Compilation-assisted Performance Acceleration for Data Analytics

by

Craig Mustard

B.Sc, Simon Fraser University, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

September 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Compilation-assisted Performance Acceleration for Data Analytics

submitted by **Craig Mustard** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

Examining Committee:

Alexandra Fedorova, Department of Electrical and Computer Engineering
Supervisor

Ivan Beschastnikh, Department of Computer Science
Supervisory Committee Member

Andrew Warfield, Department of Computer Science
University Examiner

Karthik Pattabiraman, Department of Electrical and Computer Engineering
University Examiner

Additional Supervisory Committee Members:

Mieszko Lis, Department of Electrical and Computer Engineering
Supervisory Committee Member

Abstract

Fundamental data analytics tasks are often simple – many useful and actionable insights can be garnered by simply filtering, grouping, and summarizing data. However the sheer volume of data to be analyzed, demands of a multi-user operating environment, and limitations of general purpose processors make it challenging to perform these operations efficiently at scale. This thesis presents two techniques that address these challenges to improve the response time of data analytics tasks: (1) Newly emerging programmable network processors can perform data analytics tasks at terabits per second. However, existing data analytics systems, like Apache Spark, cannot readily use network processors because network processors are very limited and cannot execute the tasks generated by existing analytics systems. Using network processors for analytics requires re-designing how existing systems compile and execute data processing tasks. This thesis introduces Jumpgate, a system that enables existing data processing systems to execute relational queries using network processors. Jumpgate compiles client requests to a novel execution model that coordinates execution on heterogeneous network processors. Jumpgate can already improve performance by 1.12-3$\times$ on industry standard benchmarks, and paves the way for adopting network processors for data analytics tasks. (2) Analytics systems often process similar queries, either submitted by the same or different users. Cross program memoization (CPM) is a technique to re-use results of prior computations across programs and users. However, CPM is often not enabled because prior implementations have high overhead and are unable to re-use the output of user-defined functions (UDFs). This thesis presents KeyChain, a CPM implementation that identifies equivalent UDFs and has low overhead so CPM can be always be enabled.

Lay Summary

This thesis presents two systems that improve the performance of data analytics, which translates into faster answers and cost savings for users. (1) New processing accelerators are being developed that surpass the speed of general purpose processors. However, these accelerators achieve greater processing speed by being fundamentally limited. Existing data analytics systems could benefit from these accelerators, but were built with commodity processors in mind, and cannot adapt to the limitations of these accelerators. This thesis redesigns how analytics tasks are run, and proposes Jumpgate, a system that can run analytics tasks on emerging accelerators and makes it easy for existing analytics systems to use them. (2) KeyChain enables analytics systems to re-use data from prior computations and improve response time for users. While re-using results is not new, KeyChain enables more data re-use by quickly identifying data and is able to find more equivalent results.

Preface

Chapter 2 has been published in HotCloud 2019 (C. Mustard, F. Ruffy, A. Gakhokidze, I. Beschastnikh, and A. Fedorova. Jumpgate: In-network processing as a service for data analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, 2019. USENIX Association). Chapter 3 is under submission. In these projects, I was the lead investigator, responsible for initiating and leading the research direction, as well the system design, implementation, experiments, and the bulk of the writing of the submitted work. However, Jumpgate was a group effort, and other authors contributed writing and implementations used by the system. Alexandra Fedorova and Ivan Beschastnikh were supervisory authors and contributed valuable guidance, editing, and writing. Fabian Ruffy contributed writing to Chapter 2 and to the implementation of Jumpgate's networking. Anny Gakhokidze contributed the performance estimations in Chapter 2. Swati Goswami contributed the implementation of the P4 aggregation as well as writing to Chapter 3. Niloo Gharavi and Joel Ahn contributed to the implementation of Jumpgate's JSON and ORC parsers, respectively.

A version of Chapter 4 (KeyChain) has been published in IEEE BigData 2019 and received an award for Best Paper (C. Mustard and A. Fedorova. Practical cross program memoization with keychain. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 262–271, 2018). Unless otherwise specified, all writing, experiments, and code discussed was written by me.

Table of Contents

List of Tables

List of Figures

Acknowledgments

It takes a village to do research. I was lucky enough to have a wide breadth of support in both my academic and personal life while working towards my research goals. I am humbled to have the chance to chart my own research path and to have the support of so many people along the way.

Thanks to my supervisor, Sasha, for your advice, patience, and support over the many years of working together. It was a blessing to have a supervisor who was willing to explore new areas and support me during the learning process. Thanks to Ivan for your insightful viewpoint and especially your encouragement during the early stages of the Jumpgate project. Thanks to Mieszko for comments on this thesis and the reassurance that choosing the right problem to work on can be one of the toughest challenges in research.

Work only gets better in collaboration with others. Thanks to all my collaborators and co-authors. Fabian, Swati, and Joel N., thank you for your critical and skeptical eyes on my crazy ideas. Svetozar and Snehasish, my former lab-mates from SFU, thanks for all the late night discussions and encouragement! I am proud that I have gotten to play a small part in helping other students explore research: Thanks to Niloo, Anny, Joel A., Osama, Andi, and Henry for all your help.

I am indebted to the context that I found myself in for the last few years of my PhD. The NSS / Systopia lab provided a welcoming and productive environment to discuss research and meet other students. The many reading groups, impromptu discussions, and more were critical in helping me understand the greater context of research and helping me to chart a path in my own research.

Thanks to all my friends and family, who know how long of a road this has been and have been with me along the way. Special thanks to my partner, Amy, who has experienced many of the ups and downs of PhD life alongside me. Thanks to my friend, Alison, who encouraged my journey into research since the beginning. Thanks to my Dad and Kathy, who upon hearing I decided to work on a PhD, both laughed and said ‘I thought you might’.

Chapter 1

Introduction

Large scale data analysis plays many important roles in society. Data analytics helps humans to understand the course of diseases, social and economic behaviours, climate change and environmental issues, and much more. However, large scale data analytics are expensive, in terms of both operational costs and people-hours spent waiting for results or acting on stale results. So, it is important to enable cheaper and more timely analysis. Both costs can be reduced by improving the performance of data analytics systems: faster systems reduce waiting time and the system costs less to operate per-query.

I chose to work on this problem because data analytics is impactful and improving performance requires solving interesting and novel computer systems problems. This thesis presents two approaches to improving data analytics performance: *One looks towards the future of analytics, and one re-uses results of past queries.* Both leverage compilers and compilation techniques.

