Jumpgate: Automating Integration of Network Connected Accelerators

Craig Mustard
University of British Columbia
craigm@ece.ubc.ca

Swati Goswami
University of British Columbia
sggoswam@cs.ubc.ca

Niloofar Gharavi
University of British Columbia
niloo987@student.ubc.ca

Joel Nider
University of British Columbia
joel@ece.ubc.ca

Ivan Beschastnikh
University of British Columbia
bestchai@cs.ubc.ca

Alexandra Fedorova
University of British Columbia
sasha@ece.ubc.ca

ABSTRACT

Network-connected accelerators (NCA), such as programmable switches, ASICs, and FPGAs can speed up operations in data analytics. But so far, integration of NCAs into data analytics systems required manual effort.

We present Jumpgate, a system that simplifies integration of existing NCA code into data analytics systems, such as Apache Spark or Presto. Jumpgate places most of the integration code into the analytics system, which needs to be written *once*, leaving NCA programmers to write only a couple hundred lines of code to integrate new NCAs. Jumpgate relies on dataflow graphs that most analytics systems use internally, and takes care of the invocation of NCAs, the necessary format conversion, and orchestration of their execution via novel *staged network pipelines*.

Our implementation of Jumpgate in Apache Spark made it possible, for the first time, to study the benefits and drawbacks of using NCAs across the entire range of queries in the TPC-DS benchmark. Since we lack hardware that can accelerate all analytics operations, we implemented NCAs in software. We report on how and when analytics workloads will benefit from NCAs to motivate future designs.

ACM Reference Format:
Craig Mustard, Swati Goswami, Niloofar Gharavi, Joel Nider, Ivan Beschastnikh, and Alexandra Fedorova. 2021. Jumpgate: Automating Integration of Network Connected Accelerators. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463770

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463770

1 INTRODUCTION

As Dennard scaling and Moore's law reach their limits, system designers are turning to domain-specific accelerators. *Network-connected accelerators* (NCA) show promise for data analytics. NCAs implemented on top of programmable switches, such as Cheetah, NetAccel, PPS and Sonata, showed a 2-8× speedup for a join-and-group-by operation, a 6.5× speedup for string search and a 3-7 orders of magnitude reduction in network traffic [15, 20, 31, 53]. More generally, FPGAs, SmartNICs and network-based software accelerators demonstrate speedups of 2-10× for analytics (§2).

Two steps are required to use an NCA inside an analytics system: (1) writing the NCA code itself, and (2) *integrating* it into the analytics system. This paper focuses on the second step – integration, which up to this point has been done manually, for each new NCA.

Integration involves conversion of input data into a format suitable to the NCA, the invocation of the NCA, and the orchestration of the execution and data exchange. NCAs need format conversion, because they are usually limited in resources and cannot parse common storage formats. They need orchestration because they have limited storage and (typically) must stream data as it is made available by the sender and ingested by the recipient. Performing these steps manually for every new NCA and every analytics system puts unnecessary burden on programmers and slows development and adoption of these promising accelerators.

We present Jumpgate, a system that simplifies NCA integration. The key insight in Jumpgate's design is that data flow graphs, used internally by query processing engines in analytics systems, provide a convenient abstraction for using NCAs. At the heart of Jumpgate is a compiler that (upon query submission) generates the appropriate format converters for the available NCAs, specializes the existing NCA code for the operations in the query, and ensures NCAs can communicate with one-another and the analytics system. To process a query, Jumpgate orchestrates the execution of NCAs, converters, and the analytics system using an execution paradigm called *staged network pipelines*. Jumpgate divides the integration effort between analytics system programmers and NCA programmers: We add Jumpgate to Apache Spark with 2200 LoC, and to Presto with 1870 LoC. (This paper focuses on the Spark implementation). NCA programmers creating new accelerators need to write only a few hundred lines of Jumpgate integration code: 186-609 LoC, in our experience.

Using NCAs for analytics presents trade-offs that need to be quantified across many queries: data formats for NCAs can inflate intermediate data volumes, and format converters and orchestration add overheads. In the past, it was difficult to study these overheads at scale, because manual integration limited how many queries could use NCAs. For example, the most substantial work known to us studied only 9 queries [53]. With Jumpgate we are able to offload operations in all of *99* TPC-DS[1] queries and study the implications of using NCAs across a wide range of workloads.

Since we had limited access to hardware NCAs (only a simple *aggregation* operation implemented in a programmable switch was available to us), we implemented five NCAs in software to ensure good coverage; in the end, we were able to offload 60% of TPC-DS operation to software NCAs and run integer aggregation queries on the programmable switch. We found that using NCAs creates a trade-off: instead of materializing the data in memory of an analytics system, a similar volume of data is sent on the network. Offloading is beneficial when NCAs can reduce the volume of data received by the analytics system (often by orders of magnitude in our experiments), which reduces work the client system must perform. Performance improves when the network and NCAs transfer and process data faster than the analytics system: our studied NCAs can accelerate certain queries by $1.12 - 3\times$ in these conditions. Finally, most NCA pipelines we studied were bottlenecked on converting input data to a format suitable for NCAs, indicating that accelerating format conversion should be a future research direction.

In summary, our key contributions are the architecture of Jumpgate, its implementation in Apache Spark, and the study of pros and cons of using NCAs to execute TPC-DS.

2 BACKGROUND AND RELATED WORK

Data analytics engines translate a user's query into a *logical* graph of operations, and then map it to a *physical* graph of implementations. Analytics dataflow graphs are so similar between systems that they can be translated from one system to another [12, 37, 42, 43]. Jumpgate builds on this property and translates a dataflow graph into a graph of NCAs from a set of known NCA implementations. This design makes it easy to integrate Jumpgate with various analytics engines.

Analytics systems leverage the dataflow graph for distributed execution on *worker nodes* in a cluster. At first glance, it seems trivial to swap a conventional node for an NCA, but in reality this requires non-trivial integration effort, because NCAs are built with unconventional and typically limited processing hardware and constrained storage.

Programmable dataplane switches, equipped with custom ASICs, were used to speed up analytics tasks by up to 10-1000× [13, 15, 21, 22, 31, 48, 52], but they can store at most tens of MB of intermediate data and process no more than ≈100s of bytes per packet. *Storage accelerators* [10, 14, 19, 23, 26, 55], *SmartNICs*, and FPGA-based accelerators have their own constraints [25, 33, 34, 44].

