_2 to be hashed to the same value if their distance is farther than or equal to r_2. Obviously, a useful LSH function should enable $r_1 < r_2$ and $P_1 > P_2$. Some common LSH families have been proposed based on different types of "distance", such as MinHash for Jaccard distance, E2LSH for Euclidean distance and SimHash for Angular distance.

Generally speaking, for a d-dimensional point $p \in R^d$, k ($d > k > 0$) LSH functions (i.e. h_1, h_2, ..., h_k) are randomly selected to respectively perform hash computation so that k values are generated. Then the generated hash values are concatenated to form a k-dimensional vector that represents the signature of point p, denoted as $S(p)$ = ($h_1(p)$, $h_2(p)$, ..., $h_k(p)$). In addition, L ($L > 1$) hash tables are generated, each of which is equipped with k LSH functions. Then point p obtains a k-dimensional vector signature for each hash table. As a result, it finally can contain L signatures (*i.e.* $S_1(p)$, $S_2(p)$, ..., $S_L(p)$). It is clear that the more signatures point p has, the higher probability it is indexed by another similar item.

LSH is typically conducted in two phases. The first phase is data pre-processing. The target of this phase is to insert data into hash tables. Specifically, one data item needs to be projected into a k-dimensional signature through k LSH functions for the i-th ($1 \leq i \leq L$) hash table. Based on the signature, the item is inserted into the corresponding bucket in the i-th hash table. Finally, this data item is inserted into all the hash tables. The second phase is data querying. Similar to pre-processing, a query first needs to figure out its signatures by using the same LSH functions. With these signatures, it then locates its buckets in the hash tables, where the candidate answers are. It then evaluates these candidate items to obtain the final answers.

2.2 LSH-based Distributed Search System

To achieve high throughput and low response times, there have been recent attempts to distribute LSH across multiple machines. Most LSH-based distributed search systems adopt two main approaches: MapReduce and Active Distributed Hash Tables (Active DHT). Both frameworks process data in the form of (Key, Value) pairs. MapReduce is a simple computation model based on two user-defined functions (*Map* and *Reduce*), while Active DHT is a distributed store that enables real-time data processing.

These LSH-based distributed search systems are deployed clusters that store hash tables and data contents (*e.g.* images, videos, text). Each hash table associates a (Key, Value) pair to each data item, where the Key is the signature[1] of this data and the Value is the pointer (or data ID) to its contents. These (Key, Value) pairs are stored over distributed servers so that the pairs with the same Key are located in the same or nearby machines.[2] When a query arrives, the search system computes its signature and interrogates a set of hash tables located in different servers. Therefore, a set of candidate answers are generated, *i.e.* (QueryID, Results) pairs. These must then be transmitted to a centralized agent. The tasks of the agent is to combine and rank these candidates to obtain the final answers.

Performance bottlenecks. One query may generate a large set of candidate answers from different servers. These candidates must be transmitted to the centralized agent(s). In addition, a useful distributed search system *must* support tens of millions of concurrent queries (*e.g.* imagine performing search on a popular website). As a result, network congestion is inevitable, which in turn will have a deleterious effect on performance. For example, our experiments show that network communication accounts for about $25 \sim 30\%$ of the overhead per query in our small distributed experiment (see Section 5.2). For larger scale experiments, the percentage will be even higher. Thus, we believe that there are significant benefits to be gained by *reducing network communication overheads* for LSH-based search systems.

2.3 In-Network Computation

There are various ways to reduce network communication. A straightforward solution is to encapsulate multiple candidate answers into one packet at the server side. While this approach can reduce overheads, it offers little benefit if candidate answers are on different servers.

Recently, many network switches have become able to provide computational capacity. This allows tasks to be offloaded to network devices (from end-servers). There are many examples of these programmable switches, *e.g.* Barefoot Networks' Tofino switch (a.k.a P4 switches [22]). P4 switches have a flexible parser and a match-action forwarding engine, allowing programmers to dynamically configure the switch to perform diverse control functions. To process packets at high speed, the architecture of P4 switches have two multi-stage pipelines: ingress and egress. Each pipeline stage has a fixed amount of time to process packets in memory (TCAM and SRAM). Moreover, the switch supports certain Boolean and arithmetic operations using a set of ALUs.

We argue that this offers a powerful foundation for streamlining LSH performance. Thus, we propose NetSHa, an in-network computing approach that exploits the computational capacity of programmable switches to reduce the cost of network communication. Specifically, NetSHa offloads candidate query answer reduction and aggregation onto programmable switches. We choose this approach for three reasons. First, programmable switches are very common today. In modern data centers, programmable switches have been widely deployed. Second, programmable switches provide a friendly programming environment — programmers can use high-level programming languages to customize the logic of hardware. And third, programmable switches connect to servers directly, and they can "see" the distributed candidate query answers. As we will see later in this paper that by offloading some computation tasks onto programmable switches, both the network communication overhead and the processing time on the agent are significantly saved because of reduced packet volumes as well as the improved quality of answers that are sent to the agent. The rest of the paper present NetSHa, built upon the capabilities of P4 switches.

3 Design of NetSHa

In this section, we first present a high-level overview and discuss the challenges of NetSHa. We then describe the specifics of NetSHa in detail.

3.1 Overview of NetSHa

Overall, NetSHa follows the architecture of a typical LSH-based distributed search systems. We propose two key groups of optimizations using in-network computation: (*i*) answer reduction, and (*ii*) answer aggregation. Figure 1 shows the framework of NetSHa. Next, we briefly describe these optimizations.

Primer. Logically, NetSHa utilizes one programmable switch to connect to multiple servers, each of which may contain multiple candidate answers for the same query. These candidate answers are encapsulated in IP packets and transmitted to the agent server via a shared switch. We posit that we can exploit this shared switch to analyze, reduce and aggregate candidate answers. We illustrate how NetSHa achieves this in Figure 1. For every incoming packet from a server, NetSHa parses its data pairs (detailed in Section 3.3), and then identifies its query IDs. For the same query candidate answers, they will be dispatched to the same (logical) register (a.k.a on-chip memory), and operated on by NetSHa as follow.

Candidate answer reduction. NetSHa offloads part of the reduction task from the agent server to the programmable switch. Specifically, NetSHa checks each candidate answer and decides whether its distance to the query is small enough (*i.e.* similar to the query). Candidate answers with a small distance will be transmitted to the agent, while the others should be preserved in the register (for further comparison) or dropped directly. Consequently, poor quality

1. This signature is computed by the specific LSH functions associated with this hash table.

2. Considering the accuracy of LSH, similar data items may be located in different servers.

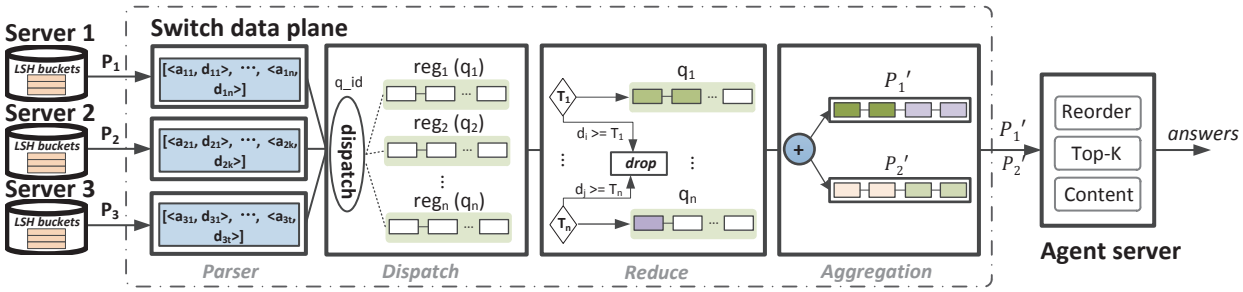


Fig. 1. The framework of NetSHa. In this figure, P_i refers to an answer packet from server i while q_i represents the i-*th* query. A slot (rectangle) with white color means empty. The slots with the same color carry candidate answers for the same query.

answers will be removed on the switch, which can alleviate the computation load of the agent server and improves the search efficiency.

Aggregating answer packets. Candidate answers, which should be transmitted to the agent server, will be aggregated into one or more packets based on a customized protocol (see Section 3.3). Furthermore, NetSHa aggregates those candidate answers belonging to different queries into one packet. This answer aggregation can significantly reduce incast traffic[3] and reduce the I/O cost of the agent servers.

Summary. In short, on receiving a query request, NetSHa selects the servers to be interrogated for the query. Then each selected server returns candidate answers. Next, these candidate answers are passed to the centralized agent (via the network) in order to produce the final answer list. Any switches that observe answers for the query, perform in-network reduction and aggregation. In this way, NetSHa reduces communication cost and improves the search efficiency.

3.2 Challenges

NetSHa has to overcome five challenges as follows:

1) First, programmable switches are originally designed for network packet forwarding. NetSHa must enable point-to-point communication between distributed servers and centralized agents without affecting regular network functions (Section 3.3).
2) Second, though each candidate answer associates a *distance* value to refer to its similarity with the query, it is hard to define "poor quality" candidate answers for answer reduction. Thus, we need a way to quantify this so that programmable switches do not drop global-scope good answers, while also removing poor quality answers (Section 3.4).
3) Third, a programmable switch aggregates candidate answers by maintaining a status and data structure for each query task. However, the memory of programmable switches is limited. This means a programmable switch cannot conduct aggregation for many tasks simultaneously. NetSHa must employ effective scheduling to conduct aggregation so that it can be deployed in high throughput and high concurrency distributed environments (Section 3.5 and Section 3.6).
4) Fourth, answer aggregation on the programmable switches should improve the throughput of the system, but should also improve the response time for end-users. Thus, we must find the right balance between throughput and latency (Section 3.7).
5) Fifth, it is not necessarily feasible that all servers connect to one switch running NetSHa, or that all traffic is routed through this switch. To this end, NetSHa should support multi-stage switches that are widely adopted in data centers (Section 3.8).

The remainder of this sections describes how we address the aforementioned challenges.

