Towards Accelerating Data Intensive Application's Shuffle Process Using SmartNICs

JIAXIN LIN, The University of Texas at Austin, USA
TAO JI, The University of Texas at Austin, USA
XIANGPENG HAO, University of Wisconsin Madison, USA
HOKEUN CHA, University of Wisconsin Madison, USA
YANFANG LE, Intel, USA
XIANGYAO YU, University of Wisconsin Madison, USA
ADITYA AKELLA, The University of Texas at Austin, USA

The wide adoption of the emerging SmartNIC technology creates new opportunities to offload application-level computation into the networking layer, which frees the burden of host CPUs, leading to performance improvement. Shuffle, the all-to-all data exchange process, is a critical building block for network communication in distributed data-intensive applications and can potentially benefit from SmartNICs.

In this paper, we develop ***SmartShuffle***, which accelerates the data-intensive application's shuffle process by offloading various computation tasks into the SmartNIC devices. SmartShuffle supports offloading both low-level network functions, including data partitioning and network transport, and high-level computation tasks, including filtering, aggregation, and sorting. SmartShuffle adopts a *coordinated offload architecture* to make sender-side and receiver-side SmartNICs jointly contribute to the benefits of shuffle computation offload. SmartShuffle carefully manages the tight and time-varying computation and memory constraints on the device. We propose a *liquid offloading* approach, which dynamically migrates computations between the host CPU and the SmartNIC at runtime such that resources in both devices are fully utilized.

We prototype SmartShuffle on the Stingray SoC SmartNICs and plug it into Spark. Our evaluation shows that SmartShuffle improves host CPU efficiency and I/O efficiency with lower job completion time. SmartShuffle outperforms Spark, and Spark RDMA by up to 40% on TPC-H.

CCS Concepts: • **Networks** → **In-network processing**.

Additional Key Words and Phrases: SmartNIC; data analytics; hardware offloading

ACM Reference Format:
Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. 2023. Towards Accelerating Data Intensive Application's Shuffle Process Using SmartNICs. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 36 (June 2023), 23 pages. https://doi.org/10.1145/3589980

1 INTRODUCTION

Modern data-intensive applications run on distributed machines and induce significant network communication. Various hardware technologies (e.g., programmable switches) have been developed to add intelligence to the network to reduce communication overhead and accelerate performance. SmartNIC, in particular, is an emerging technology that extends a conventional network interface

Authors' addresses: Jiaxin Lin, The University of Texas at Austin, USA; Tao Ji, The University of Texas at Austin, USA; Xiangpeng Hao, University of Wisconsin Madison, USA; Hokeun Cha, University of Wisconsin Madison, USA; Yanfang Le, Intel, USA; Xiangyao Yu, University of Wisconsin Madison, USA; Aditya Akella, The University of Texas at Austin, USA.

2476-1249/2023/6-ART36
https://doi.org/10.1145/3589980

card (NIC) with extra computation and memory. SmartNIC adoption has grown in recent times, with widespread deployments in cloud data centers (e.g., AWS Nitro [2] and Azure [20]).

Existing SmartNIC acceleration solutions focus on offloading specific packet processing functions, such as traffic scheduling, security [49, 52], and network virtualization [2, 2, 20, 22]. With a focus on such simplistic acceleration today, SmartNIC deployments leave significant NIC computational resources unused. In this paper, we address how best to leverage these resources toward systematically accelerating data-intensive workloads. We discuss key arguments for when/why SmartNIC-acceleration of data-intensive applications matters and develop building blocks for enabling it using the example of SmartNIC-accelerated *shuffle* operators.

The shuffle (a.k.a. *Exchange*) operator is a common building block in data analytics applications (e.g., MapReduce [19], Spark [56], and analytical databases [13, 31]) and is used to perform intermediate data exchange. Shuffle has been identified as a key performance bottleneck in these applications [12, 13, 17, 42, 45, 48, 57], and has been optimized through various techniques in both software [9, 17, 48, 57] and custom hardware [1, 31, 50, 51].

We argue that an operator like shuffle is a good candidate for acceleration because the SmartNIC is computationally capable of conducting shuffle-related tasks and has a system-wide view of the network behavior that the host applications may lack. Accelerating the shuffle could lead to significant savings in host CPU. Further, because shuffle forms the intermediate stages of data-intensive applications, the CPUs saved from shuffle acceleration could be leveraged toward completing such applications faster by scheduling downstream application stages for execution earlier than otherwise possible; the saved CPUs can also be used to launch additional data-intensive workloads earlier than otherwise.

A key challenge in offloading shuffle to SmartNICs is the limited hardware resources of the NICs. Compared to a host CPU, SmartNIC processors are typically less powerful with a smaller memory capacity; as such the shuffle and the associated computation tasks may not entirely fit in the SmartNIC. Furthermore, with network processing functions offloaded to the NIC, and with the time-varying demands of both network processing and data-intensive workloads, shuffle offload needs to be adaptive so as to be effective.

Our SmartNIC-optimized shuffle framework, **SmartShuffle**, opportunistically and fully utilizes on-NIC hardware resources by carefully coordinating the computation across the host CPU and SmartNICs so that both computational substrates are fully utilized. SmartShuffle is exposed to a big-data analytics system via a unified API and can dynamically determine whether the associated computation should be executed on the SmartNICs, the host, or a hybrid of the two.

SmartShuffle supports not only the basic network communication functions in data-intensive applications, but also the associated tightly coupled common operators, including data partitioning, filtering, aggregation, and sorting functions. By carefully offloading these functions to the SmartNIC, SmartShuffle reduces the burden of host CPU computation and minimizes network traffic volume.

In a typical shuffle, data is exchanged from "map" tasks to "reduce" tasks. To maximize the SmartNIC's resource utilization, SmartShuffle leverages the SmartNIC computation from both the mapper and reducer nodes. SmartShuffle adopts a *coordinated offload architecture* where both the map-side and the reduce-side SmartNICs of a shuffle jointly contribute to the shuffle offload and relevant computation. The map-side SmartNIC offload merges many small I/O requests, reduces network traffic, and frees up host CPUs by applying application functions such as filtering, aggregation, and sorting where available in the input data analytics query (e.g., a SQL query over large datasets). The reduce-side SmartNIC offload further reduces computation load or data traffic to the host CPU by applying similar functions over incoming data from different nodes.

To manage limited SmartNIC computation and memory capability, SmartShuffle uses a new technique called *liquid offloading* to avoid the SmartNIC becoming a new performance bottleneck

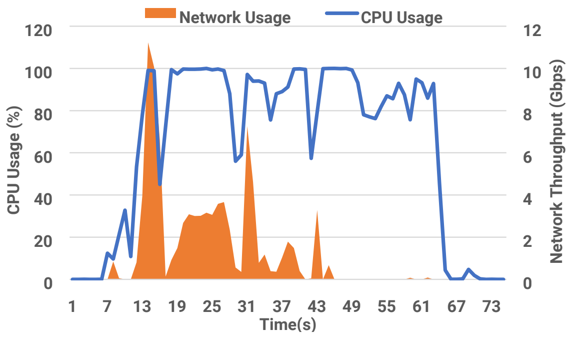


Fig. 1. **Running 320GB TeraSort using Spark. Measure the CPU and network usage of an executor.**

when too much computation is offloaded to it. Liquid offloading has two components. First, to ensure that offloaded tasks fit the memory capacity on both map-side and reduce-side SmartNICs, we perform partial offloading for certain stateful operators (e.g., aggregation and sorting) and stream computation results to the next hop to release the on-NIC memory resources. A key research challenge is to identify appropriate streaming strategies for different types of operators to achieve the best performance. Second, to maximize the amount of work offloaded to the SmartNIC while avoiding the typically slow SmartNIC cores becoming a bottleneck, we detect SmartNIC load and dynamically move some of the shuffle computation from the SmartNIC to the host cores. We design a novel rate-based dynamic migration policy that adapts to the runtime computation behavior of batch analysis jobs.

We have developed a prototype of SmartShuffle in Spark, a popular big-data analytics framework. We evaluate the performance of SmartShuffle on synthetic workloads as well as representative TPC-H queries. Our evaluation shows that SmartShuffle can decrease Spark job completion time by 10%–30% and reduce CPU usage by 30%–70% compared to the original Spark. The evaluation on the TPC-H [8] workload shows that SmartShuffle outperforms Spark, Spark RDMA [1] (i.e., an RDMA-based shuffle optimization in Spark), by up to 40%.

Even though we focus on shuffle, we argue that the ideas underlying SmartShuffle are general enough to help accelerate other types of data-intensive applications [5, 15, 46]. For example, a distributed ML training framework [5] can utilize SmartShuffle's framework to accelerate the collective communication and gradient averaging process on SmartNIC cores. Likewise, a distributed database's query planner [46] can co-design closely with SmartShuffle to offload network communications as well as a portion of query execution down to the NIC. Our work opens up exciting possibilities for such acceleration.