The Future – The future of many computer systems is specialized hardware. General purpose processors have fundamental performance limitations due to their generality. So, system designers are turning to domain-specific architectures to continue scaling performance [45, 72]. Instead of aiming to support all processing tasks, domain-specific architectures are specialized for better performance on *certain tasks*. Data center operators can help users improve performance over general purpose processors for *many* use-cases by making several different heterogeneous architectures available for use.

Some promising domain-specific architectures are what we call *network connected accelerators* (NCAs), which include *in-network processors*. NCAs are attractive because they can communicate with other networked systems, can be scaled independently, and can be allocated on-demand so that many systems can share a few accelerators. Prior work showed that in-network processors can accelerate analytics tasks by 2-10$\times$. However, these accelerators are not yet integrated into data analytics systems because NCAs can be very limited and existing analytics systems are designed to target general purpose processors. *To allow NCAs to work together and with client systems, we need a new execution paradigm that targets them.* At the same time, because no data analytics systems currently support NCAs, there is little established ground-work to allow for investigation into the myriad opportunities and challenges of NCAs. Chapter 2 explores the promise and challenges of using NCAs for data analytics in detail and proposes a vision of providing NCAs as a service to analytics clients.

Chapter 3 presents *Jumpgate*, a system that enables existing systems to use NCAs to execute data analytics tasks. Jumpgate decouples the client analytics system from the specific NCAs in use, so that client systems are not burdened to integrate every new NCA. Jumpgate makes several fundamental contributions to using NCAs that provide a foundation for future work. First, I show that existing systems can easily use NCAs by exporting a program representation that they *already use*. Second, I design a novel task execution and communication paradigm that enables NCAs to *work together* within the constraints of the most restricted NCAs we are currently aware of. Finally, I design two hardware agnostic interfaces that enables existing NCAs to be easily added to Jumpgate in a few hundred lines of code.

Jumpgate's fundamental contributions open up opportunities for further research, and the Jumpgate prototype is an artifact that can be built on by others. Since Jumpgate enables existing systems to use NCAs and NCAs to work together, new questions can be addressed, such as: the best NCA designs for analytics or the best way to allocate and schedule NCAs with respect to network topology. Before Jumpgate, there would have been no reason or ability to investigate such questions because the groundwork for how analytics systems would use NCAs was not established. We explore potential future work in §3.7.

The Past – Many analytics systems can re-use intermediate and final results from prior queries to answer new queries quickly. Such systems maintain a cache of results from previously executed queries. Existing systems like Apache Spark have a separate cache for each user session. But, there are also opportunities to re-use results *between users and programs.* However, finding out whether or not a *particular* past result exists can be an expensive operation, especially if the analytics system needs to analyze any user-provided code across multiple users. This cache lookup overhead can be successfully amortized when data re-use is guaranteed, but the potential for data re-use varies across deployments and user-bases, making it difficult to predict in advance. Since data re-use between queries cannot be predicted before queries are submitted, data analytics systems need *low overhead* techniques to identify prior results that can work across query, program, and user boundaries to extract the most benefit.

Chapter 4 presents KeyChain, a low-overhead technique to compute lookup keys from data analytics programs that can be used to index into a cache. KeyChain's fundamental contributions are providing a general technique that can be used in many analytics systems to identify data to be re-used. KeyChain examines how existing compilers can be used to help identify equivalent, but syntactically programs, and contributes a benchmark to examine how well this can work in practice.

Compilation-assisted Techniques – A unifying theme of both Jumpgate and KeyChain is the application of compilers and compilation techniques to solve systems challenges. My work views compilation as a broad technique that progressively takes a domain-specific program, applies domain-specific optimizations or transformations, and re-maps the program to a lower level domain, where further optimization and mapping is performed until machine code can be emitted. Such compilation techniques can be a powerful tool to solve many integration and performance challenges encountered when building systems. Chapter 5 elaborates on this viewpoint.

Jumpgate uses compilation techniques to map client requests to implementations of analytics operators written for NCAs. Jumpgate applies code generation to specialize these implementations for specific queries. The generated code relies on the strong optimization capabilities of native compilers such as GCC [2] and LLVM [106]. Jumpgate's analytics operations can outperform JVM-based systems by more than a factor of 2× thanks to this compilation approach.

KeyChain relies on the semantics of analytics programs to identify memoization opportunities and re-write programs to exploit them. KeyChain shows how it is possible to exploit the internal canonicalization passes [61, 119] of optimizing compilers to quickly detect equivalent but syntactically different programs in under 350ms. When integrated into Apache Spark, KeyChain can look up data in the cache with negligible overheads on-par with the inherent variaton in execution time of Apache Spark. When cached data is found, KeyChain can speed up execution by 2× - 281×, depending on how data is read from the cache.

The remainder of this thesis proceeds as follows. Chapter 2 describes the challenges and opportunities of network connected accelerators. Chapter 3 describes the design and evaluation of Jumpgate. Chapter 4 describes the design and evaluation of KeyChain. Chapter 5 situates both works in the compilation space and presents overall lessons I learned from designing compilation-assisted techniques.

Chapter 2

Promises and Challenges of Network Connected Acceleration

In-network processing, where data is processed by special-purpose devices as it passes over the network, is showing great promise at improving application performance, in particular for data analytics tasks. However, analytics and in-network processing are not yet integrated and widely deployed. This chapter presents a vision for providing in-network processing as a service to data analytics frameworks, and outlines benefits, remaining challenges, and the early research directions towards realizing this vision.

This chapter was accepted for publication at the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19) [135].