3.3 Network Protocol Extension

To support answer aggregation on programmable switches, NetSHa performs a simple extension of the Internet Protocol (IP), so that the programmable switches can efficiently distinguish candidate answer packets. For simplicity, we refer to this as the NetSHa protocol. Figure 2 shows the packet format. We use the reserved bit in the IP Type of Service (ToS) field to identify NetSHa packets (*i.e.* packets that contain candidate answers).

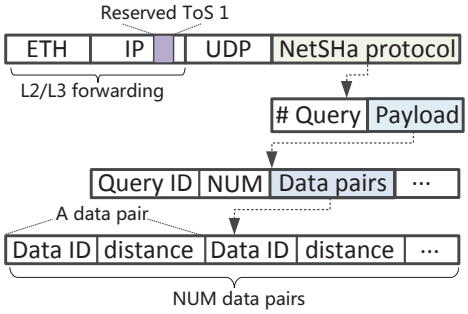


Fig. 2. Packet format in NetSHa.

NetSHa protocol packet. A NetSHa packet, whose IP ToS reserved bit[4] is set to 1, is encapsulated as a User Datagram Protocol (UDP) message. This is widely used for customized protocols in data center networking [24], and does not affect the traditional IP network. A NetSHa message consists of two portions: (*i*) NetSHa header and (*ii*) its payload. The

3. Packets are from multiple input ports to one output port.

4. Distributed search systems are always deployed in data centers so that we can customize packets and reuse some reserved bits without affecting regular packets.

NetSHa header contains Query # (1 byte), which refers to the number of queries its payload contains. For each query, it contains Query ID, NUM and an array of data pairs. The Query ID has 4 bytes and is used to identify the query, while NUM has 2 bytes and indicates the number of answers (data pairs) for the query encapsulated in the packet. Each data pair is composed of a 4 byte data ID (a.k.a answer ID) and a distance value (8 bytes), denoted by $\langle a_i, d_i \rangle$. The distance value is used to measure how far away the answer data is from the query data. A standard Ethernet packet can contain up to 1,500 bytes, so one NetSHa packet is able to carry up to 120 candidate answers.

Note that the programmable switches will only intercept and process NetSHa protocol packets, whereas other packets will be forwarded directly. Thus, NetSHa does not affect regular network functions. Note that the switch parser needs to parse the packets to extract the data pairs for subsequent comparison.

3.4 Answer Reduction

To restate, the primary mission of the agent server is to receive candidate answers from one or more servers, reorder them and select the Top-K answers. This process is known as "answer reduction", which consumes significant computational resources on the agent server. As the scale of the distributed search system increases, the agent server inevitably becomes a bottleneck.

To this end, NetSHa offloads the reduction task from the agent server to the network by using programmable switches to drop poor quality answers. This offloading is feasible since programmable switches can "see" all candidate answers that will arrive at the agent server and utilize the distance value associated with each answer to decide their quality. This is in contrast to individual servers, which only have a local perspective. Nevertheless, it is challenging to identify poor candidate answers which should be dropped on programmable switches.

Basic approach. The basic solution to address the above challenge is to set up a threshold (T) for each query in the switch. All candidate answers whose distance value is larger than the threshold, should be dropped directly. Otherwise, they should be forwarded to the agent server for further processing. Obviously, the threshold parameter is crucial. To this end, we count the number of answers for the same query that the switch receives, and forward the first K candidate answers to the agent server directly, while recording the worst answer among them. Here, K is the number of answers that should be returned to users by the agent server (*i.e.* the top-K answers). Thus we can determine the value of K ahead of time. Here, we can use the distance value of the worst answer to configure the threshold, T. The candidate answers, whose distances are smaller than T, are aggregated and transmitted to the agent server (see Section 3.5).

Though this basic approach is straightforward, its efficiency is low. In the worst case, it does not reduce any poor candidate answers if the first K answers received at the switch contain the globally worst ones. In other words, the worst answer is used to configure the threshold.

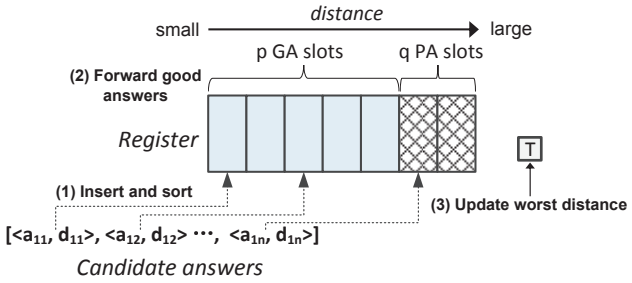


Fig. 3. Overview of the sort-reduce approach.

Sort-reduce approach. To address the above shortcomings, NetSHa proposes a sort-reduce approach to reduce poor candidate answers while preserving search quality. This is done using registers of switch's memory. Specifically, we associate with each query a *logical* register that is similar to an array and contains limited slots (*i.e.* array entries). It is worth noting that, as we will describe in Section 4, a logical register spans over multiple physical registers in the switch, in order to allow multiple read and write operations on the logical register in one pipeline stage. To ease the description, registers in this paper refer to logical registers unless otherwise specified. A logical register contains several slots to store its corresponding candidate answers, where smaller sequence number slots store better candidate answers (see Figure 3). With this basis, NetSHa divides the slots of each register into two parts: good answer (GA) and poor answer (PA) slots. Each register has p GA slots and q PA slots.

When a batch of candidate answers for one query arrives, NetSHa will insert them into the corresponding register based on their distance values and also keep the order property. If all GA slots are occupied, NetSHa will perform an aggregation operation for the GA slots (see Section 3.5), send the p GA answers (the local top-p answers) to the agent server and update the worst answer it sent (*e.g.* the threshold T in Figure 3). Otherwise, it begins to process the next batch of candidate answers. The above iteration will stop when K answers for the query have been sent to the agent server. At this moment, we can safely clear all the remaining candidate answers in the registers for this query as they are useless to send to the agent. NetSHa then resets the threshold T as the largest distance value among the answers that have been sent to the agent.

NetSHa then moves to the second stage, where q is set to 0 — we do not need PA slots anymore because we now have the threshold T. The subsequent answers with distance values larger than T are dropped because they are worse than the top-K answers that have been sent. Those with distance values less than T, on the other hand, are inserted into the register for the corresponding query. The answers in the register will be sent to the agent when it is full or when the timer expires (see Section 3.7). The pseudo-code for the above sort-reduce approach is listed in Algorithm 1.

The highlight of our sort-reduce approach is to utilize PA slots to store poor answers (till now) and avoid the drawback of the basic solution. This ensures that the first K answers sent to the agent server do not involve the globally bottom-q answers (the q worst answers), where q is the number of PA slots. Considering the limited memory

of programmable switches (*e.g.* tens of megabytes on-chip memory), we assume $K > (p+q)$. For a register with a fixed size (*i.e.* $p+q$ is fixed), a smaller q results in a lower reduction efficiency. In the extreme case where $q=0$, our sort-reduce approach will degenerate into the basic solution. A larger q, on the other hand, will degrade the effect of answer aggregation described in Section 3.5 as p (the size of GA slots) becomes smaller. Our experimental results in Section 5 confirm this intuition.

To implement ordered candidate answer insertion into the register, we borrow ideas from Bubble Sort [25] to achieve our sort-based mechanism (denoted as $Slot\_Sort()$ in Algorithm 1). Specifically, for each input candidate answer a_t, it should check the corresponding register slots (see Figure 3) from right to left. In other words, the last slot in PA, which stores the current worst answer, should be considered first. If the slot is empty, a_t will be inserted directly. Otherwise, a_t needs to be compared with the answer stored in the slot (denoted as a_s). If the distance of a_t is larger (*i.e.* a_t is worse), a_s will be backed up first, and then replaced by a_t. In this case, we need to find a proper slot for a_s, where its distance is larger than the answer in the left slot but smaller than that in the right one. If the distance of a_s is larger, a_t moves to the next slot left until it finds a proper slot to be inserted into. The complexity of the above insertion is thus $\mathcal{O}(n)$, where n is the number of the slots (*i.e.* $n=p+q$). We will detail how to implement this algorithm on programmable switches in Section 4.

Theoretical analysis. Next, we discuss the theoretical benefits brought from the answer reduction. Let us consider d servers and one agent. Each server returns n locally sorted optimum candidate answers for one query, while the agent finally selects the top-K answers from the $n \times d$ candidates. NetSHa filters out the globally worst answers via answer reduction. The best case for the answer reduction is that the threshold is configured as the distance of the globally $(k+1)$-best answer. In this case, $(n \times d - k)$ candidate answers would be reduced. For the worst cast, the threshold is configured as the distance of the globally $(q+1)$-worst answer. This is because NetSHa allocates q PA slots to store and hide globally bottom-q answers. In other words, it can reduce q answers at least. As to the overhead, NetSHa needs to insert candidate answers in the correct order into the registers to configure the threshold; the worst-case complexity of this insertion operation is $\mathcal{O}(n)$. Once the threshold is configured as the distance of the observed worst answer, the insertion operations will no longer happen. Instead, NetSHa directly utilizes the threshold to filter out the subsequent worse answers on the fly, whose overhead is low. As such, the overhead introduced by NetSHa is negligible; our experiments also confirm it (see Table 3).