2 BACKGROUND AND MOTIVATION

Shuffle is a key building block in distributed data analytics applications including big data platforms like MapReduce [19] and Spark [56] and distributed online analytical processing (OLAP) databases. In this section, we analyze the performance overhead of shuffle using Spark as the target system (Section 2.1); the conclusions can be generalized to other distributed systems as well. We then discuss the opportunities (Section 2.2) and challenges (Section 2.3) of using SmartNIC offload to improve shuffle performance.

2.1 Shuffle CPU Cost Analysis in Spark

Shuffle is the inter-node communication mechanism in Spark. In Spark's shuffle process, mappers first create data partitions and re-arrange the data tuples within each partition using appropriate

	Operation Type	TeraSort	TeraCount	PageRank
Shuffle Map Stage	RDD Computation	11.4%	50.4%	58.1%
	Shuffle Pre-processing	27.8%	39.4%	20.1%
	Byte Stream Generation	11.3%	1.0%	15.0%
	Misc	2.1%	5.3%	3.3%
	Total	51.6%	96.2%	97.1%
Output Stage	Total	48.3%	3.8%	2.8%

Table 1. **The percentage of time spent on RDD computations, shuffle pre-processing, and byte stream generation in different computation stages. The shuffle pre-processing phase includes both partitioning and the shuffle data-reorganization process.**

data-reorganization operators, i.e. aggregation, and sorting, to facilitate parallel computation in the reduce stage [13, 55, 58]. The mappers then perform *byte stream operations*, i.e., data serialization and compression, dump the intermediate data to disk, and notify the Spark driver of the data location. The reducers fetch the data over the network and execute user-defined reduce functions.

While other systems may not explicitly include data reorganization operators in the shuffle process (as the data-reorganization functions can be implemented as the aggregation/sort operator in the query plan [46, 58]), we follow the Spark model of including them as part of the shuffle. However, regardless of the exact definition of shuffle, the analyses and conclusions in this section will still apply.

Figure 1 shows the measured network and CPU usage when running a 320 GB TeraSort workload in a 4-node Spark cluster; the detailed experimental setup will be explained in Section 6. The result shows that the bottleneck in the shuffle process is CPU computation rather than network operations, which is consistent with the conclusions of previous studies [1, 40, 51, 54]. In this shuffle-intensive workload, CPU utilization is close to 100%, while network utilization is generally lower than 50%, indicating that the CPU bottleneck results in underutilization of the network bandwidth.

We identify two key reasons for high CPU computation overhead in these shuffle-intensive workloads:

Shuffle Pre-processing Cost. We have broken down each function of the shuffle process at the mapper side for three Spark workloads [24]: *TeraSort*, *TeraCount*, and *PageRank*. Based on the blocked time analysis [40], Table 1 shows the relative time spent during the shuffle process. The pre-processing overhead in shuffle is non-trivial (> 20%), especially for TeraSort (28%) and TeraCount (40%), since these two workloads incur pre-sorting and pre-aggregation overhead when preparing the shuffle data. In PageRank, the only pre-processing is hash partitioning, which still incurs more than 20% overhead. The byte stream generation overhead is correspondingly high in TeraSort and PageRank (> 10%). As for TeraCount, since the mapper output has already been pre-aggregated, it generates less intermediate shuffle data and has minimal serialization/compression overhead.

Overhead of Small I/O. During shuffle, each reducer issues a large number of remote read requests to collect partitions from the designated mappers. This not only incurs small network I/Os for remote fetching but may also result in a massive number of small random disk read I/Os on the mapper nodes if the shuffle data is materialized on the disk [40, 48, 57].

The number of I/Os grows quadratically with the number of tasks in a job, and the I/O size shrinks quadratically. In hyper-scale clusters, the daily I/O request count is very high (tens of billions), with the average block size as small as 10s of KBs [48, 57]. These small I/Os stress the host CPU [9, 29, 31]. Furthermore, to achieve high I/O throughput, recent works propose to use dedicated servers or tasks to merge small disk/network I/Os [48, 57], which incurs even higher host CPU overhead.

2.2 Benefits of SmartNIC Offload

SmartNICs extend traditional network interface cards (NICs) with integrated processing units (e.g., general-purpose cores, FPGA, AISC), onboard memory, hardware accelerators, and an on-chip network traffic manager. We mainly focus on multicore SoC SmartNICs because of their general-purpose programmability [14, 16, 36], which enables flexible offloading of user-defined functions that involve non-deterministic data structures or complex algorithms. We identify the following key benefits of offloading shuffle-related operations onto SoC SmartNICs:

SmartNIC location is ideal. Since shuffle data must pass through the NICs, offloading shuffle-related computation to the SmartNICs does not incur extra scheduling or network transfer overhead.

SmartNIC as a new pipeline stage. As shown in Figure 4b, the SmartNIC provides a natural asynchronous pipeline stage between mappers and reducers. This means that a job's shuffle-related computation that happens on a SmartNIC can be made to overlap with other jobs' map and reduce stages running on the host CPU or with the same job's downstream map stages. By offloading CPU-costly shuffle operations, we can not only improve the execution time of a job (by allowing its tasks to be scheduled earlier than otherwise) but also pack more jobs' computation onto the same host to improve efficiency.

Host-level I/O merging and data pruning. The SmartNIC is a perfect place to perform host-level network I/O merging, data aggregation, and data filtering, because the NIC has a complete view of the output of all map tasks from the local host, as well as all the shuffle data received from the network. For example, a SmartNIC can merge small partitions from multiple map tasks to the same destination into larger chunks to avoid small network I/O operations, optimizing communication efficiency and CPU usage.

The data exchange between the SmartNIC and the local host occurs through PCIe, whose bandwidth exceeds that of the NIC in many cluster settings. Figure 2 shows the read and write performance of both DMA and RDMA for a Stingray 25Gb NIC [14], in which the NIC are using the PCIe 3.0 x8 interface. The result shows that the network throughput of the NIC (NIC $\rightarrow$ r-host) is 25Gbps while the network DMA bandwidth between NIC and local host (NIC $\rightarrow$ l-host) can be up to ~58Gbps. This brings opportunities for on-NIC pruning, e.g., shuffle data aggregation and filtering on the NIC to reduce outbound network traffic. In the near future, the industry will universally adopts PCIe 4.0 (maximum throughput 256 Gbps) and eventually PCIe 5.0 (512 Gbps), which further increases the opportunity for on-NIC pruning.

Why not use host cores: SmartNICs are closer to the network than host cores. By aggregating and pruning data received over the network before sending it to the host, the NIC cores can help reduce data movement across PCIe.

Furthermore, SmartNICs are already widely deployed in data centers [2, 20], with existing clusters leaving significant NIC computational resources unused. Thus, there is a growing body of research on SmartNICs with a focus on utilizing these free on-NIC compute units to improve specific types of applications [26, 33, 34, 47]. Our work is aligned with this research and aims to leverage these unused resources to accelerate data-intensive workloads.

2.3 Challenges of SmartNIC Offloading

We identify the following challenges in offloading shuffle-related computation to SmartNICs:

Limited memory. Shuffle stages typically transfer data far larger than the on-NIC DRAM (4GB--16GB). Offloading stateless operators, such as partitioning, can be easy as they can operate with a small local working set. However, offloading stateful operators such as aggregation and sorting requires a large view of the dataset. Such operators usually cannot fit into the on-NIC memory, making it difficult to fully offload them.

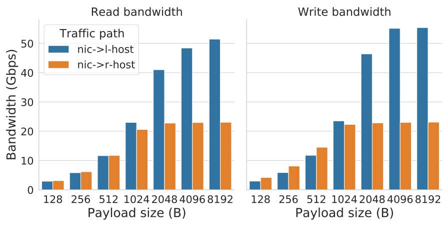
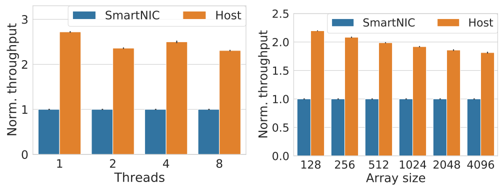
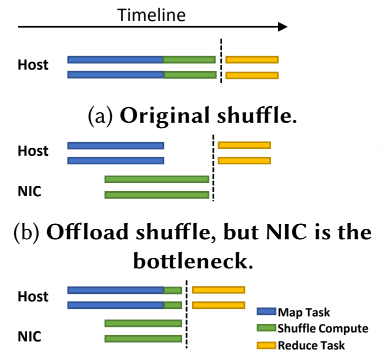


Fig. 2. **SmartNIC's (R)DMA bandwidth. NIC-lhost measures the DMA bandwidth between the NIC and the local host. NIC-rhost measures the RDMA bandwidth between the NIC and remote host.**



(a) **Wordcount** (b) **Iterative merge sort**

Fig. 3. **Comparison of computation performance between SmartNIC ARM cores and host x86 cores.**



(a) **Original shuffle.**

(b) **Offload shuffle, but NIC is the bottleneck.**

(c) **SmartShuffle with dynamic migration.**

Fig. 4. **The timeline of data analytic job when with/without SmartShuffle**

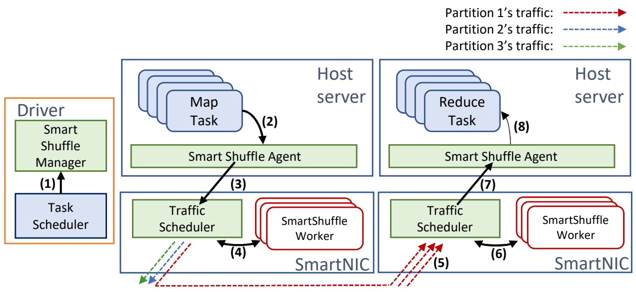


Fig. 5. **SmartShuffle Overview**

Limited CPU. SmartNICs today have fewer and slower cores than the host server. In Figure 3, we measured the performance of three shuffle-related computations on the ARM64-based NIC core (@3.0GHz) using the Stingray NIC [14], and the Intel E5-2650v4 x86 host core (@3.2GHz). We found that the NIC's per-core throughput is generally 1.5×–2.5× slower than that of the host core.