In contrast, analytics systems (e.g., Apache Spark [5], Hadoop [6], Dryad [18]) assume worker nodes can store intermediate data (often tens or hundreds of GB) in local memory or storage. Presto [51] pipelines concurrent tasks and buffers data in memory until requested by a consumer. JetScope [8] pipelines tasks but requires workers to write intermediate data to local storage for fault tolerance.

Most NCAs do not have the storage, computational, or memory resources of a conventional worker node, so integrating them into an analytics system requires new approaches to orchestration and data exchange. NCAs must often run concurrently with data producers/consumers and must stream data from/to the source and the destination. Similarly, NCAs using specialized processors are unable to parse arbitrary data formats. As a result, Cheetah, NetAccel [31], DAIET [48], PPS [20] and SwitchML [49] each had to write custom programs to convert input data for their NCAs. Jumpgate takes care of these integration tasks so that NCA programmers just need to write code to add their NCA to Jumpgate's library.

Jumpgate delivers on our early work that described the challenges of using NCAs and motivated the design presented here [38]. The most recent related system is Cheetah [52, 53], which interposes between Spark master and workers to prune data just before it is returned to the analytics system. This approach limits the operations that can run on NCAs to just a final filtering operation, so Cheetah is only evaluated on 9 queries. By contrast, Jumpgate can support any dataflow graph that contains operations for which there are NCA implementations.

3 JUMPGATE DESIGN

3.1 Overview

Jumpgate enables existing systems to execute relational (SQL) queries on NCAs. Jumpgate sits between the analytics

[1] TPC-DS is a popular SQL benchmark for data analytics.

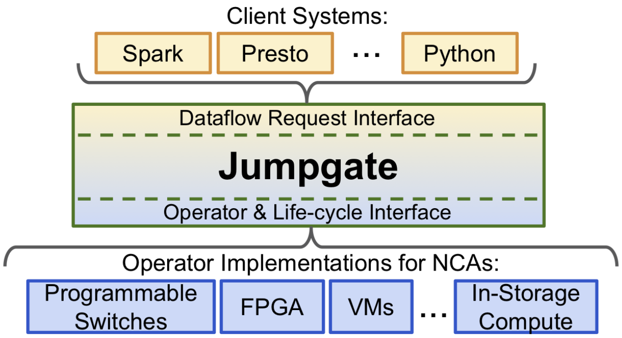


Figure 1: Jumpgate bridges analytics systems and network connected accelerators (NCAs).

system and the set of NCAs that were added to its library (Figure 1). Since Jumpgate positions the analytics system to use the services of NCAs, we will refer to the analytics system as the *client*.

Jumpgate provides two new interfaces for adding NCAs to its library: (1) an *operator interface* that describes the relational operations and data format compatibility of each NCA, and (2) a *life-cycle interface* that describes how to launch, execute and communicate with the NCA (§3.3). To add a new NCA, programmers write a small amount of code to implement these APIs.

To execute client requests, Jumpgate uses a task execution and communication paradigm called *Staged Networked Pipelines* (SNPs)(§3.4). SNPs simplify communication and stage data streaming to enable the execution of analytics tasks in constrained NCAs.

To address the need for simple data formats, Jumpgate introduces *network tuple formats* (NTFs). NTF is a data serialization format where the precise data layout is determined *before execution.* (§3.5).

Jumpgate allows the network transport to vary between two given NCAs or the client and the NCA, and ensures each pair is compatible. Prior systems that used programmable switches for analytics sent a tuple/row per UDP packet, sometimes with an added reliability layer [22, 49, 52, 53, 58]. This approach requires changes to clients to receive UDP packets at high speed using DPDK [46], which can limit adoption [35]. Jumpgate's design allows the client to pick its desired protocol and leaves NCAs designers to implement their own protocol, as prior work has done. So, designing a reliable datagram transport for NCAs is out of scope for this paper. The software-based NCAs in our study can use either TCP for flow control and exactly-once delivery, or UDP when they need to interact with the programmable switch.

Scope and Limitations. This work contributes integration of individual NCAs, but does not address resource allocation and scheduling. We employ simple algorithms in our implementation (described below), but for production deployments, resource schedulers like Kubernetes [27], Mesos [17] or OpenStack [50] would be a better choice.

Jumpgate assumes the client will retry failed queries, and our Spark integration does this. This decision is in-line with Themis [47] and Presto [51] that note failure recovery is expensive with little benefit, even at $\approx$ 1000 nodes, when job times are under a few hours. We designed Jumpgate to execute queries that process a finite amount of data from storage. For Jumpgate to execute streaming queries that process perpetually arriving data, NCAs would need fault recovery mechanisms for operation state [57].

3.2 Jumpgate Step-by-Step

We begin with an example showing how operations of a SQL query are offloaded to NCAs via Jumpgate. The steps, described below, are summarized in Figure 2.

1 A user submits a SQL query to Spark to calculate total sales from each store for a given item, grouped by the store's state. **2** Spark parses the SQL query and computes a *query plan* consisting of relational operations. The plan *reads* from the sales and store tables and *filters* and *projects* each output, then *joins* and *aggregates* the results.

3 Spark generates a **dataflow API** request for Jumpgate using its existing query plan and submits it to Jumpgate. Spark may submit the full or partial query plan (see §3.3.1); in our example, it splits the aggregation operation into a partial aggregation to be done by Jumpgate and a full aggregation to be finished by Spark workers.

4 Jumpgate begins the *compilation phase.* During this phase it maps the dataflow request to a set of NCA implementations that are able to run the requested operations, computes the **network tuple formats** (NTF) that NCAs will use to communicate, and specializes the NCA implementations for the operations and the NTFs. Jumpgate uses the **operator interface** to query the NCAs in its library and find ones that can run the requested operations and communicate with adjacent NCAs or the client (§3.3.2). In this example, Jumpgate chooses **NCA1 – 4** from the available NCAs shown in the figure.

5 Jumpgate coordinates execution by organizing NCAs into a **staged networked pipeline** that ensures that producers and consumers run concurrently and guarantees that stateless NCAs do not need to store intermediate results.

In stage 1, **NCA1** reads the store table and sends data to **NCA3** to build an in-memory hash-table used for the join. In stage 2, **NCA2** reads the sales table and sends data to **NCA3** to probe the hash-table. **NCA3** sends joined tuples to **NCA4** to be aggregated, which forwards partially aggregated results to Spark workers.