Algorithm 1: SORTREDUCE(*a, r, n, p, K*)

Input: a, a candidate answer
Input: r, a register
Input: n, the number of slots in r
Input: p, the number of GA slots
Input: K, the number of top answers the agent should return to users.

```
if r.k < K then
    // r.k: the number of packets that have been sent
    pos ← Slot_Sort(a, r, n);
    //Insert a and get the insertion position pos
    if pos ≡ 0 then
        // register r is full
        if (K − r.k) ≥ p then
            Send(r, p); // Send all the answers in GA slots
            r.k ← (r.k + p);
            Set(r, p, r.T);
            // Update r.T with the worst distance
        else
            Send(r, K - r.k);
            // Send the top (K - r.k) answers in GA slots
            r.k ← K;
            Set(r, K - r.k, r.T);
            Clear(r);
            // Clear all the remainder candidate answers in r
else
    if a.d ≥ r.T then
        Drop(a); // Drop subsequent worse answers directly
    else
        pos ← Slot_Sort(a, r, n);
        if pos ≡ 0 then
            Send(r, n);
return;
```

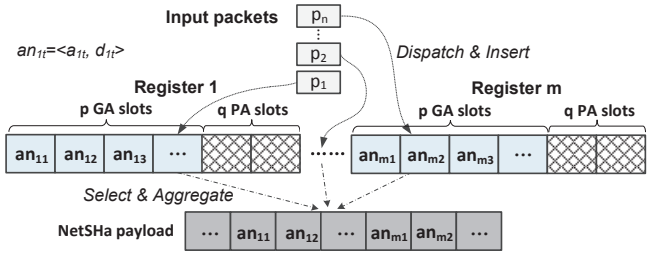


Fig. 4. Overview of the answer aggregation.

3.5 Aggregation in Programmable Switches

We next present another major component of NetSHa — the aggregation of multiple candidate answers into one or several packets on the programmable switches. The aggregation aims to decrease the volume of transmitted packets. Indeed, the answer aggregation can mitigate network congestion (*i.e.* in-cast traffic) and reduce the I/O cost of the agent server.

Overview of answer aggregation. NetSHa aggregates candidate answers on programmable switches (see Figure 4). The switches receive a batch of input packets from distributed servers, parses their carried candidate answers and dispatches them into registers. NetSHa then performs answer reduction. Next, NetSHa aggregates the stored candidate answers into one or more NetSHa packets (if the an-

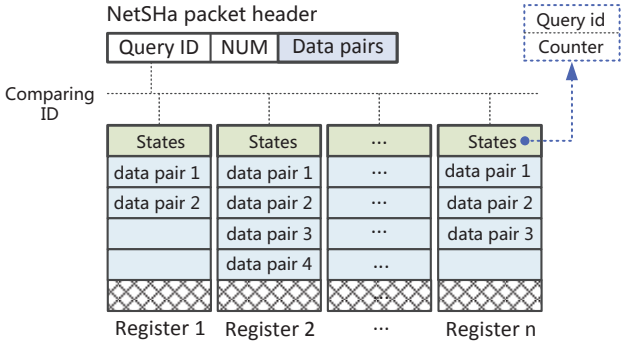


Fig. 5. The data structure for packet aggregation in programmable switches.

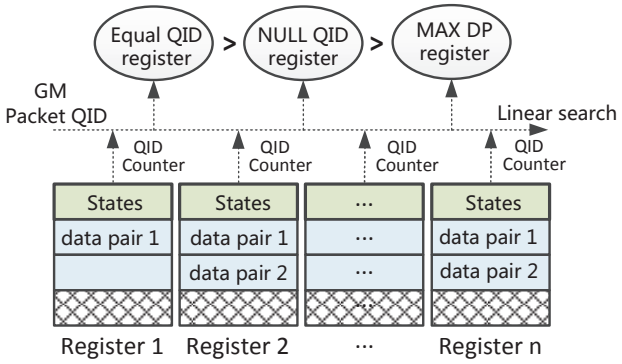


Fig. 6. The register selection for the input NetSHa packet. QID is short for query ID and GM refers to global maximum value; DP is short for data pair; $>$ represents the selection priority.

swers need to be sent). Finally, these packets are transmitted to the agent server.

Register data structure. The switch performs the lightweight packet aggregation using registers (see Figure 5). To achieve the aggregation task, the switch initializes and maintains a global "two-dimensional array" based on its registers. Each register stores two types of data: *states* and *data pairs*. The states record a Query ID for identifying a specific query, and a counter for denoting the number of carried data pairs in this register. Data pairs (candidate answers) are the results of the corresponding query.

When a NetSHa packet enters the switch, it will select one register to insert its data pairs to. Our existing implementation adopts a linear search to determine the register. If there is already one register with the same Query ID, the packet appends its data pairs to the tail of the register until it is full. And the insertion follows our heuristic sort-reduce approach (Section 3.4). Otherwise, NetSHa first finds an "empty" register and inserts the data pairs in the packet, and then sets the register states, including the corresponding Query ID and counter value. Once a register is full, NetSHa will send the data pairs in GA slots to the agent following the sort-reduce approach, and reset the states correspondingly.

Basic register replacement. The number of registers in a switch determines how many aggregation tasks (queries) it can perform in parallel. Unfortunately, for financial reasons, the number of registers is limited. This means that a NetSHa packet with a new Query ID cannot be processed if all registers are occupied. To address this problem, we adopt a *replacement strategy* in order to select an appropriate register.

This strategy is a weight-based selection mechanism. Put simply, the register that has carried the most data pairs, will be selected if all registers are occupied. As shown in Figure 6, a NetSHa packet visits the registers one by one. It compares its Query ID with that of the register. If they are identical, the register is returned. Otherwise, it will traverse all registers to record the first "empty" register and update the register with the carried data pairs. If an "empty" register is found, this register will be returned. Otherwise, the register with the *most* carried data pairs will be selected (called the replacement register). This procedure is illustrated in Algorithm 2. To avoid losing data, if a register is replaced, it must first aggregate its existing data pairs, construct a NetSHa packet via modifying one current input packet header and padding its payloads. After the checksum has been recalculated and updated by the switch, this packet will be transmitted to the agent. Following this, the register can be cleared and used for the next incoming NetSHa packet. We note that, besides the basic replacement strategy, there are alternative approaches to select an appropriate register (*e.g.* a time-to-expiration mechanism). We leave exploring these alternatives to our future work. It is also noteworthy that we do not adopt a bypassing-forwarding (BF) mechanism that directly forward the candidate answer packets when there is no empty register available. The reason is that while this mechanism avoids register replacement, poor answers carried by the bypassing-forwarding packets cannot be reduced by switches, degrading the efficiency of in-network processing. Our experiments also confirm this.

Algorithm 2: WEIGHTSELECTION(q, R, n)

Input: q, the arrival query id
Input: R, a vector of registers
Input: n, the number of registers in R
Output: a selected register
$weight \leftarrow 0$;
$slot_r \leftarrow$ **NULL**;
$slot_e \leftarrow$ **NULL**;
for $i = 0$ *to* $n - 1$ **do**
  if *R[i].q* == q **then**
    `return` R[i];
  if *R[i].q* == -1 && $slot_e$ == **NULL then**
    $slot_e \leftarrow$ R[i];
  if *R[i].w* $>$ $weight$ **then**
    $slot_r \leftarrow$ R[i];
    $weight \leftarrow$ R[i].w
if $slot_e \neq$ **NULL then**
  `return` $slot_e$;
`return` $slot_r$;

3.6 Concurrency of NetSHa

In the worst case when there is no register available, NetSHa directly forwards the answer packets from servers to the agent without any reduction or aggregation. That said, NetSHa does not degrade the overall concurrency of the search system (measured by the number of query tasks that can be "run" simultaneously). Nevertheless, the available on-chip memory that NetSHa can use impacts the performance improvement by our approach.

The weight-based selection mechanism that we propose in the last subsection enables the switches to replace query tasks. Yet it does not decide whether a new query task needs to trigger the replacement. For example, considering tens of millions of concurrent query tasks, it is possible that some tasks are replaced out even if they were only recently inserted into the registers.

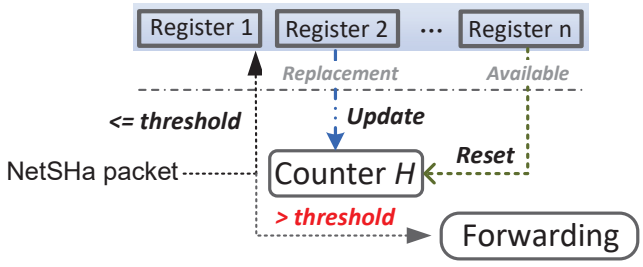


Fig. 7. A best-effort replacement mechanism.

To fix this issue, NetSHa introduces a "best-effort" mechanism (see Figure 7). More specifically, it utilizes a counter to record how many times register replacements have been triggered. As long as it exceeds a threshold value, H, the subsequent replacement operations will not be triggered. The corresponding answer packets are forwarded directly without reduction and aggregation; these will be processed by the next switch or the agent. Note that the counter will be reset back to 0 once there is an available register (*i.e.* no longer be used by any query task). As the threshold affects the number of query tasks that can be potentially accelerated by NetSHa, we define $\lambda = H \div$ #registers as the concurrency parameter (λ) of NetSHa. A larger λ means more tasks can be accommodated by a switch for answer reduction and aggregation at the cost of possibly more caching misses due to frequent replacements. On the other hand, a smaller λ may reduce the benefits of NetSHa because poor answers carried by newly arrived packets for other query tasks cannot be reduced by the switch. We evaluate the impact of λ on the performance gain in Section 5.8.

3.7 Achieving Low Response Time

NetSHa aggregates packets in programmable switches, however, this must be balanced against response time. Naturally, the longer the packet is retained within the switch, the longer the response time is. To address this challenge, we equip each register with a timer.[5] Once the timer expires, NetSHa *must* construct an aggregated packet for the query task that this register holds, regardless whether the number of data pairs reaches the threshold.

5. Timer is a very common component in programmable switches.

An implementation challenge here is that programmable switches cannot generate new packets by themselves. Consequently, the candidate answers in the register cannot be aggregated and transmitted even when its timer expires if there is no input packet. To address this challenge, NetSHa leverages *packet recirculation*, a feature of programmable switches, to let one packet always traverse the switch pipeline. Specifically, packet recirculation enables one packet, which has already passed through the switch pipeline, to re-enter the pipeline. Thus, NetSHa can capture and clone the traversed packet[6] for aggregating and transmitting candidate answers if there is no available input packets. It is noteworthy that at most one recirculation packet is needed for a register, and the packet payload can be empty — it is used only for triggering the answer aggregation if needed. Thus, the overhead is very limited. For example, for a packet with 64 bytes and 100K registers, it only consumes 6.4MB bandwidth at most, which is very small compared to the 3.2Tbps [23] throughput of switches. Note that recirculating packets in programmable switches has been used in many works [26].