Even with the emergence of new SmartNIC architectures with stronger compute units, such as the NVIDIA Bluefield DPU [37] and the Intel IPU [25], the performance gap between the host and SmartNIC still exists. Recent research on the newest Bluefield DPU shows that the on-NIC embedded processors are weaker than the general-purpose server cores in most computation tasks (as shown in Figures 7 and 8 in [32]). Therefore, when offloading, great care must be taken not to overwhelm the NIC hardware.

Figure 4b demonstrates why a slow SmartNIC can lead to less effective offloading. The example shows that a slow SmartNIC can cause a performance bottleneck, and the reduce task will be hindered by the slow SmartNIC, resulting in an extended job completion time. Our main design challenge is to ensure that on-NIC computation does not block the host while maximizing the amount of work offloaded to the SmartNIC.

3 SMARTSHUFFLE OVERVIEW

SmartShuffle is a high-performance data shuffling service for big data analytics that leverages SmartNIC offloading. SmartShuffle allows big data frameworks to offload a set of common operators that are tightly coupled with the shuffle process. These operators as well as the network communication process are offloaded onto SmartNICs in a dynamic, best-effort manner.

SmartShuffle uses the following ideas to address the challenges in Section 2.3 as well as fully leverage the benefits mentioned in 2.2: (1) To deal with limited on-NIC memory, we develop *partial offload and spilling* to ensure that the offloaded stateful operators fit within a SmartNIC's memory. SmartShuffle identifies appropriate spilling strategies for different types of operators to achieve the best performance. (2) To maximize the amount of work offloaded to the SmartNIC without causing the typically slow SmartNIC cores to become a bottleneck, SmartShuffle utilizes a rate-based dynamic migration policy. This policy monitors the SmartNIC's load and controls the amount of work offloaded to the NIC at runtime (as shown in Figure 4c). (3) To achieve the best I/O efficiency, we employ a *coordinated offload architecture* to leverage both map-side and reduce-side SmartNICs, minimizing both outbound network traffic and inbound NIC-PCIe traffic.

Figure 5 shows the overall architecture of SmartShuffle. With the coordinated offload architecture, both map-side and reduce-side SmartNICs are involved, and the architecture includes four logical components: 1) A **shuffle manager**, which monitors the execution process of map and reduce tasks and controls the shuffle operation across the cluster. 2) The **shuffle agent**, which is launched in each node of the cluster when the SmartShuffle service starts. It manages host-NIC communication and enforces the rate-based dynamic migration policy. By detecting whether the NIC is the performance bottleneck in the shuffle phase, the shuffle agent determines how much work is offloaded to the NIC and how much work is performed on the host. 3) **Shuffle workers**, each of which is an individual on-NIC thread that runs one or more offloaded stateful/stateless operators. The worker enforces *partial offloading*, in which the stateful operator uses spilling to avoid the worker state exceeding the memory limit. 4) The **traffic scheduler**, which is an on-NIC orchestration thread that offloads shuffle's all-to-all network communication process. It also balances incoming network data among different on-NIC worker threads.

Workflow Overview Figure 5 illustrates the basic data flow when applied to an analytics framework such as Spark. At the beginning of each shuffle phase, the SmartShuffle manager fetches control plan information, such as map/reduce task locations and partition strategies, from the framework's task scheduler (① in Figure 5). The SmartShuffle manager then passes these control messages to SmartNICs, which initiate shuffle work threads based on these control messages.

When a map task finishes execution, it requests a DMA-registered memory region from the buffer pool maintained by the SmartShuffle agent. The map task then writes shuffle data to this memory region through the zero-copy interface (② in Figure 5). Then, if the NIC currently is not overloaded, the SmartShuffle agent notifies the map-side SmartNIC's traffic scheduler of the data location in the buffer pool. Otherwise, the SmartShuffle agent launches extra threads on the host to on-load part of the shuffle computation.

The map-side SmartNIC's traffic scheduler fetches the mapper output from the host through DMA reads (③ in Figure 5) and then load-balances the data across parallel shuffle worker threads (④ in Figure 5). Each worker thread runs the partitioning function and other user-defined common operators, such as aggregation, sorting, and filtering. When the on-NIC worker finishes processing a batch of data or when the memory usage of the stateful computation exceeds the SmartNIC limit, the worker spills the processed data to the traffic scheduler, which sends it to the network.

The traffic scheduler manages network transfers and dispatches data partitions to the destination SmartNIC (i.e., where reduce tasks are running). This induces an all-to-all communication process

API	Description
Driver	
registerShuffle(id:Int, loc:LocReduce, parti:BaseOP, op:ReorgOP, spill:SpillStrat)	Register a new shuffle, users need to specify reduce tasks' locations and the base shuffle operator. Optionally, users can specify the on-NIC data-reorganization operator and its spilling strategy.
Map Task	
bufAcquire(size:Int)	Acquire a DMA registered buffer from SmartShuffle Agent.
bufProduce(buf:Buf)	Indicate the data in this buffer is ready to be read by the SmartNIC.
Reduce Task	
bufConsume(id:ReduceTaskId)	Get a ready-to-read shuffle block from SmartShuffle Agent.
bufRelease(buf:Buf)	Release the buffer block back to SmartShuffle Agent.

Table 2. **SmartShuffle APIs**

Shuffle Operators	Description
Base Shuffle Operator	
Partition	Partition tuples according to the key.
Data-reorganization Operator	
Sort	Sort tuples according to the key.
Aggregate	Merge same-key tuples using the reduce function.
Filter	Drop tuples when key/value satisfy conditions.

Table 3. **Shuffle operators defined in SmartShuffle**

between mappers' and reducers' SmartNICs (⑤ in Figure 5). The reduce-side SmartNIC collects all the fragmented partitions from mapper nodes, merges them into large blocks, and assigns them to the on-NIC worker threads (⑥ in Figure 5). The reduce-side shuffle worker receives data from all mapper nodes. It then performs a second round of aggregation, sorting, and filtering across the nodes.

Finally, the traffic scheduler on the reduce-side SmartNIC uses DMA write to place the pre-processed, merged large data blocks (MBs) into the SmartShuffle agent's buffer pool. Once the reduce task launches, it uses large sequential zero-copy reads to directly access these shuffled data blocks.

4 SMARTSHUFFLE DESIGN

In Section 4.1, we describe the APIs and two categories of shuffle operators in SmartShuffle: the basic shuffle operator and data-reorganization operators. Then in Section 4.2, we explain SmartShuffle's *coordinated offload* and how it handles both compute and I/O. Next in Section 4.3, we introduce the *liquid offloading* technique, which consists of two parts: *partial offloading* and *workload migration*. Finally, in Section 4.4, we describe how we load balance workloads among multiple on-NIC workers, and how to support multiple jobs/queries.

4.1 SmartShuffle APIs and Shuffle Operators

We first describe the SmartShuffle APIs and shuffle operators, and how they can be integrated into a framework like Spark.

SmartShuffle APIs. The high-level APIs are shown in Table 2. Before a shuffle phase starts, the Spark framework calls the *registerShuffle()* function to specify the locations of reduce tasks and the operators that will run on the SmartNIC worker.

The SmartShuffle agent exposes two categories of zero-copy APIs for map/reduce tasks to operate on its DMA-registered buffer. To operate on the DMA-registered buffer provided by the SmartShuffle agent, a map task can use the *bufAcquire()* function to acquire an empty buffer and then call *bufProduce()* to notify that the data in this buffer is ready to be posted to the NIC. A reduce task can call *bufConsume()* to obtain a ready-to-read shuffle block and then use *bufRelease()* to return the buffer to the pool when computation is complete. These APIs for tasks are asynchronous which facilitates pipelining of host and SmartNIC processing.