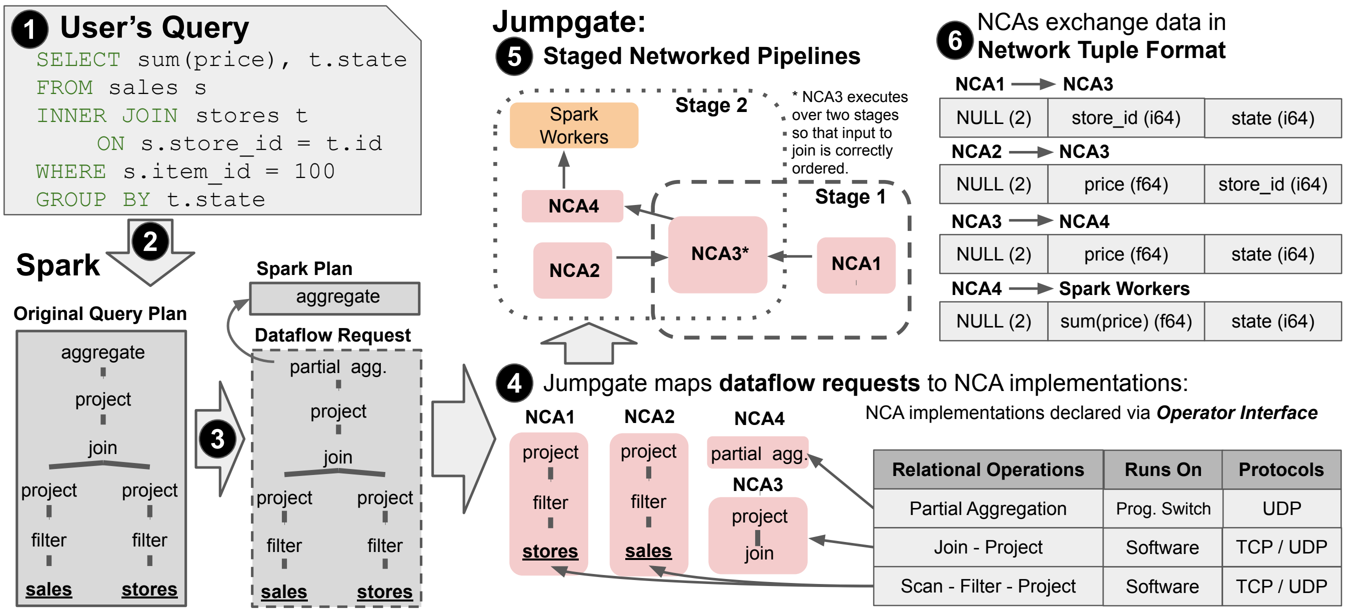


Figure 2: How Jumpgate interacts with client analytics systems and NCAs to deliver processed data to clients.

6 Jumpgate begins execution. Jumpgate initializes each NCA and signals them to execute using the **life-cycle interface**(§3.3.3). Running NCAs exchange data in NTF format shown in the figure.

Summary This overview highlighted the important parts of Jumpgate's design and how they help address our goals: **1. Dataflow API** allows easy interfacing with analytics systems. **2. The operator and life-cycle interfaces** allow Jumpgate to query the capabilities and limitations of various NCAs and to control them. **3. Staged Networked Pipelines** enable correct input ordering to relieve NCAs from storing or buffering intermediate data unless required by the semantics of the operation they implement. **4. Network Tuple Formats** solve the per-packet read limitations of NCAs by simplifying the data format transmitted between NCAs and the client. The next sections describe these parts in detail.

3.3 APIs used in Jumpgate

3.3.1 Dataflow API. Clients use Jumpgate's dataflow API to construct dataflow requests *bottom-up*, starting with reading data from storage. Prior work required analytics systems to know the specific NCAs to use, but the dataflow API means clients do not need to know the set of available NCAs.

The client decides which parts of the query plan to send to Jumpgate based on the operations that Jumpgate supports, shown in Table 1. Most are common analytics operations and the set can be expanded as needed. Jumpgate adds one important operation for sending data to the client: *Send* specifies the address, port, and transport protocol of the client machines that will receive data. Clients can insert a *shuffle* operation to partition results between multiple client nodes.

The client may decide to send the entire query plan to Jumpgate or parts of it. Presently this decision is greedy (every supported operation is sent), but it can be enhanced to account for the costs and benefits of using Jumpgate. For example, §5 describes a simple heuristic we use in our implementation to offset the start-up overhead.

Jumpgate iteratively transforms a dataflow request into a graph of NCA instances by repeatedly picking an operation to replace and checking if any known NCAs can implement the operation. Jumpgate generates candidate graphs and greedily picks one with the smallest number of nodes so operations are fused together. Future implementations could apply prior work on query and dataflow optimization to improve this algorithm as needed [30, 56].

Upon receiving the request. Jumpgate returns a job ID, a description of the chosen NCAs, and the network tuple format the client will receive (§3.3.2). When the client machines are ready to receive data, the client submits the job ID to begin execution, and Jumpgate signals the NCAs to begin working to send data to the client.

3.3.2 Operator Interfaces. Jumpgate has a library of NCA implementations we call *operators*. NCA designers implement the *operator interface* to help Jumpgate find NCAs that can run operations in the client's request (see Table 2). `match_input` and `match_operations` check if the NCA can receive data in a given format and implement the given operations. `match_output` returns the NTF the NCA would emit and the transport that would be used, given the same parameters passed to the first two functions.

Operation	Parameters	Description
read	*path, format, schema*	Reads data from *path* in *format*, returns records in the given *schema*.
filter	*expression*	Filters input records according to *expression*.
project	*expressions, output_schema*	Applies *expressions* to the input data and emits a new record in *output_schema*.
shuffle	*shuffle_key*	Records with the same *shuffle_key* are forwarded to the same destination.
join	*inner, outer, condition, join_type*	Joins records from *inner* to *outer* according to *join_type*.
aggregate	*key, expressions, output_schema*	Groups records by *key*, applies aggregate *expressions* and outputs records as *output_schema*.
send	*host, transport, format*	Send records towards *host* on the given *transport* in the given *format*.

Table 1: Jumpgate's Client API: supported operations and their parameters.

Name	Meaning
match_input	Return true when the NCA accepts the input NTF on a given transport.
match_operations	Given a DAG node, return the node and any subsequent ones if the NCA can implement them.
match_output	Return the NTF/transport the NCA would emit.

Table 2: Operator interface used to query if an NCA can replace a logical operation.

Name	Meaning
compile	Compile a binary or configuration to implement the NCA's assigned operations.
allocate	Start the NCA instance. Returns IP/port of listening NCA.
configure	Configure the destination IP/port(s) of this NCAs output.
execute	Start processing and sending data. Called multiple times for many-stage operations (i.e. join)
destroy	Called on completion/failure to clean up NCA.