3.8 NetSHa with Multi-stage Switches

Recall that a programmable switch parses and identifies NetSHa packets for reducing poor candidate answers while other packets are forwarded directly. Good answers are selected and aggregated into new NetSHa packets that are transmitted to the agent server. Thus, both the input and output of the switch is NetSHa packets. This means that NetSHa can easily be extended to multiple switches.

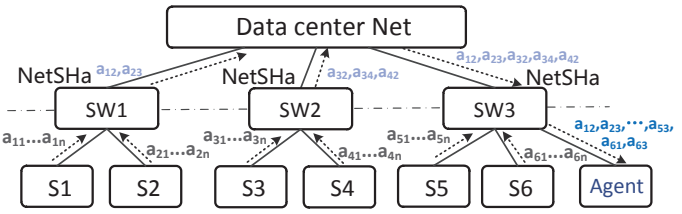


Fig. 8. NetSHa with multiple switches. SW refers to a programmable switch while S represents a server. a_{ij} denotes the j-th candidate answer from the i-th server.

Figure 8 shows an example of how switches can work together without requiring explicit coordination. Each ToR (Top of Rack) switch[7] runs NetSHa to reduce candidate answers. For example, $SW1$ reduces poor candidate answers from server $S1$ (a_{11},...,a_{1n}) and $S2$ (a_{21},...,a_{2n}) while the good answers (a_{12} and a_{21}) are aggregated and transmitted to the agent. Finally, $SW3$, which connects to the agent, further filters out poor answers from the network (*i.e.* $SW1$ and $SW2$ ToR switches) and its local servers ($S5$ and $S6$). This example highlights that switches do not need to negotiate with each other—they can run NetSHa independently.

This brings two benefits: (*i*) it does not require packets to be routed to a specific switch so that packets with candidate answers are reduced and aggregated on the fly; and (*ii*) through multi-stage switches deployed with NetSHa,

6. The original packet will continue its recirculation after the clone.
7. Though we here only consider ToR switches, it equally applies to other types of switches.

the quality of the candidate answers arriving in the agent server can be improved. Intuitively, multi-stage switches collectively provide more on-chip memory available for use by NetSHa. The poor answers that are not reduced by the switches in earlier stages can be further captured by those in the late stages for further reduction and aggregation. Besides, switches in the later stages may have a better coverage of answer servers, leading to more precisely reduction of poor answers.

4 Practical Considerations

In this section, we discuss several aspects of NetSHa's practicality.

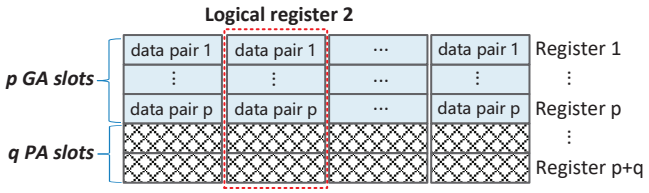


Fig. 9. Mapping between logical registers and physical registers.

Programming restrictions. Note that programmable switches have a number of restrictions, particularly in terms of physical register access. Indeed, one physical register in a stage can only be read and written once per packet processing, which makes it difficult to deploy our algorithms (*e.g.* $Slot\_Sort$) that may require multiple read and write operations of the answer slots. To overcome this barrier, we adopt a virtual memory mapping mechanism. Note that a query task is associated with a logical register with p GA slots and q PA slots. We map each slot to one physical register. That said, a logical register spans over $p+q$ physical registers (see Figure 9). For a query, we hash the query ID into a specific value, k, and use all the k-th slots of the $p+q$ physical registers to form its logical register. In doing so, we can perform multiple read-write operations on a logical register in one stage.

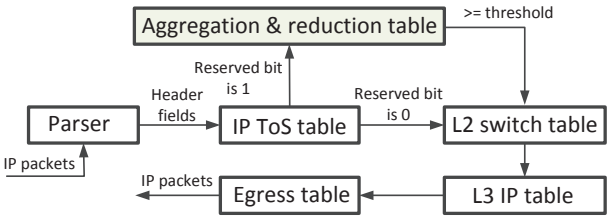


Fig. 10. Data flow graph in programmable switches.

Switch pipeline processing. Most modern programmable switches adopt a multi-table pipeline model to process packets (this is supported by Barefoot Tofino switches). Using the open programming interfaces, developers are able to customize the switch pipeline and configure the pipeline tables. NetSHa relies on such switches. Figure 10 presents the pipeline processing of the programmable switches for answer reduction and aggregation. Specifically, the programmable switch receives IP packets from physical ports and parses them into a vector of header fields based on its pre-configuration state automata. Next, it configures a table (IP ToS Table) to identify NetSHa packets whose IP ToS reserved bit is 1. For the NetSHa packets, they need to jump to the Aggregation & Reduction table for further processing (for answer reduction and aggregation). The other packets, whose IP ToS reserved bit is 0, are viewed as regular packets and subsequently forwarded as normal.

We use a pre-programmed matching rule in the Aggregation & reduction table. The table only contains one rule that will process all packets that enter this table. The table rule has a wildcard match pattern while its action behavior is customized via composing a few action primitives. Specifically, it first performs a hashing operation to obtain a query ID, and then uses it to locate the logical register. Finally, it runs our $Slot\_Sort$ algorithm to reduce bad candidate answers, and further aggregates good candidate answers. We have implemented NetSHa with the above pipeline in Barefoot Tofino switches (3.2 Tb/s).

Search quality. We emphasize that NetSHa does not change any functional properties of the LSH family. Instead, it only relies on common LSH functions (*i.e.* TLSH functions) to compute similar data items. Therefore, NetSHa preserves the theoretical basis of its adopted original LSH functions in terms of search quality. As a result, NetSHa can achieve the same search quality as typical LSH-based search. This is also confirmed by our experiments (see Section 5.6).

Fallback . NetSHa can also fall back to the case where candidate answer packets are directly forwarded without answer aggregation and reduction via manipulating flow table rules to change their pipeline processing paths.

Coexistence with regular network functions . NetSHa does not affect regular network functions (*e.g.* forwarding) as it only operates on NetSHa packets. Other regular packets are forwarded as normal. NetSHa can also adjust the register resources it uses, in order to guarantee that regular network functions operate as usual.

Benefits of NetSHa . NetSHa offloads the computational load from agent servers to programmable switches. This offloading is both feasible and beneficial as programmable switches are wildly available in data centers. Indeed, there are often free computational resources on programmable switches [27], [28] , which can be leveraged for in-network computation. As we will show through experiments that such a offloading reduces the packet volumes and improves the search efficiency. As a further benefit, it has also been shown that offloading some computational tasks from the server to the network can improve power-efficiency [29].

Query server selection. In a typical LSH-based distributed search framework, for a new query task, the agent needs to first compute and select those servers that may store candidate answers to avoid answer search on useless servers (*i.e.* those holding answers far from the query). Other than conventional solutions that adopt a software-based approach on agent servers, NetSHa also offloads this task to programmable switches via a table-lookup mechanism in order to facilitate the selection of servers on the fly. That said, this offloading enables hardware-based implementation of the query server selection.

5 Experimental Evaluation

In this section, we evaluate NetSHa. As a baseline, we compare it against two alternative solutions: (*i*) Layered-LSH [17], a conventional distributed MapReduce-based variant of the LSH algorithms; and (*ii*) A software-based TLSH [14] algorithm implementation for generic platforms. Different from the layered-LSH that uses conventional LSH functions, the TLSH adopts a ternary matching mechanism to accelerate the identification of similar answers on individual servers. NetSHa uses the same mechanism as TLSH on servers.

5.1 Evaluation Methodology

Testbed Setup. We evaluate NetSHa on a testbed with 6 servers, each of which is equipped with 32 cores of Intel(R) Xeon(R) E5-2682 CPU @ 2.5GHz, 256GB RAM with Ubuntu 16.04 and Linux kernel 4.15.0-132. All servers are directly connected to a Barefoot Tofino switch (3.2 Tbps). Additionally, we run 4 virtual machines (VMs) on each physical server where each VM monopolizes 4 CPU cores and occupies 32GB memory.

Workload and datasets. The experiments involve four different types of datasets.

- **Random.** This dataset is constructed by sampling points from the normal distribution, $N^d(0,1)$ with $d = 100$. The dataset contains 200M data points. The queries are generated by adding a small perturbation to the positive distribution $N^d(0,\sigma)$ with $\sigma = 0.3$. In total, we generate 10M queries. This type of dataset has been used in many LSH experiments [14], [17], thus we also use it to solve the (c, r)-NN problem with $c = 2$. The expected distance for each query to its closest data point is $r = 0.5$, and there is a high probability that the data point is within the distance $c * r$.
- **Text.** We use three real-world text datasets: Wiki [30], Enron [31] and Glove [32]. For the Wiki dataset, we use the English Wikipedia corpus of July 2019 for calculating the corresponding TF-IDF vectors. The Wiki dataset contains 200M data points. In addition, we use two larger datasets, Enron and Glove, where both of them contain about 1000M data points. Enron originates from a collection of emails, while each Glove data item consists of 100-dimensional word feature vectors extracted from Tweets. For the three text datasets above, we generate 10M queries with $d = 100$.
- **Image.** We adopt two image datasets. One is a Corel image collection [17] where we directly use the 32-dimensional color histogram extracted from each image to obtain 210M data points. The other is a larger Deep [33] dataset, which contains deep neural codes of natural images obtained from the activations of a conventional neural network. This generates 1,000M 256-dimensional data points with 10M queries.
- **Audio.** The audio [34] dataset consists of 192-dimensional audio feature vectors extracted using the Marsyas library from the DARPA TIMIT audio speed dataset. It consists of 1,000M data points and 10M queries.