2.1 Introduction

In-network compute capability is growing. Figure 2.1 illustrates the network-attached compute resources that are (or are becoming) available in modern data centers. New programmable hardware includes packet processors that can efficiently process packets using specialized hardware designs, such as SmartNICs [59, 173], programmable switches [37], and NPUs [39, 139, 184]. In parallel, software data paths are becoming increasingly programmable via tools like the Data Plane Development Kit (DPDK) [50] or the eXpress Data Path (XDP) [74], while

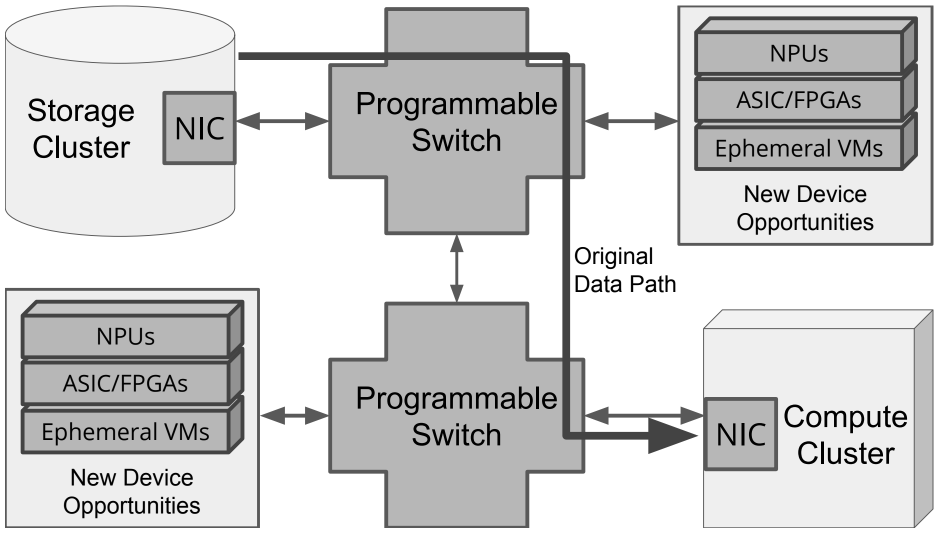


Figure 2.1: Opportunities for in-network processing in the data center. Darker boxes show new programmable opportunities for processing. The solid arrow shows the path that data takes from storage to compute nodes.

network function virtualization platforms are being optimized for flexible high-performance packet processing [145, 149, 194].

Programmable network processors can improve application performance. Prior work has shown the benefits of taking advantage of compute that is conveniently located along the data-path [66, 85, 86, 111, 113, 114, 123, 124, 166]. However, existing systems that have used in-network processing have been application specific [111, 113, 114, 124] or have only been feasible in the research lab. Often, they require invasive changes to infrastructure or the use of unreliable network protocols [66, 111, 124]. *How can we deploy in-network processing in real settings?*

Our vision: We propose In-Network Processing as a Service (NPaaS)[1]. End-users continue to write high level programs (e.g., SQL) and use existing data processing frameworks such as Apache Spark [27], but the framework would use NPaaS to offload operations to processing elements along (or near) the data path. A common scenario might be NPaaS initiating the read from storage, pre-processing the data,

[1]We pronounce it similar to ‘impasse’.

and directing the processed data to compute nodes of the framework. NPaaS could be operated by cloud providers or as a user-run service that requests lower level resources from the provider. Realizing this vision will require designing solutions to challenges presented by this new type of computing, including changes to the existing system stack (§2.2). To drive research into NPaaS, we are developing a prototype called Jumpgate (§2.3).

2.1.1 Motivation

Why should data analytics adopt in-network processing?

Data analytics need to leverage new high performance hardware. Data analytics is an expensive but common task. Substantial effort has been spent to accelerate query workflows with high performance devices. We view network processors as another heterogeneous processing device to be adopted in order to improve performance. Just as data analytics systems have integrated GPUs [162], TPUs [9], DPUs [12], and FPGAs [94], they should also coordinate with in-network processors. Since many analytics tasks can be CPU-bound [143], offloading operations to network devices helps to alleviate valuable CPU resources [59] while decreasing workload execution time for the end user. Prior work and our estimates show that in-network processing can accelerate data analytics tasks by orders of magnitude, detailed in §2.1.2.

We believe that network processing devices, special-purpose and resource constrained as they may be, can act as specialized compute stages as part of data analytics pipelines. For example, Reconfigurable Match Table (RMT) switches [37], programmed in P4 [38], present a very challenging and restrictive programming environment. In spite of this, prior work has used programmable switches to perform join-and-group-by operations [111], aggregations [166], build replicated key-value stores [85, 86], optimize consensus and transaction protocols [113, 114], offload network telemetry queries [69, 138], and implement several network management algorithms [171, 172]. Work like Domino [172] and FlowBlaze [154] provides even higher-level languages for these devices.

We can't move all compute to storage. Existing data processing systems try to place compute near storage nodes to reduce data movement costs [191]. But, modern cloud architectures are decoupling compute and storage to achieve better resource utilization and fault tolerance. Instead of general purpose machines with attached storage, we now have special-purpose data serving machines [70, 96, 137] with very little compute capability. In these deployments, the classic 'move compute to data' optimization is no longer possible.

For instance, Facebook's Bryce Canyon storage server combines 72 high capacity hard drives with two Xeon D-1500 SoCs [54, 55], each with at most 16 cores – *less than half a core per drive.* Since we can no longer place a lot of compute on storage nodes, *we should find ways to place computation near data, and our next option is the network path.*

Data analytics need flexible deployments. Data processing systems behave inefficiently if their nodes are statically allocated in advance and they assume no other compute resources are available. For instance, to filter a large volume of data quickly, we require many nodes. After applying a highly selective filter, the data will be 10-1000x smaller but distributed across the same nodes used to filter it. However, it is often faster to process small datasets on a single machine because it avoids distribution overheads [127]. NPaaS could be used to push the filter operation into the network and ensure all relevant records arrive at a single host.

Ephemeral (serverless) compute service (e.g., AWS Lambda [1]) can address this need for flexibility [89], but *are currently hamstrung by the lack of direct access to the network* [70] and instead rely on inefficient cloud file systems for state exchange [97]. *If they had direct network access, we could treat ephemeral compute services as in-network processors.*

Why provide network processing as a service? As we've seen, in-network processing has a growing number of hardware implementations, but as far as we know, *there are no proposals to manage and expose this plethora of new hardware to end-users*. However, the burden of managing all this hardware should *not* fall to each data analytics framework, as Lerner et al. propose [111]. Instead, analytics systems and in-network processing should be decoupled with an interface that allows the framework to specify desired operations, while NPaaS manages instantiating requested operations on appropriate hardware. NPaaS allows hardware to be managed independently from the data analytics systems while saving analytics developers from implementing device-specific code. We describe potential general interfaces in §2.2, and our NPaaS prototype in §2.3.