Shuffle Operators. There is a set of operators running on the SmartNIC workers when calling the *registerShuffle()* function. Table 3 shows two categories of shuffle operators. The *base shuffle operator* provides basic shuffle offloading like partitioning mapper outputs according to a key using a hash or a range partition function.

Data-reorganization operators are those that the user optionally wishes to perform right before or after the shuffle phase in an execution plan. These operators rearrange the tuples within each partition based on keys or values to either reduce the volume of network traffic or facilitate reduce task computation.

To illustrate how these operators help with complex queries, we use one TPC-H query [8] as an example. Specifically, we focus on a part of TPC-H query 13, which involves a filter, a join between two tables, a groupBy operation, and a count. Figure 6 shows its execution plan in Spark SQL, which involves three *exchanges* [1], two for join and another for aggregation.

$$customer.join(order, ”c\_custkey” === order(”o\_custkey”)$$
$$\&\&!special(order(“o\_comment'')), “left\_outer'')$$
$$.groupBy(”o\_custkey”).agg(count(”o\_orderkey”))$$

In this example, SmartShuffle offloads the three *exchanges* (red boxes) using the base shuffle operator as well as the *sorting* and *local hash aggregate* (green boxes) because they are either immediately before or after the exchange.

In the original execution plan, after the *exchange*, each partition needs sorting before running merge join. By offloading the sorting to the SmartNIC, the reduce task can directly apply sort-merge join on the ordered data blocks instead of the unordered data. In addition, in the execution plan, each map task does a local aggregation before the *exchange*. By offloading this aggregation, less work is performed on the host thus map tasks can finish earlier.

4.2 Shuffle Offloading

In this section, we first explain how SmartShuffle offloads shuffle operators using the coordinated offload architecture. Then we explain how SmartShuffle offloads network I/O operations using the on-NIC traffic scheduler.

4.2.1 Coordinated Offload Architecture.

Two-level partition: In vanilla Spark, each map task creates partitions based on the number of reduce tasks (shown in Figure 7(a)). If there are n map tasks at each node and a total of m reduce tasks for a job, the total number of partitions in a single mapper node is $n \times m$. Thus, if we consider a job of 1000 nodes, with 50 map tasks and 50 reduce tasks per node, the total number of partitions created on each node would be $\sim 2.5M$. This huge number of partitions can result in small and random shuffle I/O requests. This severely degrades the system performance by: 1) Increasing the number of CPU cycles to deal with these requests [9, 29]. 2) Stressing the network/storage system due to IOPS (I/O operations per second) limitations [48, 57]. 3) Resulting in a bottleneck at the partitioner due to high memory buffer requirements. [43]

To overcome these limitations, SmartShuffle uses the two-level partition, which turns the shuffle process from per-task granularity to per-node granularity (shown in Figure 7(b)). Here, the map-side SmartNIC workers merge the output from multiple map tasks and partition data according to the

[1]An exchange in Spark SQL is the same process as a shuffle in Spark.

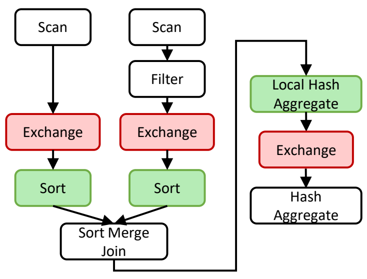
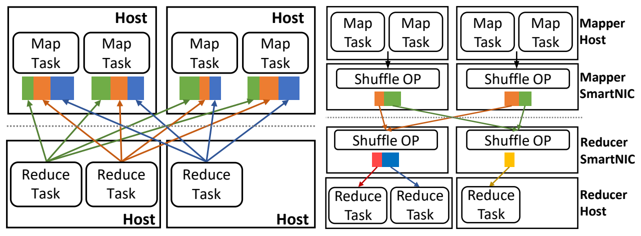


Fig. 6. **TPC-H Q13's Spark SQL execution plan (partial).**



(a) **Standard Shuffle: all-to-all shuffle graph** (b) **Coordinated offload architecture in SmartShuffle**

Fig. 7. **Topology of a standard shuffle and SmartShuffle.**

number of the reducer nodes. The reduce-side SmartNIC gathers all the data partitions sent from multiple map-side SmartNICs, and repartitions the data based on the local reduce task number at the node.

With the same example above, the number of per-node partitions in SmartShuffle is only $1K$. This avoids small and random shuffle I/O requests. Also, this avoids remote block fetch overhead on the reducer, as reduce tasks can directly read large shuffled data blocks from a contiguous memory region prepared by the SmartNIC.

Two-level computation: SmartShuffle further offloads the data-reorganization operator through its coordinated offload architecture. This reduces the host CPU overhead and minimizes the traffic amount to the network and to reduce tasks.

Data pruning operations such as aggregation and filtering on the map-side SmartNIC can reduce outbound shuffle data to the network. Data reduction operations on the reduce-side SmartNIC can reduce the data delivered to the host CPU across multiple data sources.

As for the sorting operator, the reduce-side SmartNIC can benefit from the pre-sorted output from the first level, and further sorting at the reduce-side SmartNIC reduces the workload for reduce tasks running on the host CPUs.

4.2.2 Network Transport Offload. The on-NIC traffic scheduler handles the all-to-all network communication process. SmartShuffle leverages RDMA as the underlying transport due to its low CPU overhead for network I/O [29]. As Figure 8 shows, when an on-NIC worker completes processing its assigned workload, it calls a *spill* function to notify the traffic scheduler that certain data is ready to be sent to the destination endpoints. The *spill* function is triggered when: 1) stateless operators finish processing a batch of data; or 2) the state in a stateful operator exceeds the on-NIC memory limit. All network transfers are batched into large RDMA Reads, as each worker thread batches shuffle data into large chunks for each partition, and spills the chunks only when they reach the spilling threshold.

4.3 Liquid Offloading

Partial offloading and spilling: Stateful operators, such as sort and aggregation, require a large view of the data to be processed, which can exceed the onboard memory capacity of the SmartNIC. To solve this problem, SmartShuffle uses *partial offloading* and *spilling* — the SmartNIC partially aggregates/sorts the data sent from the workers, and does not guarantee a fully aggregated/sorted result. When a worker's memory usage exceeds the limit, SmartShuffle triggers spilling, which sends partially aggregated/sorted results or unprocessed intermediate data to the network/host, i.e., the map-side SmartNIC spills the data to the reduce-side SmartNIC and the reduce-side SmartNIC spills the data to the host. Subsequently, the receiver SmartNIC or reduce workers only need to process the remaining part of the computation (e.g., final aggregation/merge sort), which significantly reduces the workload on the receiver side.

SmartShuffle supports two different spilling strategies:

(1) **Spilling the worker states**: SmartShuffle can spill the current worker's state to free up memory. For example, in the aggregation operator, the worker maintains a hashmap to aggregate incoming tuples. When the size of the hashmap exceeds the spilling threshold, the worker spills all the entries inside the hashmap to the receivers, frees up the hashmap, and creates a new one for future incoming data.

(2) **Spilling the incoming data**: Another spilling strategy is to spill incoming data instead of spilling the worker's state. Again using the aggregation operator as the example, when the hashmap reaches its size limit, SmartShuffle can spill incoming data to the next hop if it cannot be aggregated with the current records inside the hashmap.

One side-effect of spilling incoming data is that the SmartNIC might spill unprocessed data blocks directly to the next hop. For example, in the sorting operator, the SmartNIC might send unsorted data blocks to the host. Therefore, the spilling strategy should be decided according to the reduce-side logic. For example, if the reducer uses TimSort [7], both spilling strategies would work as TimSort does not assume that incoming data blocks are sorted. However, if the reducer uses merge sort, the NIC should choose to spill the worker states. The spilling strategy is defined when the user calls the "registerShuffle" function (Table 2).

Spilling is also triggered when all map tasks finish. SmartNICs flush the remaining data in the worker to the reducer.

Workload Migration: We use *dynamic job migration* to launch extra worker threads on the host and move some of the shuffle computation from the SmartNIC to the host cores. The shuffle agent decides how much work to offload to the NIC and how much to perform on the host. When designing the migration, two factors are considered: (1) the signal that captures the SmartNIC's slowness, and (2) the amount of work to migrate to the host worker.

For (1), the buffer occupancy of the dedicated buffer pool shared by the map tasks and the SmartNIC is used as the signal. High occupancy implies that the SmartNIC is relatively slow in draining the mapper outputs, and more data from the map tasks will likely be blocked due to the lack of buffer slots.