For each experiment, we use the above datasets to evaluate NetSHa and report the average along with the 95% confidence intervals values if possible. In each run, we issue 10M queries randomly selected from the query set of the used dataset. The default system parameters are listed in Table 1, where we use 100 registers with each having 12 slots in total and 8 slots for GA (p) on each switch. By default, the agent will return the top 50 answers (*i.e.* $K = 50$).

TABLE 1
The default setup of the system parameter.

# of registers	# of slots	p	K
100	12	8	50

Evaluation roadmap. We are particularly interested in five aspects: (*i*) the network overhead in typical LSH distributed search systems (Section 5.2); (*ii*) the performance improvement of NetSHa compared to the baselines (*i.e.* layered LSH [17] and TLSH [14]) (Section 5.3); (*iii*) the effect of each major component, *i.e.* reduction and aggregation (Section 5.4); (*iv*) the impact of parameters (Section 5.5); (*v*) the search quality (*e.g.* the recall rates) of NetSHa (Section 5.6); (*vi*) the scalability (Section 5.7) and the concurrency (Section 5.8) of NetSHa.

5.2 Network Overhead

We have previously argued that the overhead of network communication seriously affects the performance of existing search systems. Before continuing, we briefly wish to confirm this assumption. Thus, we deploy the Layered-LSH [17] and the software-based TLSH [14] in our testbed. For transport, we use UDP and evaluate each solution. We measure the processing time for each query, which includes four parts: the server selection time at the agent server[8], search processing time (*i.e.* hash bucket matching) on the distributed servers, network communication time and the processing time at the agent server (for candidate answer sorting). The total processing time for a query, t, is the interval between the agent beginning to operate a query task and returning Top-K answers. The server selection time t_0 and the processing time of the agent (t_1) are recorded locally on the agent; the search processing time on a distributed server i is the time interval between the query arrival time and the sending time of its corresponding answers (denoted by st_i). Finally, the network communication time is estimated as $t - t_0 - t_1 - \max_{i \in m}\{st_i\}$, where m is the number of distributed servers.

Table 2 reports the query time distribution (average over all queries) for individual queries, confirming that the network *does* contribute significantly to the overall query processing time. Specifically, the network contributes $25 \sim 30\%$ of total processing time. We also note the network contributes more for TLSH than that in Layered-LSH. This is because TLSH optimizes the hash buckets matching process at the server side. This lowers the processing time on the

8. Note that NetSHa also offloads this task onto programmable switches.

TABLE 2
Average query time distribution (%) and the average query time (μs) for individual queries.

Dataset	*Solution*	*Query time distribution (%)*				*Time (μs)*
		select.	search	net.	agent	
Random	Layered	15.24	33.76	26.76	24.24	2.97
	TLSH	14.64	28.81	30.37	26.18	1.49
	NetSHa	7.14	56.89	18.69	17.28	0.53
Wiki	Layered	14.61	31.86	27.95	25.58	2.58
	TLSH	13.51	28.63	30.17	27.69	1.32
	NetSHa	7.12	62.39	14.26	16.23	0.45
Enron	Layered	19.84	34.27	23.54	22.35	2.87
	TLSH	19.19	27.68	28.26	24.87	1.43
	NetSHa	6.67	56.13	17.78	19.42	0.55
Glove	Layered	14.99	32.17	28.33	24.51	2.93
	TLSH	14.38	27.49	31.26	26.87	1.49
	NetSHa	6.63	54.6	18.69	20.08	0.54
Corel	Layered	18.32	32.49	25.57	23.62	3.24
	TLSH	17.05	28.97	28.57	25.41	1.93
	NetSHa	9.73	58.17	16.64	15.46	0.67
Deep	Layered	15.97	32.84	26.57	24.62	3.57
	TLSH	15.2	27.86	30.53	26.41	1.84
	NetSHa	8.72	52.47	16.25	22.56	0.73
Audio	Layered	16.87	32.13	27.34	23.66	2.91
	TLSH	14.43	29.11	30.27	26.19	1.76
	NetSHa	8.75	55.72	16.64	18.89	0.7

distributed servers, which means that a larger proportion of the delay is attributed to the network. Thus, for distributed search systems with optimized servers, the network is indeed one of the major performance bottlenecks. To have an overview of NetSha's performance, we also report the average query time and the distribution over the four parts in the table. We can observe that the query time for NetSha is much lower than the two baselines. Next, we detail the evaluation of NetSHa.

5.3 Evaluating NetSHa

Next, we evaluate NetSHa and compare it with the two baselines. The setup is the same as Section 5.2. This set of experiments uses the default parameters shown in Table 1.

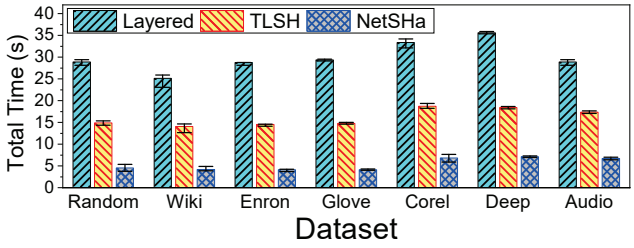


Fig. 11. The total processing time for 10M queries.

Figure 11 presents the experimental results. NetSHa achieves significant performance improvements compared with the two baselines (for all datasets). This is because the programmable switches drop poor answers and aggregate answer packets, thereby reducing the packet volumes and network congestion. Fewer packets imply shorter queues on the switch and agent, fewer I/O calls on the agent, and lower processing time on the agent.

Thus, answer reduction and aggregation not only benefit the network but also the agent. To better highlight the processing improvements on the agent, we count the number of

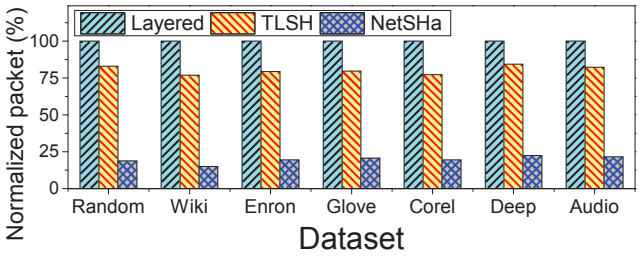


Fig. 12. The number of packets (normalized with the *Layered* solution) that the agent received.

packets that the agent receives when executing 10M queries as a proxy for the I/O and processing overhead on the agent. Figure 12 reports the results where we use the number of received packets in the Layered LSH as the baseline. We see that the number of packets is greatly reduced in NetSHa compared to Layered and TLSH. Figure 13 further examines how this impacts the processing time at the agent. We see that the processing time at the agent is in proportion to the number of received packets, and can be reduced by about 80% using NetSHa.

In summary, NetSHa reduces the packet volume by about 4∼10× and improves the search efficiency by over 3× compared with TLSH and Layered LSH. By reducing the packet volume to the agent and dropping poor answers in the switch, our approach reduces both the network communication overhead and the processing time in the agent. Besides, NetSHa also adopts a TLSH-like method to achieve fast answer search on individual servers, which reduces the processing time on distributed servers (compared with the Layered-LSH solution). Furthermore, a typical query task often needs to first compute and select those servers that may store similar candidate answers, in order to avoid answer search on useless servers. While both Layered-LSH and software-based TLSH solutions adopt a software implementation on the agent for the above server selection task, NetSHa offloads this task to programmable switches for facilitating fast selection of servers on the fly. In summary, as also showed in Table 2, the significant improvement on search efficiency by NetSHa is owned to the reduction of processing time related to the server selection, network communication and final answer processing on the agent.

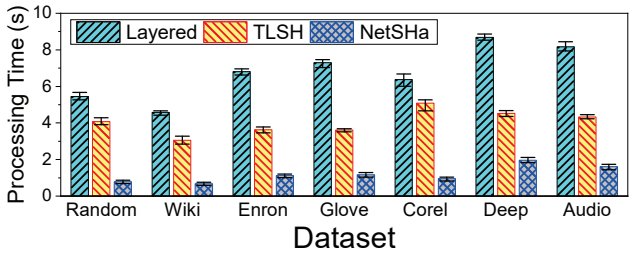


Fig. 13. The processing time of the agent server.

Next we show how NetSHa performs with different query workloads (*i.e.* different levels of network utilization). To this end, we scale up the query rates by utilizing multiple CPU cores for query generation (see Figure 14). Specifically, the baseline query rate ($QueryRate = 1.0$) is the case when

NetSHa only uses one core to generate query requests. To increase the query rate, we multiply the number of cores that take part in query generation. As expected, a higher query rate results in (slightly) lower performance improvement. This is because a higher query rate results in more cache misses because of more frequent register replacements. Figure 15 further compares NetSHa with the two baselines under different query workloads in the the *Deep* dataset[9]. Other than NetSHa, both Layer-LSH and TLSH are insensitive to the query rate because they do not have the issue of cache miss. Nevertheless, even with high query rates, the improvement of NetSHa is still significant.

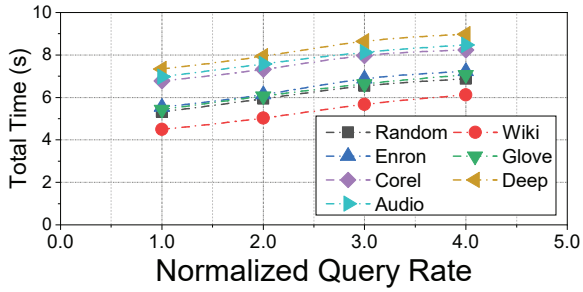


Fig. 14. Total processing time of NetSHa for 10M queries under different query rates.

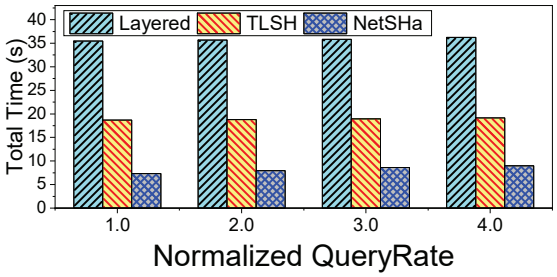


Fig. 15. Total processing time of three solutions under different levels of network utilization with the Deep dataset.