We want to inspire more network-aware hardware designs. We believe network processors are good hardware accelerators for data analytics. Recent work on near-storage [49, 67, 82, 88, 99] or in-memory processors [12, 31, 65, 187] tightly couples storage with specialized processors, but as we've discussed, storage and compute are better utilized if they are decoupled. We think more data processing devices could be designed assuming data arrives over the network as coordinated by NPaaS, and hope that our work inspires new network-aware data analytics accelerators.

2.1.2 Estimating Performance Potential

In-network processing can reduce the overall work required of end-hosts by performing computation in network, reducing data volume and speeding up applications. To demonstrate, we run an experiment and draw on prior work to show that in-network processing will benefit data analytics.

Experiments. We estimate the effect of offloading common analytics operations to the network by measuring reduction in time and bytes transmitted on a query[2] run using Apache Spark. We simulate offloading operators using Spark SQL to query pre-processed files that contain data that would come from the offloaded operators. We offload operations in the query that require no storage (filter, projection, shuffle) or can be bounded (partial aggregation). To measure traffic reduction, we pack each record in a UDP packet and drop or modify packets as dictated by the operation, measuring data sent over the wire before and after. This assumes operators work as fast as our cluster[3] so our measurements give an upper-bound on the benefits.

Table 2.1 summarizes our results, averaged over 5 runs. **Filter and Project** reduce traffic commensurate with the amount of rows or columns they remove. For our query test, *shuffle and partial aggregation also apply filter and project*. **Shuffle** eliminates the need for nodes to exchange records by sending records to the right node in the first place with a network operator. Based on traffic reduction, shuffle

[2] `SELECT item_sk, sum(quantity) FROM store_sales GROUP BY item_sk WHERE item_sk < I`, where `I` selects 50% of records

[3] A 4-node Spark cluster using Azure E4v3 VMs, with 4 cores each.

Operator Offloaded	**Traffic Reduc.**	**13.7GB**	**68.9 GB**
		Spark Query Time	
None	0%	56s $\pm$0.8s	256s $\pm$5s
Filter	50%	37s (-35%)	124s (-52%)
Project	85%	20s (-64%)	38s (-86%)
Shuffle	40%	14s (-76%)	31s (-88%)
Partial-Agg	90%	14s (-76%)	17s (-94%)

Table 2.1: Effect of in-network offload on traffic and Spark query time for two data set sizes (13.7GB, 68.9GB). Input data is TPC-DS `store_sales` as JSON.

should improve query time more, but Spark's API lacked a way to declare that data is pre-shuffled and our hand-written aggregation is slower than Spark's native SQL operators [46]. If Spark's optimizer had such an API, we could use the native operation. **Partial Aggregation**, inspired by [69, 138], aggregates within a storage limit (8MB) by evicting partially aggregated records and sending it to the end host. Overall, as we offload most operations to the network, **query time drops by 94% of the baseline (a 16x speedup).** Given partial-aggregation runs at just 4 GB/s (on 4 nodes), assuming data can be processed as fast as Spark reads seems reasonable because 40Gbps RMT switches operate at 5 GB/s and the latest software JSON parsers can operate at 2 GB/s *per core* [104, 115].

Prior work has already shown benefits of performing in-network operations. For in-network aggregation: Using RMT switches, Sonata [69] reduced network telemetry traffic to the end host by 3-7 orders, DAIET [166] reduces aggregation traffic by 86-89% (6-8x). Using middleboxes, NetAgg [124] increased search result aggregation throughput by 9.3x, and reduced total Hadoop execution time by 4.5x. TAG [123] aggregates sensor data and decreased traffic by 8x. Mellanox's SHArP [66, 128], a commercial in-network aggregator, shows a 2x performance improvement on an MPI programs [66].

More complex operations and compilation strategies are being explored: Netaccel [111] runs join-and-group-by on an RMT switch and accelerates a TPC-H query fragment by 2x. The Marple [138] compiler translates network telemetry queries to P4 code, and Sonata [69] partitions these queries between end-host ma-

	Byte stream	**Record / Datagram**
Example	TCP	UDP, DCCP, SCTP
Requirements	Flow tracking.	Record-packet alignment.
Pro	Fewer changes to existing software.	Efficient per-packet processing.
Con	Must modify flow.	Must packetize records.

Table 2.2: Comparison of transport layers for in-network processing, showing known protocols, requirements, pros, cons.

chines and programmable switches. Floem [151] enables experiment with various NIC offloading strategies, and improves data analytics throughput up to 96%.

2.2 NPaaS Design Challenges

#1: NPaaS requires extensive co-design.

Given an analytics task, data source and destination endpoint, NPaaS must orchestrate the processing of data passing over the network by selecting and configuring available devices. This is challenging because the devices that can be used depend on the data format and transport protocols used to send data, which in turn depend on where data is stored. For example, Figure 2.1 shows on-path and off-path devices. On-path devices operate at line-rate, but can only read a small number of bytes of each packet, and have small amounts of state [38, 41, 111, 166]. Off-path devices are more flexible, but slower, and introduce latency. *Achieving even basic network processing requires co-designing storage systems, network transport, and data formats with the capabilities of the available network processors.*

Network Transports. Table 2.2 covers network transports NPaaS could use to send data. Our main constraint is many network devices (see §2.1.1) only operate on single packets, and can't buffer data across packets in a network flow. To operate per-packet, a complete record must be in a single packet. In other words, each record must be *packetized*.

Stream protocols like TCP require fewer modifications to existing systems.

Flat Columnar	**Nested Columnar**	**Unstructured**
Examples:		
ORC, Arrow, Albis [19, 20, 178]	Parquet, Dremel [26, 129]	CSV, JSON
Storage Pro/Cons:		
Low storage overhead, but must be converted.		No conversion, high overhead.
Packetizing complexity:		
Fixed/length-prefixed records	Reassembly [90]	Newline search

Table 2.3: Data formats categorized by their storage-side benefits, and challenges for storage-side record alignment.

But, operating on a flow requires buffering data, observing all packets, and modifying TCP state; tasks that are outside the capabilities of packet processors. Even if records are chunked into individual TCP packets, they cannot be reordered or dropped without tracking the behavior of the TCP state machine *per flow*. In practice, modifying TCP streams requires a proxy, stream processor or middlebox.

Prior work has thus sent records over UDP-based protocols (e.g., Netaccel [111]). In practice, we need a reliable protocol to ensure data has been processed. Fortunately, reliable datagram protocols, such as SCTP [174] and DCCP[4] [98], are available in the Linux kernel today. However, the remaining downside to datagram protocols is that *the sender must packetize records.*

Data Formats. Table 2.3 covers data formats NPaaS might support. Our main concern is the ability to process popular data formats while guaranteeing the packet content is parseable by the network devices we wish to support. For instance, RMT switches can only process 200-500 bytes of fixed-length data from each packet [37, 166, 171].