For (2), a threshold for the buffer occupancy can be set to trigger migration. However, this is not effective for batch analytic jobs, as the map tasks generate shuffle data in bursts. For example, the occupancy might suddenly exceed the threshold when a batch of map tasks finishes. However, there is no need to trigger migration if the NIC can consume all the data before the next batch of map tasks start to produce data. To better adapt to batch analysis jobs, we calculate the growth rate of the buffer occupancy over a time window [2]. If the growth rate is positive, the SmartNIC is deemed the system bottleneck, and the shuffle agent launches worker threads on the host. The mapper output is then **partitioned** between the host worker and NIC workers to make the buffer occupancy's growth rate reach 0, using the given ratio R:

$$R = \frac{Mapper\_Produced\_Data\_Over\_TimeWindow}{NIC\_Consumed\_Data\_Over\_TimeWindow} - 1$$

4.4 On-NIC Workload Scheduling

SmartShuffle's on-NIC traffic scheduler launches multiple workers to run the same operators in parallel. The on-NIC workers that run the same set of operators form a *scheduling group*. If there

[2]The window size is dynamically updated according to the average map task duration.

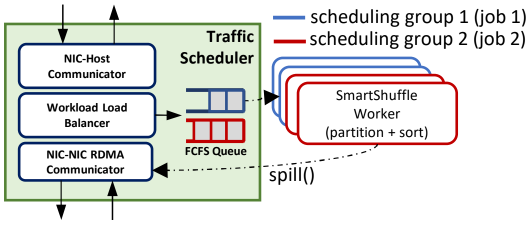

scheduling group 1 (job 1)
scheduling group 2 (job 2)
Traffic Scheduler
NIC-Host Communicator
Workload Load Balancer
NIC-NIC RDMA Communicator
FCFS Queue
SmartShuffle Worker (partition + sort)
spill()

Fig. 8. **SmartShuffle traffic scheduler (map-side).**

are multiple jobs/queries/shuffle phases running on the host, each job/query/shuffle phase has its own *scheduling group*. Figure 8 shows an example of four workers and two scheduling groups, with each worker running a chain of the partition and sort operator. The traffic scheduler load balances the incoming shuffle data among workers within each *scheduling group*; it maintains one centralized first-come-first-serve queue per *scheduling group*.

Note that SmartShuffle applies network I/O merge within a *scheduling group*. For example, even if the output of map tasks from different jobs share the same destination node, SmartShuffle only merges the data that belongs to the same job.

The traffic scheduler controls the load assigned to each worker with fine granularity. When an incoming data block is too large, (for example, the NIC uses a large RDMA read request to fetch the data), the traffic scheduler segments the large data block into appropriate sizes (approximately 100KB) before enqueuing them into the FCFS queue. This is because smaller scheduling units help balance the workload and reduce the impact of stragglers on the NIC.

5 IMPLEMENTATION

We implemented SmartShuffle components on Stingray PS225 SmartNIC [14] with 4K+ lines of C++ code. The host-side SmartShuffle agent and shuffle manager are implemented using 2K+ lines of Scala to interface with Spark 2.4. We implemented SmartShuffle agent as a shuffle manager plugin for Spark, so that the user can switch between SmartShuffle and the default Spark shuffle engine with a single line of configuration. In this section, we focus on explaining how we interface SmartShuffle with the Spark.

Accelerating Shuffle: In Spark, some RDD functions, e.g., *groupByKey, sortByKey and distinct* introduce a shuffle. For these functions, SmartShuffle offloads the shuffle operators onto the SmartNIC. Furthermore, SmartShuffle chooses to offload different types of data-reorganization operators for different RDD functions; e.g., we offload the aggregation operator for *reduceByKey, groupByKey, aggregateByKey* functions, sorting operator for *sortByKey, repartitionAndSortWithinPartitions* functions, and filtering operator for the *distinct* function.

Due to *liquid offload*, SmartNIC may give partially sorted/aggregated results to Spark reducers. We argue that reducers can directly benefit from these partially computed results. First, the default sort algorithm in Spark on the reduce side is TimSort [7]. It does not assume that the incoming data blocks are sorted, but it can take advantage of the ordering of some of the sorted sub-blocks. By providing partially sorted data segments to the reducer, it can experience a performance gain. Second, Spark implements aggregation and distinct functions using a hashmap, and by reducing the amount of data sent to the host via on-NIC pre-aggregation and pre-filtering, the overall hashmap inserting time in the Spark reducer is minimized.

Obtain the Reduce Task Location: In SmartShuffle, the shuffle phase starts once mapper tasks start to generate outputs, and when the aggregated data size exceeds the NIC memory capacity, the SmartNIC spills the data to the reduce tasks. However, in Spark, reduce tasks are scheduled only when most of the mapper tasks are finished, which might be too late for SmartShuffle. In our

Shuffle Functions	Shuffle Manager in Spark	Map Stage CPU Time (s)	Reduce Stage CPU Time (s)	Total Job Duration (s)	Offloaded Operator
partitionByKey	Spark	1010.8	590.4	27.5	N/A
	SmartShuffle	758.0 (-24%)	580.5	25.3	Partition
reduceByKey	Spark	1668.1	107.3	21.2	N/A
	SmartShuffle	1203.2 (-28%)	40.9 (-63%)	15.7 (-26%)	Partition + Aggregation
sortByKey	Spark	1381.9	1403.9	50.1	N/A
	SmartShuffle	987.5 (-28%)	912.1 (-34%)	36.2 (-28%)	Partition + Sorting
distinct	Spark	1549.4	77.8	24.1	N/A
	SmartShuffle	1296.3 (-16%)	24.0 (-68%)	17.4 (-27%)	Partition + Filtering
join	Spark	251.4/652.6	1597.8	18.1	N/A
	SmartShuffle	169.8/469.2	1472.3	15.46 (-15%)	Partition

Table 4. **Spark shuffle functions performance breakdown. Map stage and reduce stage's total CPU time is obtained by summing the CPU time of all worker cores.**

implementation, we modified the Spark task scheduler to allow SmartShuffle obtain reduce tasks' location earlier. As shown in Figure 5, the Spark task scheduler provides reduce task placement information to SmartShuffle once some map tasks start to generate outputs.

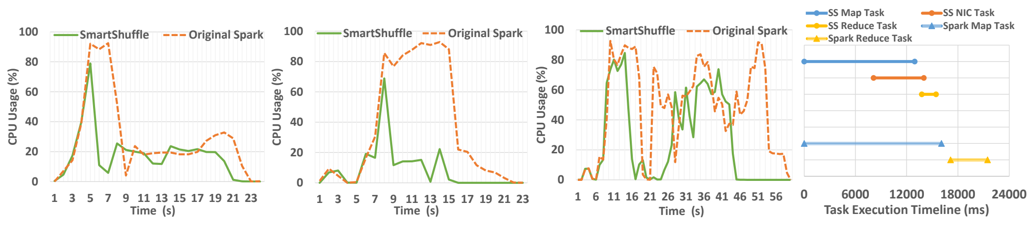
There are alternate design options. If it is difficult to obtain reduce task location in advance, SmartShuffle could send the spilled data to intermediate storage/memory nodes. After the reducer task launches, it fetches the shuffle data from these storage/memory nodes. Using intermediate nodes to store the shuffle data has been a common design choice in large-scale clusters. For example, LinkedIn uses such design to improve system scalability [48], and SmartShuffle can fit into such design to further improve shuffle performance. As another option, after SmartShuffle spills data to intermediate storage nodes, the storage nodes could coordinate with a task scheduler to launch reduce tasks close to their shuffle data location. In such cases, we can further use spilled data properties to drive when/what computation tasks run. Such a "data-driven" approach has been investigated recently [21].

Shuffled Data Persistency: The shuffled data in Spark is first partitioned and then materialized to storage at the mapper node before the shuffle phase. However, SmartShuffle offloads partitioning to the NIC and since the NIC may not have enough storage to persist partitions, SmartShuffle persists shuffled data on the reducer-side. Specifically, the reducer-side shuffle agent materializes the pre-processed shuffle blocks using large, sequential disk writes. After reduce tasks are launched, they retrieve the materialized shuffle data using large sequential disk reads and perform computations on them.

The trade-off here is that more work needs to be repeated if a reducer node fails, as the results of the incoming map tasks will be lost. If intermediate storage nodes are used, SmartShuffle could persist the shuffle data in those storage nodes, similar to the approach used by Whiz [21] and Magenet [48].

6 EVALUATION

In this section, we evaluate SmartShuffle on Spark using a combination of the Hadoop BigData [3] benchmark and the TPC-H benchmark [8]. We present the evaluation results in two parts: First, Section 6.2 presents the performance of the building blocks in SmartShuffle. We use the BigData benchmark to demonstrate: 1) Various Spark shuffle functions can benefit from SmartNIC offload, leading to lower job completion time and higher host CPU efficiency. 2) Workload migration avoids the SmartNIC becoming a performance bottleneck for non-line rate operators. 3) The coordinated offload architecture achieves better aggregation performance compared to Spark. 4) Different spilling thresholds can influence the system performance.