5.4 Breakdown Analysis

The above confirms that NetSHa achieves better search performance compared with the baselines (TLSH and Layered solutions). We then dissect the role played by NetSHa's two key components: answer reduction and aggregation. We again adopt the experimental setup in Table 1. In this set of experiments, we use *Baseline* to refer to the solution where both components are disabled, *i.e.* the programmable switch only forwards packets; *Baseline+agg* to refer to the solution where the programmable switch also performs answer aggregation; *NetSHa* uses both components.

Figure 16 shows the number of packets the agent server receives when adopting the three solutions. We find that the answer aggregation can reduce the number of answer packets to 40% of the *Baseline*, while answer reduction can further half the number of packets. Figure 17 depicts the impact of these two components on total processing time. We see they both reduce the processing time. Note that

9. Due to space limitations, we omit the results with other datasets as the results are similar.

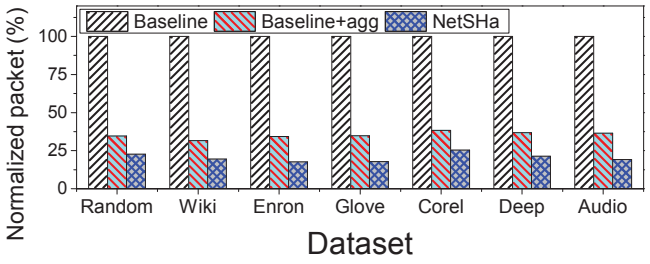


Fig. 16. The number of packets (normalized with the *Baseline* solution) that the agent received.

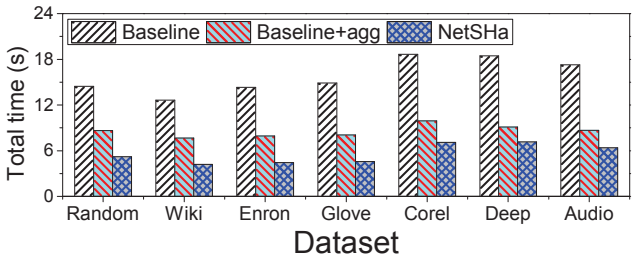


Fig. 17. The impact of the two components (answer reduction and aggregation) on total processing time.

the time reduction by the answer aggregation is due to the reduced packet volume, while the time savings introduced by the answer reduction comes from both reduced packet volume *and* the dropping of poor answers.

5.5 Impact of System Parameters

The performance of NetSHa is naturally impacted by the choice of system parameters. Hence, we next discuss the impact of the parameters: (*i*) the number of (logical) registers, (*ii*) the capacity of each register (register slots), and (*iii*) the number of GA slots in each register on the performance. The number of registers decides how many queries the switch can simultaneously process, while the register capacity affects the amount of packets it can process and store. The number of GA slots decides the efficiency of the reduction process and the number of candidate answers that can be aggregated. Note that a negative correlation exists between these three parameters, as the memory size of a programmable switch is constant.

The memory size of each switch in our experiments is restricted to 200 slots (register slots $\times$ #register). We first vary the number of registers and the capacity of each register to examine their relationship. Note that we fix GA slots in each register to 8 (*i.e.* $p = 8$). Figure 18 plots the results. We see that the larger the capacity of each register is, the smaller the total processing time is. Nevertheless, the marginal benefits reduce as we increase the number of slots of each register beyond 12.

Next, we examine the relationship between the number of registers and the number of GA slots in each register in Figure 19. In this set of experiments, we fix the capacity of each register to 12 slots. The total processing time becomes smaller as we increase the number of GA slots from a small value. However, when we increase the number of GA slots beyond 10, the total processing time increases. This is

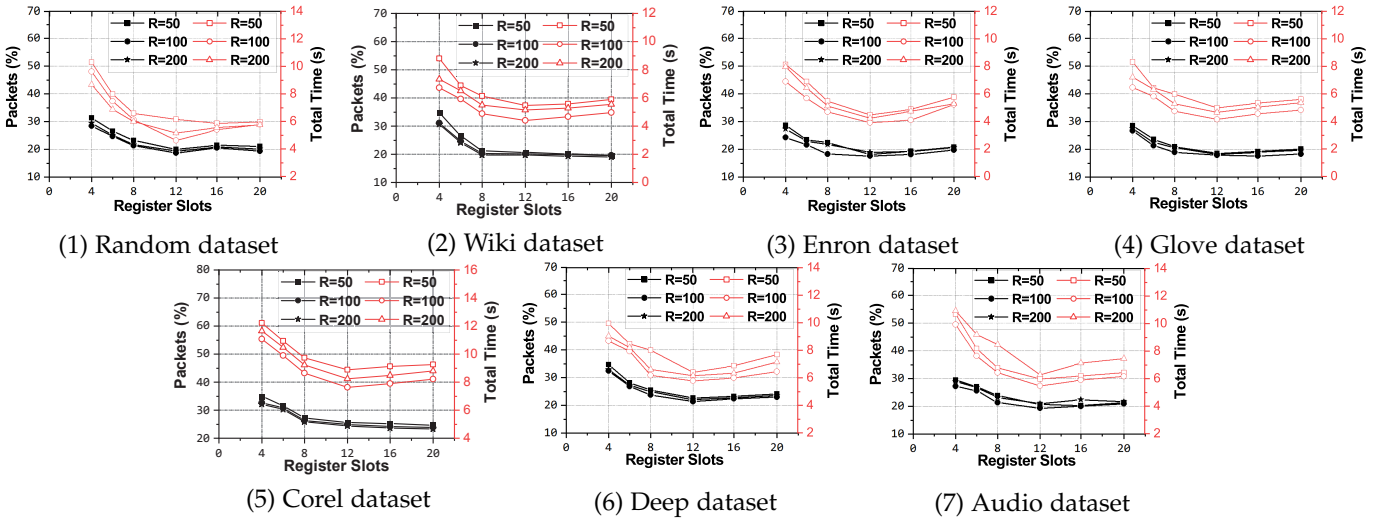


Fig. 18. The tradeoff between the number of registers and the capacity of each register (register slots). R refers to the amount of registers in the switch. Note that each subfigure has two y axises. The left y-axis represents the percentage of the packets that the agent receives (black line), while the right y-axis refers to the total processing time (red line).

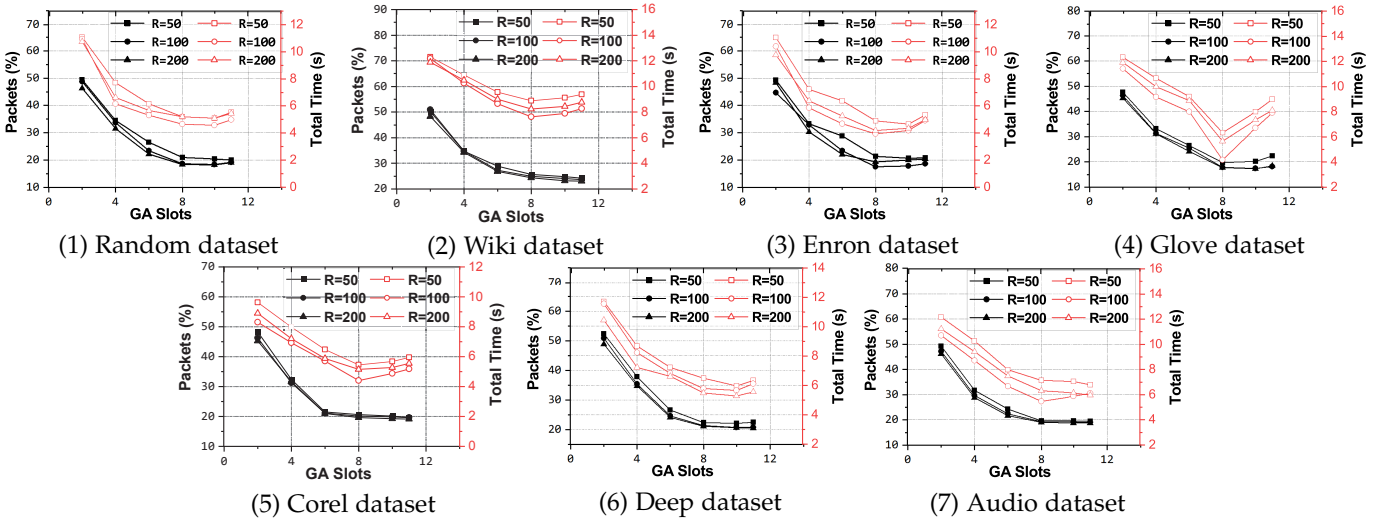


Fig. 19. The tradeoff between the number of registers and the number of GA slot in each register. R refers to the amount of registers. Similar to Figure 18, each subfigure also has two y-axises.

because too many GA slots leave a very limited number of slots available for PA, and thus will decrease the efficiency of the answer reduction.

Fig. 20. The total processing time for 10M queries: comparison of our register replacement strategy with the bypassing-forwarding mechanism.

Next, we compare the register replacement strategy adopted by NetSHa with the bypassing-forwarding (BF) mechanism. Figure 20 shows the result. Note that we increase the query rate ($QueryRate = 4.0$) to simulate a heavy workload. Indeed, our replacement strategy outperforms the BF mechanism because it can reduce and aggregate answers more, thereby mitigating the performance bottleneck.

Finally, we evaluate the extra overhead introduced by NetSHa due to the sort and insertion operations in extreme scenarios (*i.e.* low or no answer reduction). To simulate such scenarios, we implement two variants of NetSHa: the first variant sorts candidate answers and insert the input answer in the correct order but *without* any reduction (*i.e.* all answers are sent to the agent.); the second variant further disables sort and insertion operation. Then we record the total processing time of 10M queries by these two variants. The difference between their processing time thus corresponds to the overhead introduced by the sort and insertion operations. Table 3 shows the results for different datasets. We can observe that the extra overhead is indeed marginal, which is below 390ms. The reason is that the sort and insertion operations are only needed before we get the

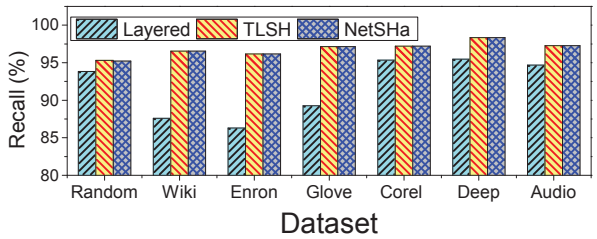
threshold for filtering out subsequent poor results. We argue that such overheads are reasonable, considering the benefits brought by NetSHa.