Table 3: Life-cycle interface to control NCAs.

The operator interface gives NCA designers freedom to implement any detailed applicability checks their device may need. For instance, some NCAs might support *any* join operation, but others may only be able to run joins with specific characteristics, such as a single 32-bit join key.

3.3.3 Life-cycle Interface. Jumpgate uses the *life-cycle interface* (Table 3) to initialize NCAs, connect them together, and start processing data. This interface is implemented by NCA designers. Jumpgate calls each function once for every NCA that will be used in the query. Some NCAs will use vendor-specific RPCs (e.g., programmable switches), while others can be controlled via SSH. The life-cycle API abstracts these idiosyncrasies.

3.4 Staged Networked Pipelines (SNPs)

SNPs solve the problem of limited NCA storage by organizing NCAs to execute in *stages*. NCAs in the same stage are guaranteed to execute concurrently, and stages are ordered to satisfy data dependencies. Since NCAs in the same stage run concurrently, SNPs ensure that only operations that inherently store data must do so: NCAs with limited or no storage can be used for stateless (or limited state) operators, such as *filter*, *shuffle*, or *partial aggregation*. Only operations that require state, such as *join* or *final aggregation*, need to be implemented on NCAs that have storage.

To compute stages, Jumpgate uses a modified topological sort: runnable operations are added to the current stage, marked, and a new stage is begun. Jumpgate repeats this process until all operations are marked. The key difference of SNPs from scheduling jobs in other dataflow systems is that they cannot rely on nodes to store intermediate results in local storage or memory: the nodes must be set up, connected and launched to ensure that producers and consumers are all available when needed, to cater to NCAs that cannot store intermediate results.

3.5 Network Tuple Formats

NTFs solve the limited parsing capability of NCAs, while supporting multiple NCAs and common storage formats. An NTF encodes the byte layout of the data that a producer NCA will send to its consumers. Jumpgate computes NTFs while mapping the dataflow request to NCAs and checks that the output data of every producer is compatible with the consumers (via the operator interface). To read data from storage formats (e.g., ORC, JSON), Jumpgate uses converters that translate this data to NTF. Jumpgate uses the life-cycle interface to compile/configure converters and NCAs to efficiently read/write a specific NTF layout before executing.

Jumpgate uses the operator interface of each NCA to check that producers and consumers can transmit/receive a given NTF. Some NCAs have a fixed output format, such as the implementation of aggregation on a programmable switch used in our evaluation that always outputs three fields: a sum, count, and grouping key. Such NCAs just return their fixed NTF specification for their output.

Prior work uses fixed data formats, assumes there is only one NCA in use, and relies on hand-coded software to convert input data into the fixed NCA format. For instance, Cheetah [52] has a format specific to a programmable switch that only supports fixed-length values and must include the number of columns in each packet.

The last step in Figure 2 shows the generated NTFs for producer-consumer pairs in our example. Each NTF includes: a bit vector for null values, fixed-length binary fields, and a variable-length section at the end. Strings are handled as offset/lengths that point into the variable-length section. Since an NTF can vary at the level of each producer-consumer pair, we expect the layout of NTFs to co-evolve with device capabilities (e.g., a column-wise format might be preferred between more capable devices).

4 IMPLEMENTATION

Since Jumpgate is not on the data-path we did not have to worry about runtime overhead when choosing the language. We implemented Jumpgate in about 5,500 LoC of Python.

Other Clients – In addition to Spark (detailed in §5.2), there are other Jumpgate clients: a Python client to aid in testing Jumpgate without requiring Spark, and a Presto [51] client that can submit jobs and receive data from Jumpgate, but does not yet offload as many operations as Spark.

NCA Implementations – The NCAs used in our evaluation are shown in Table 4. Most NCAs were written in software, because we did not have hardware NCAs for TPC-DS operations available to us. Our goal was to evaluate *integration* not NCAs, so software implementations are appropriate. One hardware NCA implementation that we could implement was partial aggregation in P4 for the Barefoot Network's Tofino ASIC (PS-Agg). We evaluate it with Jumpgate in §5.5.

JSON [28, 32, 41] and ORC [2, 4, 36, 40, 54] are popular formats used in data analytics[2]. The JSON and ORC software NCAs simultaneously read data from storage (*scan*), convert it to the desired NTF, and optionally *filter* or *project* it. Jumpgate uses simdjson [28] to parse JSON, and the Apache ORC C++ Library to parse ORC [3].

5 EVALUATION

5.1 Experimental Setup

Workloads We use the TPC-DS benchmark [39, 45], which consists of 99 parameterized SQL queries used for decision-making. We use spark-sql-perf [9] to execute TPC-DS queries in Spark; spark-sql-perf breaks up the 99 canonical queries into 104 individual ones. When Spark uses Jumpgate, it can decide to offload multiple sub-trees of a single query, instead of an entire query, producing multiple jobs per query. Spark hits memory limits executing a couple of queries, so we exclude them from the evaluation. We generate TPC-DS data in JSON format at scale factor=100 ($\approx 370GB$ uncompressed) and ORC format at SF=1000 (≈ 387 GB compressed).

[2]JSON is not an efficient format but it is endemic due to it's use as an interchange format between different systems and languages.

		Lines of Code	
Name	**Supported Ops**	**NCA**	**Integration**
JSON	scan-[agg]	505	458
ORC	scan-[agg]	586	501
Join	join-[project]	766	609
Agg	[partial] aggregation	475	344
Shuffle	shuffle	321	203
PS-Agg	partial aggregation	700	186

Table 4: Jumpgate's NCA implementations. Square brackets denote optional operations NCAs support.

Hardware Setup. We run Jumpgate on one machine, which receives jobs, compiles NCA binaries, and then sends them to other machines in the cluster for execution. Our Spark master runs on the same machine as Jumpgate. Spark worker nodes run on other machines. For equal comparison, we run Jumpgate software NCAs with the same resources as baseline Spark. Input data is stored on each machine. We explain our specific setup in the relevant sections.

Metrics. We measure query execution time using spark-sql-perf, which reports the completion time of each query. One way NCAs improve performance is by reducing the amount of data that client systems process. We measure data read and processed by Spark and Jumpgate to show this reduction. We measure lines of code using `cloc` [1].