Unstructured formats, like JSON, are prevalent but are difficult to parse quickly [104, 115, 147], although there are promising hardware accelerators [56]; binary formats can be parsed more quickly. We argue NPaaS should support both unstructured data (JSON) for general applicability and binary formats for best performance and compatibility with hardware. Not all in-network processors need to support all

[4]Assuming a reliable layer on top of the congestion control protocol.

formats, but NPaaS should allow for processing of different formats.

Data formats vary in how complex it is to find record boundaries to packetize data. JSON or CSV records are easily found by searching for newlines [28, 78] and flat binary formats just need offset calculations, but binary formats for nested schemata require complex algorithms [90].

Storage System Requirements. As said earlier, to send data via record transport, a storage system needs to packetize records. We must also be careful with distributed file systems that split data files into chunks, like HDFS [23] or Ceph [183]. Since chunking is done without awareness of records, records can span multiple chunks, and it is possible to see incomplete records at the flow level. If we want to use a record transport, one solution is to have a middlebox read all chunks that make up a file, transcoding data into a record transport on the fly.

Open Research Questions. There are many more research questions along the road to co-designing NPaaS. Here is a short selection. ***Q1**: How should we allocate and schedule processing with respect to the network topology? How long will processing pipelines need to last?* ***Q2**: Can middleboxes perform fast enough to make a difference to applications?* Even though hardware packet processors are faster, it is worthwhile to implement operators in software, even to just provide a performance baseline. ***Q3**: Are existing transport protocol sufficient for NPaaS or do we need custom protocols?* Customized protocols allow for domain-specific optimizations but require significant development effort.

#2: Integration with Analytics Frameworks.

Multiple data analytics systems need to communicate their operations to NPaaS. Is there a common format? What operations can be supported? Conveniently, most data processing systems model programs as dataflow between nodes in an abstract graph [9, 13, 64, 81, 133, 146, 148, 191] and have equivalent operators: many support SQL dialects and functional style operations (e.g., filter, project, map, reduce, group-by, shuffle) that can be directly mapped between frameworks [64].

When an analytics framework requests computation, the task for NPaaS is to map the desired operations to implementations on network processors. A simple way is to use pre-written or templated implementations for each device. To support framework-specific operators, or avoid hand-coding a multitude of operators, NPaaS could draw on cross-framework intermediate representations that enable compilation to different backend devices (e.g., Weld [146, 148], Dandelion [163]).

#3: Multi-tenancy and Isolation.

How can we run user code on hardware that lacks isolation? If NPaaS only uses client-allocated resources, such as VMs and containers, isolation and multi-tenancy is, arguably[5] addressed by existing isolation mechanisms. But, if NPaaS runs user-provided code on provider-managed devices that lack hardware isolation, such as programmable switches, a major challenge is to ensure user programs don't abuse access to the switch. A promising option is software isolation (SI), as proposed by Singularity [105], where user supplied programs are type and memory safe and restricted to use specific interfaces. Proposals to support this notion of isolation already exist for switches [125] and Netbricks [149] uses SI to eliminate hardware-isolation costs to improve performance. Similarly, Azure SmartNICs let users write policies for the Virtual Filtering Platform (VFP) [58], which implicitly restricts them to the user's network. NPaaS can provide operations which are guaranteed to only act on traffic belonging to the requester.

#4: Failure handling and debugging.

How should NPaaS recover from failure? NPaaS systems will be composed of heterogeneous devices with their own failure modes. Detecting failures will be an ongoing issue that NPaaS systems must address. Prior work handles failures by restarting jobs on new nodes [66, 111] or by routing around the failure [124]. The best option depends on the expected lifetime of the pipeline, likelihood of a failure, and any existing failure recovery mechanisms. For short-lived jobs, failure is unlikely and restarting is fast, so a lightweight mechanism like restarting is ideal [111]. For longer-lived jobs, partial recovery becomes ideal [124]. In either case, the calling framework may provide better failure recovery than NPaaS can provide (e.g., Spark's lineage graph [191]).

How can we debug in-network programs? Data processing errors are often

[5]Putting aside recently discovered side-channel vulnerabilities.

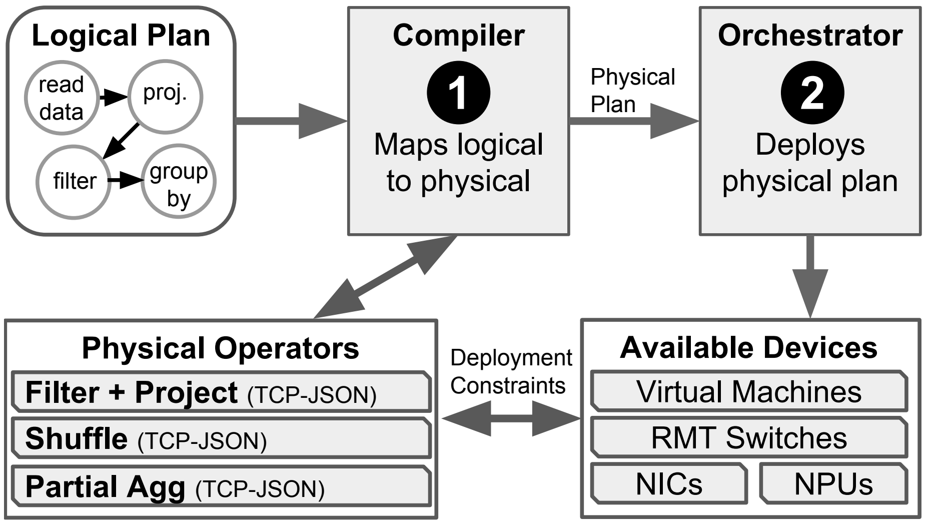


Figure 2.2: Overview of the how Jumpgate interacts with a data processing system to compile and orchestrate in-network processing.

data dependent and difficult to trace because errors are caused by a few malformed data records [80]. As with existing distributed computation systems, NPaaS should return a summary of any problems encountered during execution to the framework, including enough context to help find the problematic record(s).

#5: Who should provide NPaaS?