(a) **partitionByKey** (b) **reduceByKey** (c) **sortByKey** (d) **reduceByKey timeline**

Fig. 9. **(a) - (c) shows CPU profiling of the original Spark and Spark with SmartShuffle. (d) shows different types of tasks' average execution time and launch time when running reduceByKey**

Worker	Parti only		Parti + Agg		Parti + Sort	
Num	8B	32B	8B	32B	8B	32B
1	11.96	10.8	1.46	1.82	0.47	0.38
4	26.18	27.15	4.13	4.73	1.76	1.17
8	28.12	29.44	6.08	7.20	3.12	1.95

Table 5. **For different types of computation, NIC workers' throughput (in Gbps) under 8B/32B key size. We use a fixed value size (4B Integer) and fixed spilling threshold (16MB).**

Second, Section 6.3 presents the end-to-end performance of SmartShuffle using a standard benchmark, TPC-H. Here, we also measure the I/O aggregation performance achieved by SmartShuffle.

6.1 Testbed and Methodology

We evaluate SmartShuffle in a testbed of 4 Dell PowerEdge R630 servers. Each server has 20 cores (Intel E5-2650 v4 at 2.3GHz) and 40 hyperthreads, with 128 GB RAM. Every server is equipped with a 25GbE Stingray PS225 SmartNIC [14] which has an 8-core ARM A72 processor at 3.0GHz and 16GB NIC RAM. To support high-performance RDMA communication between NICs, we turn on Priority-based Flow Control (PFC), Explicit Congestion Notification (ECN marking), and DCTCP congestion control in both the switch and the SmartNIC [4]. These configurations allow all RoCE packets to run on a lossless network, with low-level hardware congestion management in the switch and NICs.

We use one Spark executor per machine, each with 64GB of memory and 40 logical cores. We also disable data compression during the shuffle, which we found compression will slow down the job due to CPU overhead. We run our experiments multiple times to ensure our results are reported based on a warm disk cache.

6.2 Building Blocks of SmartShuffle

In this section, we generate a 20 GB unsorted random sequence of records using the Hadoop BigData generator [3]. We first show the overall performance gain, and then investigate how each design component affects system performance.

Shuffle Function Performance: We evaluate SmartShuffle's performance using different shuffle functions shown in Table 4. We also report the host CPU utilization for a subset of the runs (Figure 9). Overall, SmartShuffle reduces job completion time by 10% to 30% compared to Spark. Additionally, SmartShuffle significantly reduces host CPU usage.

For the *partitionByKey* function, we offload the hash partition operation onto the NIC. The result shows that the map stage executor runtime decreases by 24% since all map tasks avoid shuffle-related computation and can complete earlier. This can be directly observed in Figure 9a.

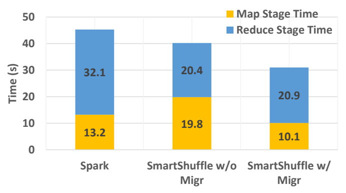
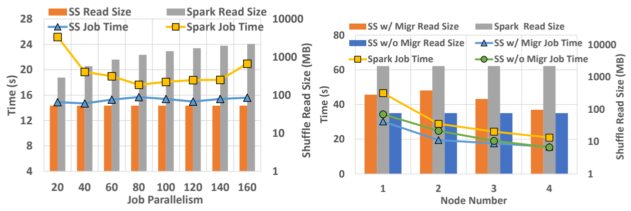


Fig. 10. ***SortByKey* performance with/without migration.**



(a) **Performance under different job parallelism**

(b) **Performance under different node number.**

Fig. 11. ***reduceByKey* performance in Spark and SmartShuffle (SS). We measure the job runtime and total shuffle read size in the shuffle stage.**

From 5s to 9s, which is the tail of the map stage, the CPU usage for the *partitionByKey* function is reduced.

The *reduceByKey* function aggregates the values corresponding to each unique key using *sum()*. SmartShuffle offloads both the partition and the aggregation operator for this function. In *reduceByKey*, Spark performs an aggregation locally on each mapper before sending the results to a reducer. This local aggregation requires the map tasks to spend significant CPU cycles (9s to 15s in Figure 9b). In SmartShuffle, the aggregation happens in the SmartNIC, which allows the map task to bypass the aggregation computation. Thus, the map stage executor runtime is reduced by 31% in SmartShuffle; there is also a substantial CPU usage reduction (Figure 9b). The reduce stage's performance is also improved by SmartShuffle because Spark does local aggregation only within each map task, but SmartShuffle does cluster-wide aggregation with its coordinated offload architecture, which has a better aggregation efficacy. In SmartShuffle, far fewer shuffle reads and computations occur in the reducer. In total, the *reduceByKey* function shows over 35% performance improvement.

The *sortByKey* first repartitions the RDD according to the given partitioner, then sorts each partition by keys. SmartShuffle offloads the partitioning operator and the sort operator, and demonstrates over 28% performance improvement. In *sortByKey*, Spark's reducer uses TimSort [7], which can benefit from partial sorted blocks from the SmartNICs. SmartShuffle improves the reduce stage performance by 34%.

As for the *distinct* function, SmartShuffle offloads the partition operator and the filtering operators. The filtering operator in the SmartNIC removes duplicated entries using a hashmap. The reduce stage runtime can be greatly reduced (by 68%), since the SmartNIC has filtered out duplicates. The total performance improvement is about 27%.

For the *join* function, the host CPU hash joins two tables based on the given join expression. The hash join includes two steps: 1) co-partition the two tables' tuples using a hash function and 2) join the corresponding partitions of the two tables. In our current implementation, we only offload the partition function for the hash join, and report the runtime of the two map stages and one reduce stage. In total, SmartShuffle improved join performance by 15%. We do not show the CPU usage for the *distinct* and *join* functions because they have similar behaviors as *reduceByKey* and *partitionBykey*, respectively (Figure 9).

Understanding Performance Gain: Figure 9d shows tasks' execution timeline with the *reduceByKey* function, which helps us to further understand SmartShuffle's performance gain. First, by benefiting from computation offload, SmartShuffle accelerates the execution pipeline. Both map and reduce tasks' average runtime is reduced, especially the reduce task, as on-NIC aggregation significantly reduces the amount of work a reducer needs to perform. Besides, although the on-NIC processing is slower than the host, SmartShuffleeffectively overlaps NIC computation with host computation, which efficiently hides NIC's processing latency. This helps reduce tasks launch much earlier than in vanilla spark, helping further improve job run time.

NIC Worker and Workload Migration: When the NIC's throughput is slower than both the mapper and line rate, workload migration is triggered. Figure 10, shows how workload migration can significantly improve the performance of SmartShuffle. In the *sortByKey* benchmark, NIC's sorting throughput cannot catch up with the map task's data generation speed, and the buffer occupancy growth rate is positive, triggering migration. With workload migration, the host will launch extra worker threads to process the mapper output on behalf of the NIC when it is congested. As shown in Figure 10, workload migration can significantly improve the performance of SmartShuffle. Without it, the map stage can be slower than in the original Spark.

We also measure NIC workers' throughput under different key sizes and computation types. As shown in Table 5, SmartShuffle uses 4 worker threads to achieve the 25 Gbps line rate for partitioning. However, for stateful operations, the NIC's throughput cannot hit the line rate. For example, for partitioning plus aggregation, the throughput is 4Gbps for 4 threads and 6Gbps for 8 threads; for partitioning plus sorting, the throughput is 1Gbps for 4 threads and 2-3Gbps for 8 threads.

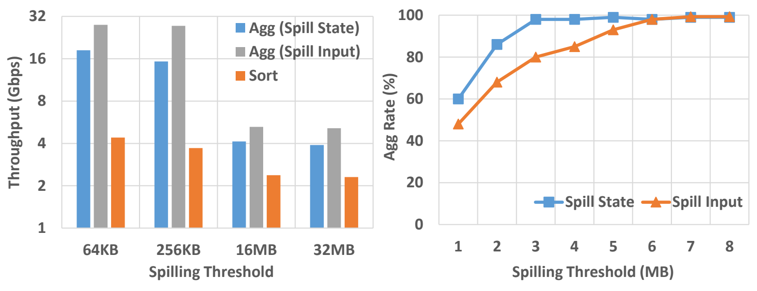
Two-level Aggregation: We now show that SmartShuffle can achieve a high aggregation rate using its coordinated offload architecture. We define *data reduction size* as the map stage's output size minus the reduce stage's input size, and *aggregation rate* as the ratio of SmartShuffle's data reduction size to the ideal data reduction size [3]. That is, the higher the aggregation rate, the more data is pre-aggregated within the SmartNIC. If the aggregation rate is 100%, meaning the shuffle data is fully aggregated before sending to the reducer. In Figure 10a, we vary the job parallelism for the *reduceByKey* function and compare the job completion time and total shuffle read size in the reduce stage.