5.2 Client Integration: Apache Spark

We made Apache Spark (2.4.4) a Jumpgate client in 2,200 LoC – a small amount compared to Spark's SQL modules, which comprise around 100,000 LoC. This suggests that using Jumpgate from client systems will not be onerous. We break down the changes as follows:

Query planning (1,100 LoC): Spark currently offloads scan, filter, projection, broadcast hash-joins, and aggregations. Spark does not offload sort-merge-joins, top-k or operations that reference results of subqueries. *Execution (450 LoC):* Spark coordinates Jumpgate job submission with worker nodes to include the network endpoints of workers in the job request. *Receiving Data (550 LoC):* Spark is built to read files, so we add code to receive data over TCP and UDP sockets from NCAs. To receive as fast as possible, we used Spark's code-generator to generate an NTF parser based on Jumpgate's response to the submitted job.

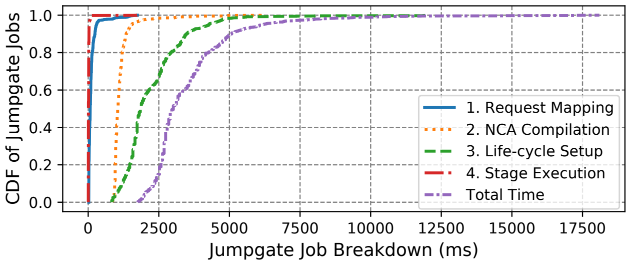


Figure 3: CDF showing the latency of the four execution phases in Jumpgate. *Request Mapping:* map a request to a SNP. *NCA compilation:* specialize NCAs for the assigned operations. *Life-cycle Setup:* upload and start binaries on worker machines. *Stage Execution:* run the query.

Offload heuristic (100 LoC): Jumpgate's prototype has start-up overhead on the order of seconds (mostly due to dynamic compilation – see §5.3), so offloading queries over small datasets is not worthwhile. The amount of data D that is worthwhile to process with Jumpgate is found by solving an equation that estimates the execution time of both systems:

$$Overhead_{JG} + D/Throughput_{JG} \leq D/Throughput_{Spark}.$$

$Throughput$ is determined experimentally by measuring an aggregation over a large table. Both $Throughput$ and D are computed *per-core* so Spark uses Jumpgate when the request would process a dataset at least $D \times numCores$ in size. We set D at $700MB$ to offset a worst-case ≈ 6 seconds of overhead.

5.3 Jumpgate Overhead

To measure Jumpgate's overhead, we set Spark to send all eligible operations from TPC-DS to Jumpgate (disabling the offload heuristic) and to use *minimal* data (each input table contains only a single record). This experiment essentially measures the time to receive "done" messages from NCAs and signal NCAs to switch stages *without* measuring the throughput of NCAs. Here we use a 64-core machine to run Jumpgate, and deploy compiled NCAs to 4 machines.

Figure 3 shows a CDF breakdown of execution time for all 1205 requests in this setup. Spark without Jumpgate takes 11-950ms for the same test. Static overheads are high, but are paid only at the start of a query: 95% of jobs take less than 6s (mean 3.6s). Request Mapping takes 0.09-2.4s, depending on job complexity. NCA Compilation takes 0.88-5s, as each NCA is compiled in parallel. Life-cycle Setup takes 1.5-5.8s because of using SSH to transfer and start binaries on remote machines. However, dynamic overheads are low: **Stage execution takes 13ms - 70ms for all jobs**, depending on the number of stages in each job. Low dynamic overheads mean *Jumpgate can get out of the way during execution, so it is possible to benefit from high performance NCAs.* Static overhead could be reduced, but it is not necessary because the offload heuristic (§5.2) ensures overhead is amortized across long running jobs.

5.4 Performance: Understanding NCA Behaviour on Real Queries

This section explores how NCAs behave when executing queries and illustrates the main factors and bottlenecks that affect performance when Spark uses Jumpgate. We examine a few queries individually to understand performance factors before presenting aggregate performance on TPC-DS.

Experimental Setup. We ran our experiments in Microsoft Azure. One setup uses four 8-core Lsv2 nodes with 3.2Gbps inter-machine bandwidth. To get a faster network with similar core counts, we use the 40GBps loopback network of 32-core Lsv2 instance and restrict each Spark worker and software NCA to use at most 8 cores to simulate the first setup. At this time, no major cloud providers offer a combination of high bandwidth and low core counts, so this is the best we could do to fairly compare Jumpgate to Spark.

Study Limitations. Since we lacked hardware NCAs that could execute all TPC-DS operations (such as data parsing), this section studies the behaviour of NCAs implemented in software using TCP. While performance of software-based NCAs may be disappointing compared to hardware-accelerated equivalents, they are useful to study communication patterns of NCAs executing real queries.

Format conversion is a bottleneck. Figure 4 shows a timeline view of Jumpgate executing TPC-DS Query 3, which has similar performance with and without Jumpgate. To explain why, recall that SNPs form parallel pipelines of NCAs. A pipeline's throughput is limited by the throughput of its slowest stage [16, 24, 29]. So, `orc-2` is the bottleneck because most of its time is spent processing (orange fraction) while other NCAs mostly wait for data (blue fraction). The ORC NCA uses a C++ ORC parser which is almost twice as fast as the Java implementation used by Spark, but the extra work of converting ORC to NTF offsets this advantage.

NCAs reduce client work. The inset bar chart of Figure 4 compares data volumes read by Q3 across different parts of the system. The ORC parser reads 25GB, which turns into 76GB after data is decompressed to memory. This volume of data would normally be processed by Spark. But when Spark uses Jumpgate, NCAs do this processing and Spark only receives 46KB, a $500,000$x reduction compared to the compressed ORC input data, and a $1,600,000$x reduction compared to the decompressed data.

Joins inflate *and* reduce intermediate data. Figure 4 shows that data volume is reduced as it moves from NCAs towards the client. But, Figure 5 shows `join-1` is the bottleneck because many tuples match each record, so while it receives 19GB, it sends 40GB, which is reduced to 378MB

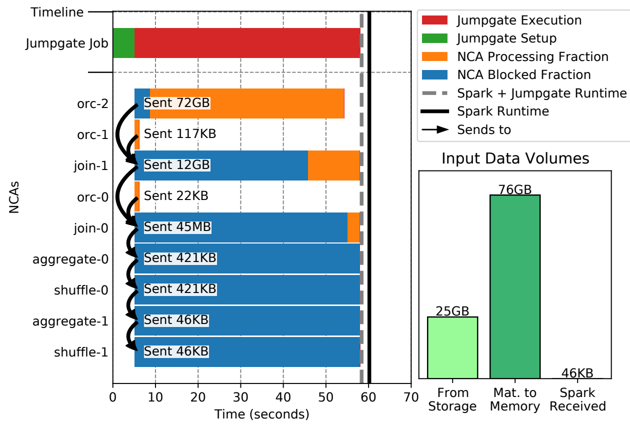


Figure 4: Jumpgate's execution of TPC-DS Query 3. Jumpgate job bars (top) break down Jumpgate's set-up and execution time for the job. Below are the NCAs used to execute the job. Arrows show how NCAs send data to one another: data mostly flows downward. `shuffle-1` is the NCA that transmits data to Spark. For each NCA bar, we show the fraction of time it spent processing data (orange) and the fraction spent waiting to read or send data on the network (blue). These two phases are highly interleaved, but are aggregated for readability. The overlaid text shows how much data each NCA sends. The inset bar chart shows the overall data volume: read from storage, materialized to memory, and transmitted back to Spark.