Is it better if the cloud provider or the user operates NPaaS? The cloud provider has a privileged view of the network topology, lower level control over the hardware and network configuration, and often more money to pay experts to develop fast operators. On the other hand, users have a better understanding of their specific workload and could implement workload-specific operators. Users may also prefer to keep any needed encryption keys on infrastructure they own. One way to capture the best of both is for the user to operate the NPaaS system, while allocating proprietary in-network operators from the cloud provider.

2.3 Jumpgate: The first NPaaS implementation

Note: This section includes a description of Jumpgate as it was envisioned when this chapter was published as a paper. Chapter 3 describes the latest implementation in more detail, but I have left this section mostly as-is, to show how the system design was refined over time. Section 3.8.1 has a description of the differences between the proposed system and the realized system.

As we've discussed, accelerating analytics with in-network processors is beneficial, and there are a number of issues that need to be explored in this space. However, the critical piece that is currently missing is an NPaaS system that integrates with existing systems to run analytics tasks on in-network processors.

Prior work either doesn't integrate into existing systems, or does so in a way that directly targets a specific in-network processor. Instead, we need a system that can integrate with existing analytics systems and work around the limitations of in-network processors to execute queries in a way that avoids manual integration and decouples analytics implementations on in-network processors from analytics systems. *Without such a system, there is no framework to evaluate the issues we disucssed above:* better data formats, network transports, failure handling, multi-tenancy, and which parties should provide NPaaS.

To address this need, we propose Jumpgate, a compiler and orchestration system that can provide NPaaS. Jumpgate provides a client-facing API and maps client requests to relevant devices using an extensible architecture that simplifies adding new operators and devices without modifying the client.

Figure 2.2 shows a sketch of a client's interaction with Jumpgate. **Step 1:** the compiler receives a logical plan of operations from data analytics frameworks (e.g., filter, project, shuffle, partial or full aggregation) and maps logical operations to stages of physical operators that can be deployed on available devices. **Step 2:** the orchestration layer coordinates runtime execution of the physical plan on devices. Devices are allocated and initialized to perform the physical operations[6]. After initialization, network addresses are known and can be propagated as needed (e.g., end-host addresses, or next-hop addresses in a processing pipeline).

[6] If any physical operators need to generate and compile code, it could happen at this point.

Client API. Jumpgate's API allows a client to specify input data sources (files or network tuples), a DAG of logical operations to apply to the input data, and the destination network tuples for receiving processed data.

Mapping. The compiler maps logical operations to stages of pipelines of physical operators. For example, joins require a build stage to load data into the device and then a probe stage to output matched records. Since in-network operations execute concurrently in pipelines, we must make sure that other operators execute at the correct time to feed and receive data from the join. Jumpgate computes all needed stages and which stage(s) each operator executes in using a dependency-driven simulation. Logical operators are then mapped to physical operators, as determined by each physical operator's matching functions (below).

Extensibility. Jumpgate supports adding new logical and physical operators. For instance, a logical operator that translates unstructured to structured data, or a physical operator that runs filters on supporting storage systems. Logical operations are represented as typed nodes in the DAG with specified input/output connectivity and stage logic. Physical operators are comprised of: (1) matching functions and (2) a controller used to drive device allocation and execution. Matching functions determine which logical operators can be replaced based on type and properties, such as available resources and operator-specific parameters.

Limitations and Future Work A compiler and orchestration layer are bare necessities for providing NPaaS, but are not sufficient for a full NPaaS system. At the very least, NPaaS also needs failure handling and topology-aware scheduling. For now, we are focused on enabling analytics applications to execute in-network computations on software and programmable in-network devices, supporting common operations, and measuring performance. Our eventual goal is to share Jumpgate with other researchers in order to answer broader challenges posed by this chapter and lay fertile groundwork for research into making in-network processing available to more users.

2.4 Conclusion

This chapter established that in-network compute capability is growing, and the benefits to data analytics are clear. To allow all analytics systems to benefit, we envision providing in-network processing as a service (NPaaS) to abstract the desired operations on data from their low-level implementation on network processors. The next chapter describes the refined implementation of Jumpgate, the first NPaaS system we know of, based on the motivation and challenges presented above.

Chapter 3

Jumpgate: Network Connected Acceleration for Data Analytics

Chapter 2 showed that Network connected accelerators (NCAs), such as dataplane programmable switches, have been able to accelerate data analytics tasks by as much as 2-10$\times$ [84, 111, 175]. But, while prior work was promising, there has been little prior work on how to use NCAs automatically from existing analytic systems. Ideally we could simply run tasks on promising NCAs from existing analytics systems. But, existing analytics systems cannot yet offload tasks to NCAs: they assume nodes can execute arbitrary operations, have 100s of gigabytes of local storage, and can exchange serialized data over complex RPC. But, NCAs are limited in the operations they can execute, how much data they can store, and the data formats they can operate on. NCAs are under active research, so manually integrating each promising NCA into each analytics system presents a challenging integration task.

This chapter details Jumpgate, a system that executes relational (SQL) queries from existing analytics systems on NCAs. Jumpgate reduces integration effort with high-level client and NCA interfaces. Jumpgate implements novel, but simple, task execution and communication paradigms that cope with NCA limitations. Jumpgate configures NCAs on a per-query basis to trade higher setup latency for low execution overhead. We made Apache Spark a Jumpgate client with a modest effort of 2,200 LoC. We integrated several NCAs into Jumpgate: our own software-

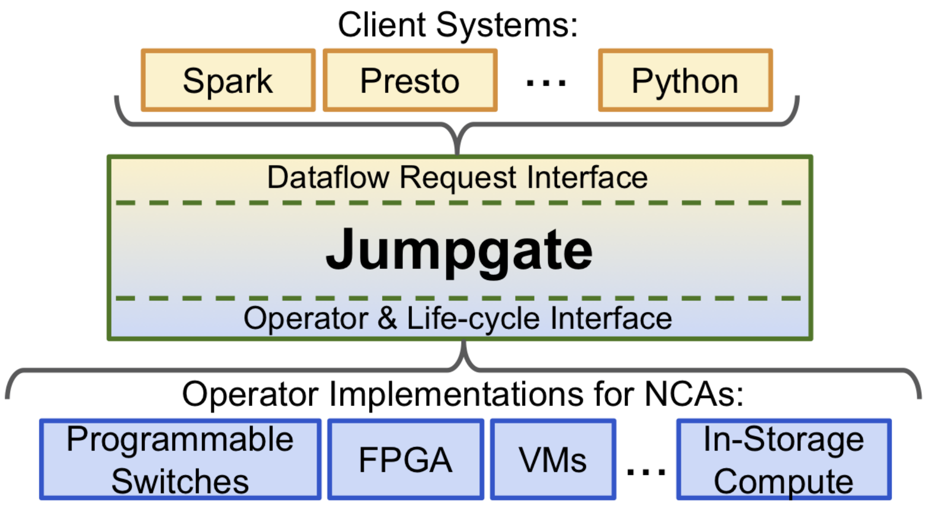


Figure 3.1: Jumpgate bridges analytics systems and network accelerators.

based accelerators and a programmable data plane switch (in 150-300 LoC), that we then use from Spark. We found that Jumpgate can reduce data volume sent to client systems by orders of magnitude and our NCAs can reduce some query runtimes by $1.12-3\times$.