In Spark, when the job parallelism is low, the system cannot fully utilize the CPU resources. Therefore, the performance is improved when we increase the job parallelism from 20 to 40. However, the performance drops when the parallelism grows from 80 to 160. This is because Spark's pre-aggregator aggregates shuffle data at task granularity, and when the map task parallelism increases, the aggregation rate decreases. More shuffle data will be transferred to the reducers, further degrading job performance.

In contrast, SmartShuffle always achieves a good aggregation rate regardless of the job parallelism, and the job completion time remains stable under different parallelism levels. Note that SmartShuffle achieves high performance even with low parallelism because the NIC reduces the host CPU load; thus we can achieve the same performance using fewer cores.

Figure 10b shows how SmartShuffle's aggregation performance changes with different node counts, and how migration influences the aggregation rate. We vary the node number in the cluster from 1 to 4 while keeping the total job parallelism the same. The lines in the figure show that under different node counts, SmartShuffle always performs better than Spark, and the gap widens as the node count decreases. When the node count is small, workload migration takes effect in SmartShuffle, as each NIC needs to aggregate a larger range of keys making it a bottleneck. The bars in Figure 10b show the shuffle read size. For both Spark and SmartShuffle without migration, the read size remains unchanged as the node count increases. When migration is turned on, the shuffle read size increases compared to no-migration SmartShuffle. This is because the migrated data bypasses the two-level computation and only does single-level aggregation in the shuffle agent;

[3]Ideal data reduction is calculated using map stage's output size minus job's final output size.



(a) **Total throughput under different spilling thresholds.**

(b) **Aggregation rate under different spilling thresholds.**

Fig. 12. **Spilling performance with 4 workers.**

as such, the aggregation rate is not optimized. However, even when migration is triggered, the aggregation rate in SmartShuffle is still higher than Spark.

Spilling Overhead: We first evaluate how spilling influences stateful operators' throughput. In Figure 12a we show operator throughput under different spilling thresholds. We test two spilling strategies for the aggregation operator: 1) spilling the worker state; 2) spilling incoming tuples. For sorting, the spilling strategy is to spill only the worker state.

The result in Figure 12a shows that all operators have higher throughput when the spilling threshold is low. This is because a large spilling threshold means more state is maintained inside the worker; thus the worker's working set will suffer from more L1/L2 cache misses. In addition, the figure shows that spilling the input data has higher throughput than spilling the worker state. This is because for the aggregation operators, spilling the worker state incurs the extra overhead of destroying/reconstructing the hashmap.

Although increasing the spilling threshold may downgrade worker performance, it has the advantage of improving the aggregation/sorting rate. As shown in Figure 12b, we change the spilling threshold for each worker and measure the aggregation rate. When the spilling threshold increases, the aggregation rate also increases, since the NIC worker uses more state to build a bigger hashmap to store more distinct values. When the spilling threshold is larger than the key range, no spilling would happen, and the NIC can aggregate all tuples that have the same keys. Figure 12b also shows that spilling the input data yields worse aggregation rate than spilling the worker state. This is because spilling the input data creates many not-fully-aggregated data blocks and downgrades the aggregation rate.

6.3 Query Performance

We run the TPC-H benchmark with a scale factor of 50 and evaluate Query 1-6 and Query 11-14. We haven't integrated SmartShuffle with Spark SQL [4], so we implement all TPC-H queries using the Spark RDD API. Table 6 shows the shuffle RDD functions that have been used in each query.

TPC-H query performance: Figure 13 compares the performance of vanilla Spark, Spark with SmartShuffle, and Spark with RDMA plugins [1] in terms of job completion time. The results show that, except for query 6, SmartShuffle brings a 15% to 40% reduction in job runtime compared to vanilla Spark. SmartShuffle does not bring any performance improvement for query 6, as only a

[4]Spark SQL uses the DataFrames API, and requires extra support because 1) Spark SQL's query optimizer changes the execution plan and uses different sets of operators to achieve shuffle. We haven't integrated SmartShuffle with the SQL query optimizer yet. 2) Currently, SmartShuffle only supports serialized shuffle data in the row-by-row fashion, which is the default in Spark. However, Spark SQL uses its own columnar storage and is quite different from Spark's.

Query ID	Involved Shuffle Functions	Query ID	Involved Shuffle Functions
Q1	reduceByKey	Q6	reduceByKey
Q2	join, sortByKey	Q11	join, reduceByKey, sortByKey
Q3	join, reduceByKey, sortByKey	Q12	join, reduceByKey
Q4	join, reduceByKey, distinct	Q13	join, reduceByKey
Q5	join, reduceByKey	Q14	join

Table 6. **Involved shuffle functions in TPC-H queries**

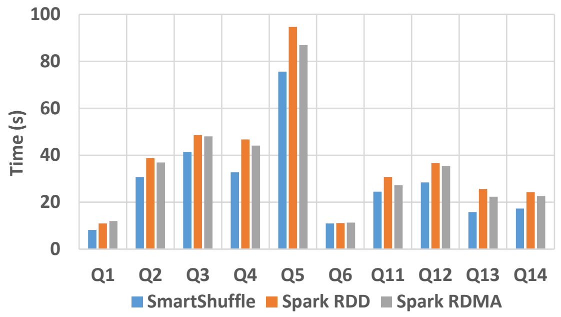


Fig. 13. **SmartShuffle's TPC-H benchmark performance. The job completion time is normalized to the Spark RDD performance.**

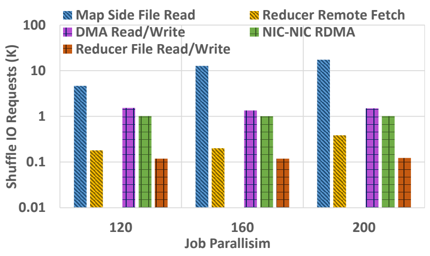


Fig. 14. **Num. shuffle I/O reqs for the join op in TPC-H Q13.**

very small amount of data (<20 KB) is shuffled in this query. The result also shows that SmartShuffle doesn't have overhead on the system when queries have no shuffle operation.

Spark RDMA can improve the performance of the baseline Spark by reducing networking-related costs, but the performance improvement is less than 10%. In our cluster, we found that Spark RDMA's performance gain is relatively small. A similar observation can be found in previous works [6] which find that Spark RDMA only brings a marginal performance gain of 3-4%.

SmartShuffle has better performance than Spark RDMA. This is because Spark RDMA improves the shuffle performance by performing only data transfers over RDMA, but SmartShuffle uses a more holistic approach and leverages computation offloads. The Spark job benefits not only from the kernel-bypass networking, but also from computation offloading and data aggregation.

I/O efficiency: To show that SmartShuffle achieves better I/O efficiency, we measure the total number of I/O requests during the shuffle. Figure 14 shows the total number of I/O requests in Q13. To make a fair comparison, we compared the I/O request count for the join operation, as SmartShuffle does not aggregate the data for join, thus the total transferred bytes are the same in Spark and SmartShuffle.

Spark's I/O overhead includes shuffle file reads on the mapper side, and remote network fetches on the reducer side. SmartShuffle's I/O overhead includes all the DMA/RDMA requests and the shuffle file writes on the reducer side.

The result shows that Spark's total I/O request count grows quadratically with a job's parallelism. While with SmartShuffle, the I/O request count is not influenced by parallelism, as SmartShuffle does node-level I/O merging in the SmartNIC. Even though SmartShuffle incurs additional I/O requests to DMA data to/from NIC cores, the total I/O request count is still much smaller than Spark. Therefore, SmartShuffle greatly reduces the number of I/O requests during shuffle.

Comparing to Spark SQL and future work: Spark SQL is generally known to have better performance than Spark RDD. This is because Spark SQL applies sophisticated join algorithms and uses the Catalyst optimizer to plan queries [10]. Additionally, the serializer in Spark SQL is faster than Spark's default serializer. Figure 15 compares SmartShuffle's performance with Spark SQL.

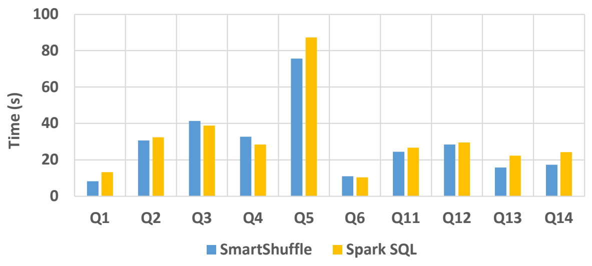


Fig. 15. **Comparing SmartShuffle with Spark SQL.**

Compared to Spark SQL, SmartShuffle has lower performance than Spark SQL in Q3 and Q4. This is because Spark SQL benefits from its optimizer and serializer, and these optimizations play a heavy role in these queries' performance. However, and interestingly, SmartShuffle brings a 5% to 35% reduction in runtime compared to SparkSQL for the many more remaining queries – Q1, Q2, Q5, and Q11-Q14. Thus SmartShuffle can offer benefits even without the help of a powerful optimizer/serializer in many cases.