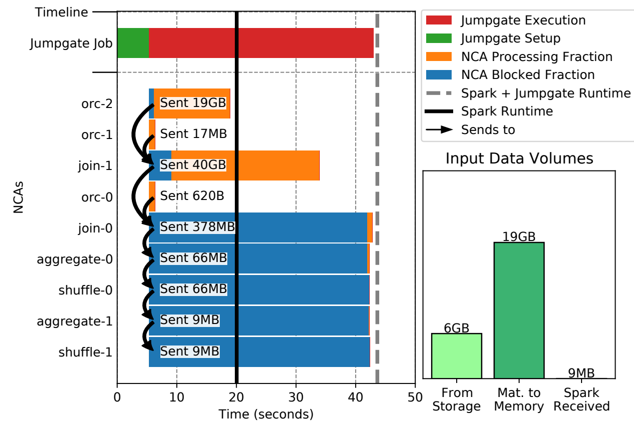


Figure 5: Jumpgate's execution of TPC-DS Query 12. Joins can increase or decrease intermediate data size.

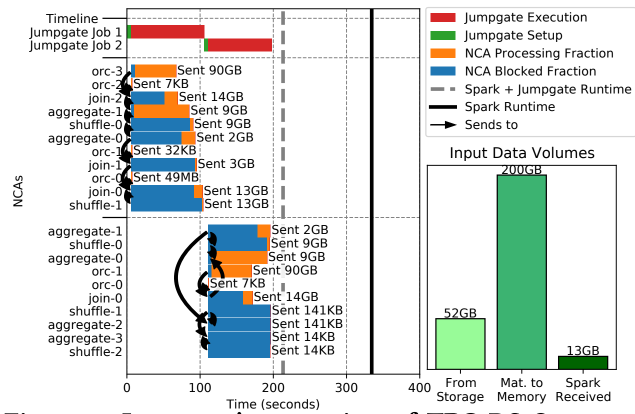


Figure 6: Jumpgate's execution of TPC-DS Query 65. Jumpgate beats Spark because aggregation is the bottleneck and Jumpgate's aggregator is faster.

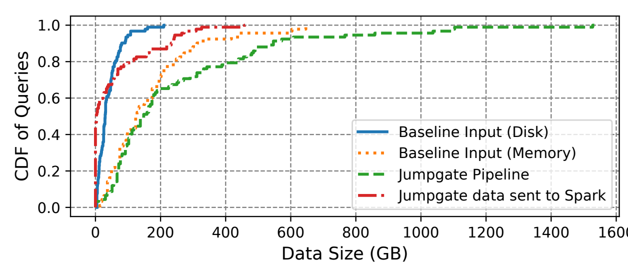


Figure 7: CDF of data size read from ORC on Disk, ORC materialized in memory, sent by Jumpgate to Spark, and sent between NCAs when running TPC-DS queries. Towards top left is better.

by `join-0`. Join ordering and fusing many joins on a single NCA are important optimizations to reduce data volume.

Faster bottleneck operations improve query runtime. Figure 6 shows Q65, where Spark runs two separate jobs on Jumpgate. Performance improves by 1.56×. The aggregation operation is the bottleneck in both Spark and Jumpgate due to lots of unique keys causing many memory allocations. Jumpgate outperforms Spark because the *aggregate* NCA uses a faster allocator (tcmalloc [11]) than Spark.

Overall, these queries illustrate the two factors in understanding when using NCAs will be beneficial:

Factor 1: *The data volume received by the client system.* Work is reduced for the client when the volume of data transmitted to Spark is reduced by "summative" or filtering operations inherent in filters, joins, and aggregations.

Factor 2: *Benefiting from this reduction requires the NCA pipeline to process data faster than the client system.* In Q3 (Figure 4), the software-based ORC parser was not quick enough, so there was only a small speed-up. With a faster ORC to NTF parser, this query would see more speed-up. In Q12 (Figure 5), `join-1` did not produce data quickly enough. But, in Q65 (Figure 6), the aggregate NCA quickly reduced input data volume and improved query runtime. Overall, the first factor shows the *potential improvement*, while the second factor tells us when speed-up can be achieved. Now, we zoom out to look at these factors for *all* of our studied queries.

Factor 1: The potential for data reduction. Figure 7 shows the CDF of data volumes in all TPC-DS/ORC queries. *Baseline Input (Disk)* and *Memory* show how much data Spark read from disk and materialized in memory for all queries, reflecting the first two bars on the inset chart in Figures 4-6.

Using Jumpgate, NCAs instead read *Baseline Input* data, and Spark only needs to process *Jumpgate data sent to Spark*. But, NCAs need to process data in *Jumpgate pipeline* faster than Spark.

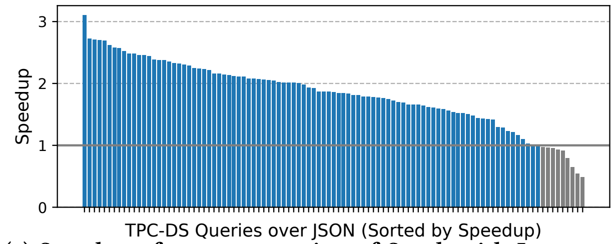
In 60% of all queries Spark receives less data from Jumpgate than it would have read from storage. 50% see a reduction greater than 4×, and 25% see a reduction over 2700×. Comparing data received to that materialized in memory (orange to red dashed lines), **94% of all queries see a reduction in data received by Spark** and 50% of queries see a reduction greater than 22×. *This reduction in data read and materialized by Spark translates into less work for Spark to do.* Lastly, data volume of the NCA pipeline (green dotted line) is in on-par with the data materialized in memory for the same query (orange dotted line), with up to 2× inflation due to joins.