> *Contents of this chapter is under submission to the USENIX Conference on Operating Systems Design and Implementation (OSDI).* An earlier version of this work was submitted to the USENIX Conference on Networked Systems Design and Implementation (NSDI). While that submission was rejected from NSDI, the reviews on the prior submission were encouraging and helped to improve the work presented below, particularly the evaluation and presentation of the contributions.

3.1 Introduction

As Dennard scaling and Moore's law reach their limits, system designers are turning to domain-specific accelerators. In-network processors show great promise as accelerators for data analytics. Recently studied accelerators 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 tasks and a 3-7 orders of magnitude reduction in network traffic [69, 84, 111, 176]. More generally, FPGAs, SmartNICs, and even network-based software accelerators provide speedups of 2-10× for analytics tasks (§3.2).

As workload accelerators, in-network processors have attractive properties because the network is their primary data path: they can be scaled independently from other systems, can be allocated on demand, and can interact with other networked systems. We take a broad view of potential accelerator implementations: we view any processor or software program capable of exchanging data via the network as a potential *Network Connected Accelerator (NCA)*.

Prior research showed NCAs can accelerate analytics queries, but presented point solutions lacking common orchestration, abstraction, and execution models. As a result, using NCAs in analytics systems, like Apache Spark or Presto, requires manual integration of each NCA into each system.

Integrating NCAs into existing systems is challenging because analytics systems generate tasks that only general-purpose CPUs can execute. Typical tasks consist of chained relational operations (e.g., *scan*, *group-by*, *join*) that may access 10-100s gigabytes of local memory and storage and exchange data via RPC or shared memory using complex data formats [30, 169, 191]. In contrast, NCAs, like *programmable dataplane switches* [37, 38, 42], achieve high bandwidth and low-latency packet processing, but have limited local storage, cannot implement complex protocols like RPC, can only operate on ≈100s bytes of each packet, and cannot parse complex data formats (§3.2). *Using NCAs, like programmable switches, for data analytics requires simplifying tasks, data exchange, and orchestration.*

To integrate these NCAs with analytics systems, prior work had to manually: (1) identify operations to offload in each query that were suitable for a given NCA, (2) devise data formats that can be read by the NCA, (3) write converters to and from those formats, and (4) orchestrate execution of the converters, NCAs, and the analytics system. Also, users of these systems are left to to decide whether offloading given operations is worthwhile given the setup cost. The goal of this work is to build a system that automates these tasks.

Contributions – This chapter presents the design and evaluation of Jumpgate. Jumpgate enables existing systems to execute relational (SQL) queries on NCAs, so users of these systems can benefit from NCA performance without changing their existing workflows (Figure 3.1). Jumpgate's design makes several contributions.

- To address the problem of manually finding applicable operations for NCAs, Jumpgate allows clients to issue requests by submitting a dataflow graph that is *already present* in existing analytics systems. Clients offload queries *bottom-op* from scans over storage which is often where the bulk of data processing work lies, enabling more opportunities to use NCAs than prior work. Jumpgate can then find applicable NCAs to use (if any). We show that this interface reduces the effort for client systems to use Jumpgate to a one-time integration cost of 2,200 lines of code.
- Jumpgate coordinates execution of NCAs to address their storage limitations. To do so, Jumpgate uses a novel task execution and communication paradigm called *Staged Networked Pipelines* (SNPs). NCAs that have limited storage need to run concurrently with their senders and receivers, so data can be forwarded as it is received. NCAs that implement hash-joins must receive all data from their build side before receiving data from the probe side. Schedulers in existing analytics systems do not provide these guarantees. SNPs groups NCAs that must execute concurrently into a stage, and SNPs schedule NCAs that implement joins receive data across two stages so that all build data is received before probe data. 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*.
- To simplify data formats so that NCAs can operate on them, Jumpgate computes simple formats for NCAs to exchange data, called *network tuple formats* (NTFs). NTF layouts are passed to Jumpgate generates software NCAs that convert input data to NTF. NTFs address the per-packet read limitations of NCAs and relieve them from having to decode complex formats.
- Prior work only targets a single NCA directly in the analytics system, but with Jumpgate, our goal is to use multiple NCAs. We expect to have several NCA

implementations with varying applicability to different analytics operations and device-specific control planes. To identify when NCAs can be used, and to allow Jumpgate to interact with multiple heterogeneous NCAs, Jumpgate contributes two hardware agnostic interfaces: (1) An *operator interface* that describes the relational operations and data format compatibility of each NCA. (2) A *life-cycle interface* that allows Jumpgate to coordinate execution between multiple hetereogeneous NCAs. To integrate Jumpgate with a new NCA, designers add an implementation of these interfaces to Jumpgate. We use both interfaces to add software NCAs for many relational operators, and an NCA in a programmable dataplane switch for aggregation.

- We built a prototype of Jumpgate to perform an evaluation of Jumpgate's approach to offloading client operations, overheads, and *to contribute insights into the behaviour and performance potential of using NCAs for data analytics to motivate future work.* We describe our findings below.

Our evaluation finds that Spark can offload $\approx 60\%$ of operations from TPC-DS (a popular SQL benchmark) to Jumpgate. The runtime overhead of SNPs can be under 100ms. But, the setup overhead of compiling software NCAs to convert input data to NTFs can be relatively large, $\approx 2-6s$, for short jobs. Compilation overhead can be reduced, but it underscores that the ability to weigh the setup costs and throughput gains is crucial when using a system like Jumpgate. We added a simple heuristic to Spark that can effectively make this trade-off.