TABLE 3
The extra overhead introduced by NetSHa in 10M queries (ms).

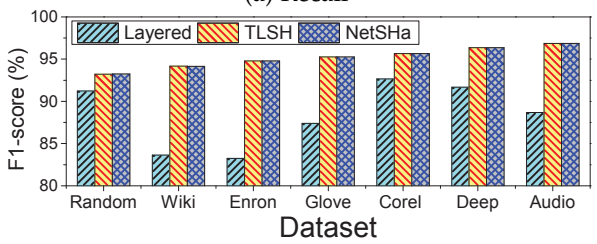
Datasets	Random	Wiki	Enron	Glove	Corel	Deep	Audio
Δ (ms)	359.1	341.6	313.3	346.3	378.9	384.3	361.2

5.6 Search Quality

Next, we need to verify that the search quality is not degraded. The experimental setting is the same as those of the previous experiments.



(a) Recall



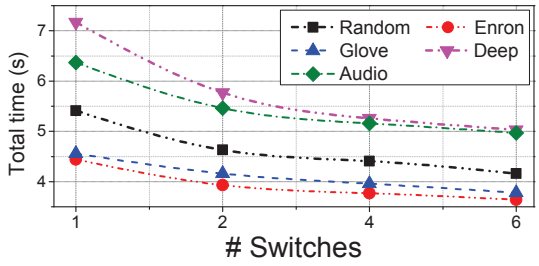
(b) F1-score

Fig. 21. The search quality of the three solutions

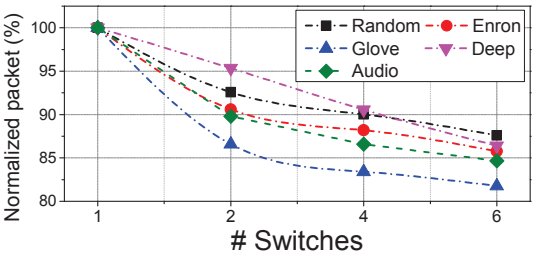
Figure 21 compares the search quality of the three solutions. The search quality is measured by the recall rates (see Figure 21(a)) and F1-score (see Figure 21(b)). For both the recall rate and the F1-score, NetSHa achieves similar search quality (exceeding 90%) to the baseline distributed search implementations. This is because NetSHa does not break any functional properties of the LSH family. Instead, it only accelerates them. Thus, the search quality of NetSHa is identical to that of the LSH functions it adopts. In this set experiments, NetSHa adopts TLSH functions to search for candidate answers on the server side, so its search quality is the same as TLSH.

5.7 Scalability of NetSHa

As described before, NetSHa also works on multi-stage switches so that it can be easily deployed in data center networks. To verify this, we use one Barefoot Tofino switch to simulate multiple programmable switches, each of which uses the default system configuration (see Table 1). Specifically, we divide the on-chip memory of the switch into multiple groups, each of which corresponds to one logical switch and performs the answer reduction and aggregation tasks independently. Consequently, when one logical switch outputs NetSHa packets, these packets might be recirculated[10] back to the physical switch pipeline and enter the next logical switch for further answer reduction and aggregation. That said, six switches use 6× on-chip memory of that in one switch.



(a) The total processing time for 10M queries.



(b) The number of output NetSHa packets.

Fig. 22. The performance of NetSHa when varying the number of switches. In Figure 22 (b), we normalize the output NetSHa packets of each switch with the that of the first switch.

In this experiment, we vary the number of logical switches from one to six, and evaluate the performance of NetSHa using the different datasets. Note that we omit the *Wiki* and *Corel* dataset due to their relatively small sizes. Figure 22(a) shows that the processing time is significantly reduced when increasing the number of switches[11]. We observe that it can achieve up to 30% improvements. This is because multiple switches cooperatively drop more poor answers, thereby reducing the overall system workload. Again, we can see from Figure 22(b) the number of packets decreases with more switches. In summarize, multi-stage switches do improve the system efficiency cooperatively. Nevertheless, using multiple switches does not offer linear speed-up. The reason involves two aspects. First, the performance improvement by NetSHa stems from answer aggregation and answer reduction. In the simulated pipelined multi-switch environment, the first switch aggregates candidate answers as far as possible so that the subsequent switches cannot perform much further aggregation. Second, the first switch also filers out many poor answers. Consequently, the subsequent switches can only drop a few poor answers.

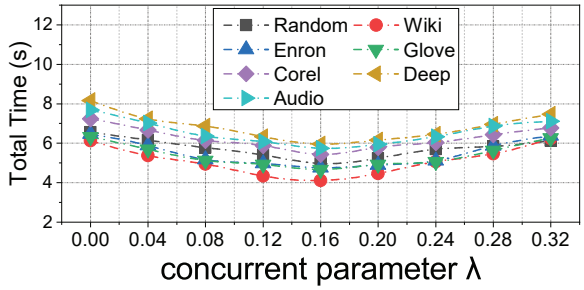
5.8 Concurrent Parameter of NetSHa

Recall, in Section 3.6 we designed a "best-effort" replacement mechanism to increase NetSHa's concurrency. This

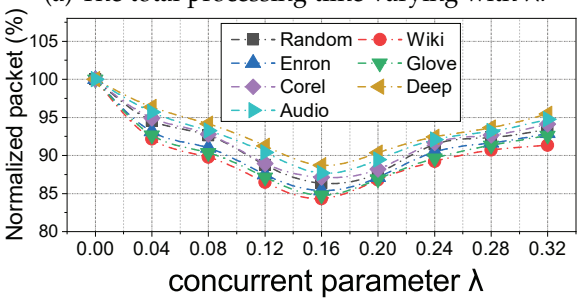
10. This is achieved by the built-in recirculation operation of the Tofino switch.

11. Note that the network latency between the simulated logical switches can be ignored. This is because we use a chip instruction (i.e., recirculation) provided by Tofino switches to simulate the multi-switch environment.

relies on a concurrency parameter, λ. We next evaluate the impact of this parameter.



(a) The total processing time varying with λ.



(b) The number of packets that the agent received.

Fig. 23. The performance of NetSHa varying the value of λ. In Figure 23 (b), we normalize the number of packets with that in λ = 0.

TABLE 4
The ptimal λ under different query rates.

QueryRate(#query=1M)	1.0	2.0	3.0	4.0
optimal λ	0.16	0.20	0.20	0.24
QueryRate(#query=10M)	1.0	2.0	3.0	4.0
optimal λ	0.16	0.20	0.20	0.24

We first take the default parameter configuration and vary the value of λ. Figure 23(a) shows the total time for processing 1M concurrent queries[12], while Figure 23(b) plots the number of packets that the agent received. We see that a larger value of λ does result in better performance. This is intuitive, as the increased concurrency allows for higher throughput. Nevertheless, we also find that the benefit is reduced as we increase the value of λ beyond about 0.16, due to increased cache misses. We also examined the optimal λ under different query rates, which were varied as in Figure 14. The results are summarized in Table 4. We see that the optimal λ is positively correlated with the query rate within limits. Note that the optimal λ value cannot keep increasing because a large λ value would lead to frequent register replacements (cache misses) which will in turn degrade the performance. Based on our results, we recommend initializing λ with an empirical value (*e.g.* 0.2), and then adjust it based on the query rate.

6 Related Work

Nearest neighbor search. Nearest neighbor (NN) search has attracted much attention over the last decade, and serves as the foundation of many applications. It is particularly important in data mining applications for web platforms, *e.g.* for event detection, sequence matching and data retrieval. Previous works have focused on developing novel data structures, such as cover tree [35], k-tree [36] and k-means tree [37] to partition the input datasets so as to prune the search space. However, these methods do not work well in high-dimensional data [38]. Hence, approximate nearest neighbors (ANN) search has been used extensively [5], [6], where exact answers are not essential.

Locality sensitive hashing. Locality Sensitive Hashing (LSH) is widely recognized as one of the most effective methods for ANN [39]. A variety of LSH approaches have been designed for different scenarios, such as MinHash [11] for Jaccard distance, E2LSH [12] for Euclidean distance and SimHash [13] for Angular distance. To guarantee high recall, these LSH functions need to generate a number of hash tables, which consumes a large amount of memory [8]. Thus, some variants of LSH have been proposed to address the memory expansion problem [9], [15], [40], [41]. The key idea is that if similar items are not in the same bucket, they are likely allocated to other "nearby" buckets in the hash table, such that they only need to interrogate those "nearby" buckets. Some other LSH variants (*e.g.* TLSH [14]) have also been proposed to accelerate similar item matching.

Distributed LSH. For large-scale web applications, LSH is required to handle very large datasets. This means that LSH running on single host cannot work well, *e.g.* in terms of precision, recall and delay [42]. Consequently, there have been recent attempts to enable efficient distributed search [19], [43]. There are a variety of well-known implementations. Some are built upon general-purpose distributed frameworks (*e.g.* MapReduce) [44], [45], [46]; however, these are not optimized for LSH-based search. This makes it necessary to translate LSH functions into their programming models, increasing system complexity. To address this gap, some researchers have designed distributed frameworks specifically for LSH-based search, *e.g.* LoSHa [18] and Layer-hash [17]. Since the network constitutes a key bottleneck in distributed search [20], some approaches have also been proposed to balance load between different servers [21] or reduce the number of network calls [17] in order to reduce the network burden. Despite this, these optimizations focus on the end systems (server side), and it is possible that the transmitted data volume (even after using these optimizations) still exceeds the forwarding capacity of the network, given the ever-growing data scale and volume of concurrent queries. As such, in this paper, we have turned our efforts to the network itself and have leveraged programmable switches for computational offloading from the agent server. That said, NetSHa is complementary to the above mentioned optimisation studies.