Sending jobs to Jumpgate has potential to be a win when data received and processed by clients can be reduced by orders of magnitude, and these results show that can happen frequently in TPC-DS. These results also validate our decision to offload operations 'bottom-up' from storage, because we are able to capture a significant amount of work from the client system. **The cost of this reduction is that NCAs must now process the input and intermediate data instead of the client system.** The overall volume of data NCAs will process is on-par with what would get written to memory by Spark. *This gives a convenient rule of thumb for understanding the demands on the network when using NCAs.*

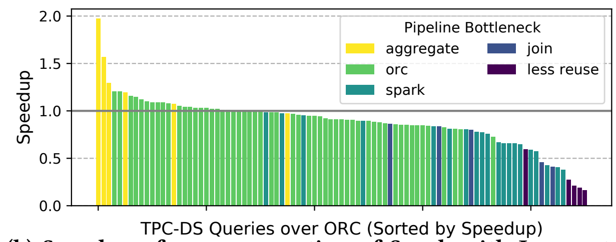
Factor 2: The current performance when Spark uses Jumpgate for TPC-DS. Figure 8a shows the speed-up of Spark using Jumpgate to run TPC-DS queries over JSON, on the 4-node cluster. Jumpgate improves performance by 1.16× −3.1× for 88 of the 104 queries, with the mean improvement of 1.8×. This is because Jumpgate's JSON parser is faster than Spark's due to Jumpgate's use of simdjson [28] over Spark's Java-based parser. Nonetheless, all queries remained bottlenecked on JSON parsing, which is generally quite slow.

Figure 8b shows the speed-up when reading ORC format. ORC is optimized for analytics performance, so it makes both Spark and Jumpgate faster, but Jumpgate becomes bandwidth bound, so we use the single node/32 core setup to increase network bandwidth. We used each query's timeline chart (see Figures 4-6) to determine the bottleneck of each query. Overall, there is less speed-up than with JSON, and many queries are bottlenecked on the ORC parser. The best speedups are on aggregations over many unique keys.

These experiments show that format parsing will be important to accelerate. Such improvements could come from:



(a) Speedup of query execution of Spark with Jumpgate over Spark (baseline of 1), for TPC-DS at SF=100 with JSON input, run on 4 machines with 8 cores each with 32 with 3.2 Gbps networking. Higher is better. All queries were bottlenecked on parsing JSON format data.



(b) Speedup of query execution of Spark with Jumpgate over Spark (baseline of 1), for TPC-DS at SF=1000 with ORC input, on 1 machine with 32 cores with a 40Gbps loopback. Bars are colored based on the pipeline bottleneck we identified for each query using the timeline charts (Figure 4). Higher is better.

Figure 8: Current performance of Software NCAs

changing the format to eliminate parsing overheads, implementing format parsing accelerators in hardware, and further optimizing the format parser.

Figure 8b attributes the worst slowdowns with Jumpgate (far right) to the NTF parser we added to Spark and less inter-operation reuse. The offload heuristic can send only scan and filter operations to Jumpgate and the bottleneck becomes Spark receiving a lot of NTF data. *This underscores the role of Factor 1: offloading operations that ensure data reduction for the client will be key to achieving speedup.* We found this was due to not offloading sort-merge joins from Spark. Such joins preceded an aggregation which would have reduced data received by the client. Another flaw of the offload heuristic was that it removed some opportunities to re-use intermediate results in other parts of the query. Future offload heuristics should offload all joins, avoid removing re-used operations, and generally avoid using Jumpgate if it won't bring a reduction in data volume.

Performance goals for future accelerators. To estimate how quickly future hardware accelerators might need to perform, we can derive how much faster the NCA pipeline should be to meet or beat Spark. To compute this, we scale the overall throughput of the NCA pipeline by the ratio of Spark to Jumpgate performance. This overestimates the performance requirements, because it attributes all slowdown to

NCA processing speed. We find NCA pipelines would have to work at 15.2 Gbps for 90% of queries, and at 30.4 Gbps for all studied queries. Jumpgate's software NCA pipelines currently operate at a mean of 8Gbps, up to 17.6Gbps. 30 Gbps is within the capabilities of a DPDK-based system (§5.5), 40 Gbps SmartNICs, and Barefoot Network's Tofino ASIC (6.4 Tbps). This is a promising result for the feasibility of future NCA development.

5.5 Performance: Programmable Dataplane Switches

We now look at how Jumpgate can accelerate data processing using a *Tofino* programmable switch from Barefoot Networks. We implemented a partial group-by NCA that operates on 64-bit integers in P4 (programmable switches typically do not support floating point arithmetic [49]). It maintains a sum and count per group using a 64K entry hashtable using on-chip SRAM. On a table collision, the packet is forwarded to a *final group-by* software NCA. The entire pipeline set up by Jumpgate consists of (1) a software NCA that parses ORC, converts the data to NTF, (2) the group-by NCA operating on the switch, and (3) the final group-by software NCA. Here, the software NCAs send and receive UDP datagrams to interact with the switch-based NCA. When an NCA is done sending data, it notifies the downstream NCA by sending several zero length datagrams.

Our P4 implementation of the group-by NCA is ≈700 LoC. We integrated it into Jumpgate using only 186 LoC, including the detection of applicable operations and the modification of control plane rules in the switch.

Our experimental setup uses a 6.4Tbps Barefoot Networks Tofino switch [7], and dual NUMA node, 8-core 2.4 GHz Intel Xeon E5-2407 v2 servers, connected to the switch with Intel XL710 40GbE NICs. We observed no packet loss since the NICs and switch were under-utilized.

TPC-DS queries only aggregate floating point values and cannot use this NCA. We wrote our own query that performs a group-by over TPC-DS store_sales table (at SF=100) counting unique items and summing up their sale price as integers. Running this query, the NCA reduces the volume of data by 43×, to 2.29% of the original input, so there is high potential for improvement (Factor 1).

Dataplane programmable switches *always* operate at network line-rate, and so the group-by NCA will run at the speed of the switch hardware: 6.4 Tbps. So, end-to-end performance will be determined by *how fast data can be sent to and received from the NCA (Factor 2).* To illustrate, we ran two experiments:

#1: Fast NCA and slow parsing means low throughput. We had the ORC parser send small UDP packets via the `send` syscall, resulting in low throughput (0.288Gbps). The parser is the bottleneck, because it is slow to send data due to high syscall overhead when sending small datagrams from user-space, so there is no improvement in end-to-end processing.