The goal of SmartShuffle – to offload shuffle computation – is orthogonal to the query optimizer whose focus is on the logical query plan. Thus we believe that when co-designed with a query optimizer and serializer, SmartShuffle can bring very significant across-the-board benefits for all queries. Integrating SmartNIC offload with the query optimizer – a challenging problem in itself – is subject for future work.

6.4 Evaluation Takeaways

Our evaluation suggests that in the data analytic context, SmartShuffle helps to reduce the CPU cost and network cost.

Reduce CPU Cost: Our results show that by offloading the data-reorganization operators (filter/aggregation/sort) to the NIC, SmartShuffle improves data-intensive application's performance. This is because offloading CPU-bounded computation to the SmartNIC reduces host CPU cycles, and the reduced host cycles can be used to perform other host tasks in the execution pipeline, therefore whole job completion time can be improved.

Reduce Network Cost: Our results shows that SmartNIC is a great vantage point to perform host-level I/O aggregation and merging. With SmartShuffle, I/O request counts, mapper-side outbound network traffic and reducer-side NIC-PCIe traffic are all greatly reduced; therefore SmartShuffle improves performance by removing network bottlenecks.

7 DISCUSSION

FPGA NICs: FPGA NICs are alternative options for offloading shuffle operations, but they have different strengths and weaknesses compared to SoC NICs. FPGA devices can offer significantly higher performance for computations that involve deterministic program logic and regular data structures, such as pattern matching, filtering, and aggregation. However, certain operations in SmartShuffle, such as sorting and serialization/deserialization, involve irregular computations with dynamic data structures that necessitate random memory access patterns. Synthesizing these irregular computations on an FPGA can be challenging [23].

Compared with Using Host Cores: Using SoC SmartNICs is an attractive approach for enhancing system performance. Compared to using host cores, using NIC cores has two key benefits. Firstly, SoC NICs are specialized for I/O intensive tasks, and their on-chip ASIC-based network accelerators (such as Nitro hardware switching in the Stingray NIC [14]) deliver better performance and efficiency compared to general CPU cores. Secondly, SmartNICs benefit from their simple microarchitecture,

resulting in greater energy- and cost-efficiency than host servers when offloading computations with low IPC (instruction-per-cycle) and high MPKI (L2 cache misses per kilo-instructions) [33, 35, 47].
Different Bottlenecks: In Spark, CPU is the primary bottleneck [40]. SmartShuffle's computation offload and shuffle data pruning can alleviate this bottleneck and greatly improve performance. When integrating SmartShuffle with different query engines [11, 18, 46], the impact of SmartShuffle may vary depending on the specific bottleneck of the system/workload (e.g., CPU, disk, memory, network). For instance, if the bottleneck is the disk, the primary benefits of SmartShuffle are expected to come from I/O merging and data pruning rather than computation offload.

8 RELATED WORK

Using Network Accelerators in Data Analytic Jobs: There has been great research interest in improving the performance of data analytic jobs using network accelerators [27, 38, 39, 53]. JumpGate [38, 39] enables existing systems to execute SQL queries on network accelerators. At the heart of JumpGate is a compiler that generates code for the operations in a query to run on a network accelerator. However, JumpGate only improves performance when the network accelerator can outperform the client system on the offloaded operations (see Section 6 of the paper). JumpGate fails to improve performance when using a wimpy NIC core because it always offloads the full operator to the network accelerator, which can become the performance bottleneck. In contrast, SmartShuffle's liquid offloading technique directly targets this problem.

Cheetah [53] accelerates database queries using switch pruning. Cheetah is aimed narrowly at offloading the filter operation. In contrast, SmartShuffle is able to handle data-intensive shuffle functions; the programmable switch cannot load these functions due to the inflexible programming model and resource/timing constraints. We note that co-designing NIC and switch offloads to jointly accelerate database queries is an interesting and open question.
SmartNIC Programming Model: Many previous SmartNIC works focus on providing a programmable framework for SmartNICs [26, 28, 30]. iPipe [33] is an actor-based distributed system that offloads distributed applications to the SmartNICs. Floem [41] is a programming system that uses the data-flow language to express the on-NIC packet processing. These approaches are not optimized for data-intensive applications. In Floem, offloaded computations are stationary, and NICs can easily become the bottleneck when tasks run out of the on-NIC CPU/memory. As such Floem mainly supports per-packet processing tasks, which can run at line rate with minimal state maintained. iPipe does not consider limited memory, and iPipe's workload migration policy is designed to minimize tail latency and does not fit batch analytics.

9 CONCLUSION

Shuffle is widely used in distributed data intensive applications and is one of the most resource-intensive and time-consuming operations. We propose SmartShuffle, an optimized shuffle service for big-data analytics frameworks. SmartShuffle helps offload the shuffle-related tasks and network communication functions. It incorporates several key building blocks – that we believe can be extended to other data-intensive workloads – to make offloaded computation fit within NIC constraints without diminishing the overall performance. SmartShuffle improves job runs times by 40% job while improving host's CPU and I/O efficiency.

10 ACKNOWLEDGEMENTS:

We thank our shepherd, Ran Ben Basat, and the anonymous Sigmetrics reviewers for their feedback. We thank CloudLab [44] for providing equipment used to test and evaluate SmartShuffle. This research was supported by NSF awards CNS-2106199. Jiaxin Lin is supported by a Meta PhD Research Fellowship.

REFERENCES

[1] Accelerated Spark on Azure: Seamless and Scalable Hardware Offloads in the Cloud. https://github.com/Mellanox/SparkRDMA.

[2] Aws nitro system. https://aws.amazon.com/cn/ec2/nitro/.

[3] Hadoop randomtextwriter. https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/RandomTextWriter.html.

[4] Netxtreme-e linux roce configuration guide. https://docs.broadcom.com/doc/netxtreme-e-roce-configuration-guide.

[5] Nvidia collective communications library. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html.

[6] Spark shuffle: Sparkrdma vs crail. https://craillabs.github.io/blog/2017/11/rdmashuffle.html.

[7] Tim sort. http://wiki.c2.com/?TimSort.

[8] Tpc-h benchmark. http://www.tpc.org/tpch/.

[9] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the network for scatter-gather workloads. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 267–280, 2020.

[10] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[11] Apache Arrow. Ballista: Distributed sql query engine, built on apache arrow. https://github.com/apache/arrow-ballista.

[12] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1463–1475, 2015.

[13] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5):517–528, 2017.

[14] Broadcom. Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[16] Cavium Corporation. Cavium LiquidIO. http://www.cavium.com/pdfFiles/LiquidIO_Server_Adapters_PB_Rev1.2.pdf.

[17] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98–109, 2011.

[18] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[21] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. Whiz: A fast and flexible data analytics system. *arXiv preprint arXiv:1703.10272*, 2017.

[22] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 681–693, 2020.

[23] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H Peter Hofstee. Fpga acceleration for big data analytics: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, 2021.

[24] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[25] Intel. Intel Unveils Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html.

[26] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. *arXiv preprint arXiv:2106.07102*, 2021.

[28] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. Uno: Uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 2017.

[29] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 225–236, 2018.

[30] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[31] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[32] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. Performance characteristics of the bluefield-2 smartnic. *arXiv preprint arXiv:2105.06619*, 2021.

[33] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[34] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 795–809, 2017.

[35] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *USENIX annual technical conference*, pages 363–378, 2019.

[36] Mellanox Technologies. Mellanox BlueField SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[37] Mellanox Technologies. NVIDIA Mellanox BlueField DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[38] Craig Mustard, Swati Goswami, Niloofar Gharavi, Joel Nider, Ivan Beschastnikh, and Alexandra Fedorova. Jumpgate: automating integration of network connected accelerators. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[39] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. Jumpgate: In-network processing as a service for data analytics. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[40] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[41] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 663–679, 2018.

[42] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[43] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. Hyper dimension shuffle: Efficient data repartition at petabyte scale in scope. *Proceedings of the VLDB Endowment*, 12(10):1113–1125, 2019.

[44] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[45] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205. IEEE, 2016.

[46] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, et al. F1 query: Declarative querying at scale. *Proceedings of the VLDB Endowment*, 11(12):1835–1848, 2018.

[47] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 740–755, 2021.

[48] Min Shen, Ye Zhou, and Chandni Singh. Magnet: push-based shuffle service for large-scale data processing. *Proceedings of the VLDB Endowment*, 13(12):3382–3395, 2020.

[49] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the*

applications, technologies, architectures, and protocols for computer communication, pages 708–721, 2020.

[50] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 767–782, 2019.

[51] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[52] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. srdma–efficient nic-based authentication and encryption for remote direct memory access. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*, pages 691–704, 2020.

[53] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2407–2422, 2020.

[54] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir] relevance of network performance for data processing. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[55] Li Yilong, Jin Park Seo, and Ousterhout John. Millisort and milliquery: Large-scale data-intensive computing in milliseconds. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[57] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[58] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 295–308, 2012.

Received February 2023; revised March 2023; accepted April 2023