Network incast. In data centers, network incast widely exists in many-to-one communication patterns [47], [48]. A large amount of prior works have been proposed to address this problem by reducing the queuing delay and packet losses on switches. To this end, researchers have designed new congestion control algorithms [49], optimized application-level data transfer patterns [50] and made switch-level modifications [51]. In contrast, NetSHa considers this problem from another perspective. We reduce the

12. We vary λ from 0 to 0.32. The interval is 0.04.

transferred traffic volume by offloading some tasks from the host side to the network.

In-network computation. With the rise of SmartNICs and programmable switch-ASICs (*e.g.* Tofino [23]), in-network computation can be used to offload certain application-specific primitives onto network devices [52]. For example, NetCache [53] uses the network to quickly reply with cached values; Beamer [54] implements a load balancer on network devices; AcclTCP [55] offloads some TCP tasks to SmartNICs; NetEC [56] offloads erasure coding to programmable switches; DAIET [57]; SwithML [28] and ATP [27] aggregate distributed deep learning model gradient updates on programmable switches. In contrast to these prior works, we have exploited in-network computation capacity to accelerate distributed search for the first time.

7 Conclusion

In this paper, we have designed and implemented NetSHa, a system to accelerate LSH-based distributed search with in-network computation. As a key design requirement, NetSHa does not change the architecture of the traditional distributed search system. Rather, it utilizes the computational capacity of programmable switches to reduce the communication cost while maintaining search quality. To this end, NetSHa uses a customized packet format, and performs answer reduction and answer aggregation on programmable switches, in order to reduce the communication overhead. NetSHa has been implemented for the Barefoot Tofino switch. Extensive experiments, with various types of datasets, have shown that NetSHa accelerates search performance, while maintaining search quality. A prominent feature of NetSHa is that it is complementary to prior optimizations in the distributed search research community and thus can work with them together.

References

[1] P. Zhang, H. Pan, Z. Li, P. He, Z. Zhang, G. Tyson, and G. Xie, "Accelerating lsh-based distributed search with in-network computation," in *International Conference on Computer Communications*, 2021.

[2] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 271–280.

[3] D. Ravichandran, P. Pantel, and E. Hovy, "Randomized algorithms and nlp: Using locality sensitive hash functions for high speed noun clustering," in *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, 2005, pp. 622–629.

[4] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, p. 623, 2015.

[5] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, "Query-aware locality-sensitive hashing for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 1–12, 2015.

[6] J. Li, X. Yan, J. Zhang, A. Xu, J. Cheng, J. Liu, K. K. Ng, and T.-c. Cheng, "A general and efficient querying method for learning to hash," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1333–1347.

[7] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[8] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and efficiency in high dimensional nearest neighbor search," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 563–576.

[9] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 541–552.

[10] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi, "Dsh: data sensitive hashing for high-dimensional k-nnsearch," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1127–1138.

[11] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[12] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.

[13] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002, pp. 380–388.

[14] R. Shinde, A. Goel, P. Gupta, and D. Dutta, "Similarity search and locality sensitive hashing using ternary content addressable memories," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 375–386.

[15] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. Society for Industrial and Applied Mathematics, 2006, pp. 1186–1195.

[16] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 1–12, 2014.

[17] B. Bahmani, A. Goel, and R. Shinde, "Efficient distributed locality sensitive hashing," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 2174–2178.

[18] J. Li, J. Cheng, F. Yang, Y. Huang, Y. Zhao, X. Yan, and R. Zhao, "Losha: A general framework for scalable locality sensitive hashing," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2017, pp. 635–644.

[19] G. Andrade, A. Fernandes, J. M. Gomes, R. Ferreira, and G. Teodoro, "Large-scale parallel similarity search with product quantization for online multimedia services," *Journal of Parallel and Distributed Computing*, vol. 125, pp. 81–92, 2019.

[20] H. Li, S. Nutanong, H. Xu, F. Ha *et al.*, "C2net: A network-efficient approach to collision counting lsh similarity join," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 3, pp. 423–436, 2018.

[21] Y. Kim, J. Callan, J. S. Culpepper, and A. Moffat, "Load-balancing in distributed selective search," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 2016, pp. 905–908.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[23] "Barefoot networks," https://barefootnetworks.com/products/brief-tofino/.

[24] Y. Li, I. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *ISCA*, 2019.

[25] "Bubble sort," https://en.wikipedia.org/wiki/Bubble_sort, 2020.

[26] T. Qu, R. Joshi, M. C. Chan, B. Leong, D. Guo, and Z. Liu, "Sqr: In-network packet loss recovery from link failures for highly reliable datacenter networks," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 2019, pp. 1–12.

[27] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. M. Swift, "Atp: In-network aggregation for multi-tenant learning," in *NSDI*, 2021.

[28] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with in-network aggregation," in *NSDI*, 2021.

[29] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[30] "Wiki dataset," https://dumps.wikimedia.org/enwiki/20190701/, 2019.

[31] "Enron dataset," https://p4.org/p4-spec/docs/PSA-v1.1.0.html/, 2021.

[32] "Glove dataset," https://nlp.stanford.edu/projects/glove/, 2021.

[33] "Deep dataset," https://yadi.sk/d/I_yaFVqchJmoc, 2021.

[34] "Audio dataset," http://www.cs.princeton.edu/cass/audio.tar.gz, 2021.

[35] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 97–104.

[36] K. He and J. Sun, "Computing nearest-neighbor fields via propagation-assisted kd-trees," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012, pp. 111–118.

[37] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, no. 331-340, p. 2, 2009.

[38] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 322–373, 2001.

[39] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[40] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu, "Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2023–2037.

[41] Y. Liu, J. Cui, Z. Huang, H. Li, and H. T. Shen, "Sk-lsh: an efficient index structure for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 745–756, 2014.

[42] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *arXiv preprint arXiv:1408.2927*, 2014.

[43] O. Durmaz and H. S. Bilge, "Fast image similarity search by distributed locality sensitive hashing," *Pattern Recognition Letters*, 2019.

[44] A. Stupar, S. Michel, and R. Schenkel, "Rankreduce–processing k-nearest neighbor queries on top of mapreduce," *Large-Scale Distributed Systems for Information Retrieval*, vol. 15, 2010.

[45] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *Proceedings of the 15th international conference on extending database technology*. ACM, 2012, pp. 38–49.

[46] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[47] Y. Chen, R. Griffit, D. Zats, and R. H. Katz, "Understanding tcp incast and its implications for big data workloads," *University of California at Berkeley, Tech. Rep*, 2012.

[48] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[49] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in data-center networks," *IEEE/ACM transactions on networking*, vol. 21, no. 2, pp. 345–358, 2012.

[50] Y. Yang, H. Abe, K.-i. Baba, and S. Shimojo, "A scalable approach to avoid incast problem from application layer," in *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. IEEE, 2013, pp. 713–718.

[51] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 63–74.

[52] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

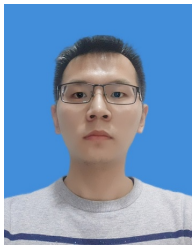
[53] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[54] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 125–139.

[55] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "Acceltcp: Accelerating network applications with stateful {TCP} offloading," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 77–92.

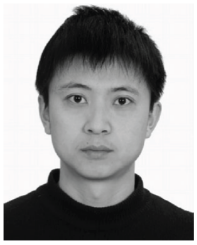
[56] Y. Qiao, X. Kong, M. Zhang, Y. Zhou, M. Xu, and J. Bi, "Towards in-network acceleration of erasure coding," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 41–47.

[57] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 150–156.



Penghao Zhang is currently a Ph.D candidate at the Institute of Computing Technology, Chinese Academy Sciences. He received his B.S. degree in Computer Science from Hunan University, Changsha, China, in 2017. His research interests include programing switch, packet classification and SDN policy anomaly detection.

Heng Pan received the PhD degree in computer science from University of Chinese Academy Sciences in 2018. He is an assistant professor at the Institute of Computing Technology, Chinese Academy Sciences. His research interests include SDN/NFV, distributed system and in-network computation.



Zhenyu Li received the BS degree from Nankai University in 2003 and the PhD degree in Graduate School of Chinese Academy of Sciences in 2009. He is a professor at the Institute of Computing Technology, Chinese Academy Sciences. His research interests include Internet measurement and Networked Systems.

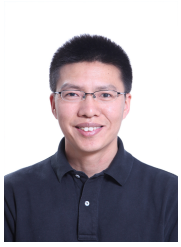
Penglai Cui is currently a Ph.D candidate at the Institute of Computing Technology. He received his B.S. degree in Computer Science and Technology from University of Chinese Academy of Sciences, Beijing, China, in 2018. His research interests include in-switch computation, distributed machine learning system, and programmable switch.

Ru Jia is currently a PhD candidate at the institute of Computing Technology, Chinese Academy of Sciences. She received her B.S. degree in Computer Science from Dalian University of Technology, Dalian, China, in 2015. Her research interests include software packet processing, Network Function Virtualization.

Peng He received the PhD degree from University of Chinese Academy Sciences in 2014. He is currently a Software Engineer with ByteDance Inc. His research interests include high-performance packet processing algorithms and network function virtualization.

Zhibin Zhang received the PhD degree in Graduate School of Chinese Academy of Sciences in 2007. He is a associate professor at the Institute of Computing Technology, Chinese Academy Sciences. His research interests include Big data, machine learning algorithms and systems.

Gareth Tyson received the PhD degree from Lancaster University in 2010. He is a senior lecturer at Queen Mary University of London. His research interests include Internet measurements, content distribution, and the future Internet.



Gaogang Xie received the PhD degree in computer science from Hunan University in 2002. He is a professor at the Computer Network Information Center, Chinese Academy of Sciences. His research interests include Internet architecture, SDN/NFV, and Internet measurement.