#2: Programmable switch performance is unlocked with faster format parsing. To improve ORC parser performance, we sent pre-recorded NTF packets to eliminate the parsing bottleneck and use DPDK [46] to bypass the kernel and reduce the sending bottleneck. Sending pre-recorded NTF packets mimics having an accelerator for parsing or using many hosts to send data to the switch. So now, the performance is determined by the final aggregator: With this setup, the ORC parser sends NTF packets at up to 27Gbps, and the final aggregator works at 12Gbps. The group-by NCA on the switch reduces the data volume by 43× and overall completion time is 1.8× faster. Accelerating the final aggregator would improve performance further.

Overall, this test shows that Jumpgate can successfully use programmable switches and highlights the limitations of current hardware, including the need to accelerate input parsing to achieve performance improvements.

6 SUMMARY

When will NCAs be a win? Our study shows NCAs are a win when the accelerators can outperform the client system on the offloaded operations and the network is able to move data quickly between NCAs and the client. We saw that NCAs can improve performance in these conditions and our findings also point towards fruitful designs for future NCAs. We expect even better performance will come with hardware and software NCAs that accelerate format conversion and reduce intermediate data volume by fusing operations.

Jumpgate enables future research. Jumpgate is a necessary step in exploring how NCAs can accelerate analytics tasks. Jumpgate's design allows existing analytics systems to execute queries on NCAs and in turn allows new NCAs to be easily added and evaluated. Before Jumpgate, this would have required the development of format converters, client integration, and potentially hand orchestrating query execution. Researchers must still develop NCA implementations, but Jumpgate relieves researchers from wrestling with integration tasks so they can start asking deeper questions about using NCAs.

REFERENCES

[1] AL Danial. [n.d.]. cloc Github repository. https://github.com/AlDanial/cloc.
[2] Apache Software Foundation. [n.d.]. Apache Arrow. http://arrow.apache.org/.
[3] Apache Software Foundation. [n.d.]. Apache ORC Core C++. https://orc.apache.org/docs/core-cpp.html.
[4] Apache Software Foundation. [n.d.]. Apache Parquet. http://parquet.apache.org/.

[5] Apache Software Foundation. [n.d.]. Apache Spark. http://spark.apache.org/.

[6] Apache Software Foundation. [n.d.]. Hadoop. http://hadoop.apache.org/.

[7] Barefoot Networks. [n.d.]. Barefoot Tofino Switches. https://www.barefootnetworks.com/products/brief-tofino-2/.

[8] Eric Boutin, Paul Brett, Xiaoyu Chen, Jaliya Ekanayake, Tao Guan, Anna Korsun, Zhicheng Yin, Nan Zhang, and Jingren Zhou. 2015. JetScope: reliable and interactive analytics at cloud scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1680–1691.

[9] Databricks. 2018. Spark SQL Performance Tests. https://github.com/databricks/spark-sql-perf.

[10] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[11] Sanjay Ghemawat and Paul Menage. 2009. Tcmalloc: Thread-caching malloc.

[12] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2.

[13] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. 2016. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *Proceedings of the First Workshop on Optimization of Communication in HPC* (Salt Lake City, Utah) *(COM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 1–10. https://doi.org/10.1109/COM-HPC.2016.6

[14] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 153–165. https://doi.org/10.1109/ISCA.2016.23

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. ACM, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[16] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[19] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1202–1213. https://doi.org/10.14778/3137628.3137632

[20] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '19)*. ACM, New York, NY, USA, 21–28. https://doi.org/10.1145/3314148.3314356

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

[23] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-performance Database System Leveraging In-storage Computing. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 924–935. https://doi.org/10.14778/2994509.2994512

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 783–798.

[25] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[26] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. ACM, New York, NY, USA, 219–231. https://doi.org/10.1145/3123939.3124553

[27] Kubernetes. [n.d.]. Production-Grade Container Orchestration. https://kubernetes.io/.

[28] Geoff Langdale and Daniel Lemire. 2019. Parsing Gigabytes of JSON per Second. *CoRR* abs/1902.08318 (2019). arXiv:1902.08318 http://arxiv.org/abs/1902.08318

[29] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *ACM Trans. Parallel Comput.* 2, 3, Article 17 (Sept. 2015), 42 pages. https://doi.org/10.1145/2809808

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[31] Alberto Lerner, Rana Hussein, and Philippe Cudre-Mauroux. 2019. The Case for Network-Accelerated Query Processing *(CIDR 2019)*.

[32] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[33] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications Onto smartNICs Using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. ACM, New York, NY, USA, 318–333. https://doi.org/10.1145/3341302.3342079

[34] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[35] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[36] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[37] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)* (Farminton, Pennsylvania). ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[38] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. 2019. Jumpgate: In-Network Processing as a Service for Data Analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mustard

[39] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 1049–1058.

[40] Apache ORC. [n.d.]. ORC Specification v1. https://orc.apache.org/specification/ORCv1/.

[41] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proceedings of the VLDB Endowment* 11, 11 (2018).

[42] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-end Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (May 2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[43] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[44] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 663–679. https://www.usenix.org/conference/osdi18/presentation/phothilimthana

[45] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, Taking Decision Support Benchmarking to the Next Level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. ACM, New York, NY, USA, 582–587. https://doi.org/10.1145/564691.564759

[46] DPDK Project. [n.d.]. Data Plane Development Kit (DPDK). https://www.dpdk.org/.

[47] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. 2012. Themis: An I/O-Efficient MapReduce. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 13, 14 pages. https://doi.org/10.1145/2391229.2391242

[48] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (Palo Alto, CA, USA) *(HotNets-XVI)*. ACM, New York, NY, USA, 150–156. https://doi.org/10.1145/3152434.3152461

[49] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR* abs/1903.06701 (2019). arXiv:1903.06701 http://arxiv.org/abs/1903.06701

[50] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012), 38–42.

[51] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[52] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2019. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (Beijing, China) *(SIGCOMM Posters and Demos '19)*. ACM, New York, NY, USA, 72–74. https://doi.org/10.1145/3342280.3342311

[53] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. *SIGMOD* (2020). https://doi.org/10.1145/3342280.3342311

[54] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis: High-Performance File Format for Big Data Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 615–630. https://www.usenix.org/conference/atc18/presentation/trivedi

[55] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. 2016. Bluecache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment* 10, 4 (2016), 301–312.

[56] Youngseok Yang, Jeongyoon Eo, Geon-Woo Kim, Joo Yeon Kim, Sanha Lee, Jangho Seo, Won Wook Song, and Byung-Gon Chun. 2019. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 177–190. https://www.usenix.org/conference/atc19/presentation/yang-youngseok

[57] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[58] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with in-Network Conflict Detection. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 376–389. https://doi.org/10.14778/3368289.3368301