Prior work has already established that individual NCAs can accelerate analytics tasks. So, our goal in this chapter and in building Jumpgate is to automate integration of accelerators, not to build new ones. Nevertheless, to rigorously evaluate Jumpgate we built an *aggregate* accelerator for a programmable switch and software-based accelerators for *scan*, *filter*, *project*, *join* and *aggregate*. We integrated them into Jumpgate in $146-343$ LoC.

We find that *shifting data processing to NCAs can reduce the data read by analytics systems by orders of magnitude* (§3.5.5). Coupled with fast-enough NCAs, Jumpgate accelerates certain queries in Spark by $1.12-3\times$.

However, offloading work to NCAs takes the data that would normally be materialized in an analytic client's memory, and puts it into the network, flowing between NCAs. Our experiments demonstrate this underscores the need for fast software networking that can feed NCAs, and perhaps development of NCAs that can accept large packets sent from user space networking.

3.2 Background and Related Work

Apache Spark. Spark [27] is a widely used data analytics system. To use Spark, users write either custom code in Scala or Python, or SQL queries. Jumpgate focuses on the latter. A SQL query submitted to Spark is compiled into a dataflow DAG and executed on a Spark cluster. Spark clusters are organized into a master with many workers. Workers process data, while the master coordinates workers.

Dataflow Graphs in Analytics Systems. Data analytics engines translate a user's query into a *logical* graph of operations, and then map this graph to *physical* graph of implementations that perform the operations. Analytics dataflow graphs are so similar between systems that they can be translated from one system to another and unified [64, 133, 146, 148]. Jumpgate leverages this property to provide a general dataflow request API for analytics clients.

Network Connected Accelerators. Jumpgate targets existing *and future network connected accelerators*. As we saw in chapter 2, in-network processing is an active research area. Researchers are applying existing in-network processors to analytics [175], machine learning [167], distributed storage [195] and consensus [113]. Hardware researchers are proposing new in-network architectures such as integrating packet processors directly with general purpose CPUs [79] and loosely coupled designs [42].

Our exemplary hardware is today's programmable dataplane switch ASICs. Programmable switches process network packets at high rates (≈6.4Tbps) and run custom protocols and switching rules, often written in P4 [38]. Chapter 2 showed recent research has used them for analytics tasks, often with 10-1000× speedups, thanks to the performance of their fixed latency pipeline [66, 69, 85, 86, 111, 166, 175]. Again, while useful, programmable switches are limited in terms of storage (≈10s of MB), bytes processed per packet (≈100s), and operations per packet [155, 166]. By targeting such restricted hardware, we aim to ensure that Jumpgate's design applies to future NCAs.

Jumpgate also targets what we call *software NCAs*: our term for software running on general purpose CPUs that inter-operates with other NCAs, allowing Jumpgate to process data other NCAs cannot. For instance, Jumpgate has a software NCA that uses simdjson [104] to transcode JSON to network tuple formats. *Software NCAs allow Jumpgate to tackle the co-design challenges around data-formats and network transports described in chapter 2 by transforming data in storage into a network tuple format more appropriate for other NCAs.*

In general, we see potential for NCAs on a growing variety of hardware, especially as researchers explore hardware resource disaggregation [8, 170]. For instance: *Storage accelerators* [49, 67, 82, 88, 99, 189] could be used to send data to other NCAs. *SmartNICs* (FPGA or ASIC-based) could be used as standalone NCAs [117, 118], or integrated with hosts for low latency communication [92, 151]

Existing Analytics Systems. A key difference from existing data analytics systems is Jumpgate's orchestration of computation as staged networked pipelines (SNPs) and Jumpgate's data exchange in network tuple formats without using RPC, which is necessary to work with limited NCAs. Analytics systems, like Apache Spark [27], Hadoop [22], or Dryad [81] require tasks to read intermediate data from memory or storage. Presto [169] pipelines concurrent tasks, but producers buffer data locally until requested by a consumer via HTTP. Stream processing systems like Apache Flink [21] run operators concurrently but schedule large tasks of pipelined operations. While none of these systems are able to seamlessly integrate NCAs, they provide valuable functionality, and Jumpgate enables them to use NCAs.

Prior Work on NCAs for Analytics. Jumpgate is the first system to automate the use of networked accelerators by analytics systems in a way that does not require manual integration of *each* accelerator into *each* analytics system.

Aside from our published work on this topic (Ch. 2), the most recent related system is Cheetah [175, 176]. Cheetah interposes between Spark master and workers to prune data just before it is returned to the client. But, Cheetah can only process (almost) final results and cannot pass data back to Spark for further processing. Instead of just processing final results with accelerators, Jumpgate reads directly from storage so that *lower-levels* of the query plan can be offloaded.

Overall, prior work demonstrates that in-network processing can improve analytics performance, but required manual effort to integrate each accelerator into each analytics system and to orchestrate their execution. No prior work we know of addresses these integration and orchestration issues directly. For instance, Cheetah, NetAccel [111], DAIET [166], PPS [84] and SwitchML [167] each had to write custom programs to send and receive data from the client's internal format into a specific format for their target accelerators. Prior work often does not discuss how accelerators are deployed and managed in depth, but most works appear to manually deploy their programs on switches. We believe accelerators from prior work could be adapted to Jumpgate. *In order for these promising accelerators to become widely used, these integration tasks must be automated.*

3.3 Jumpgate Design

3.3.1 Design Goals

1. Support important operations, even if no efficient accelerators exist yet. We developed Jumpgate to integrate current NCAs and to study how future NCAs can help analytics. Jumpgate aims to accept and run as many operations as possible, and uses software-based NCAs when no hardware NCAs are available.

2. Support real-world data formats. Jumpgate supports two input formats: JSON and ORC. JSON is endemic, despite its notoriously low parsing speeds [104, 115, 147]. ORC is one of many data formats that are optimized to improve storage and throughput of analytics [19, 20, 26, 129, 178]. No accelerators we know of can directly parse any analytics format. So, Jumpgate has software-NCAs that transcode ORC and JSON data to network tuple formats.

3. Allow clients to specify the receiving protocol. Prior work on programmable switches for analytics send a single tuple/row per UDP packet with an added reliability layer [85, 167, 176, 195]. This requires modifying the client analytics systems to be able to receive UDP packets at high speed, often using the DPDK [50]. We think this requirement can be an impediment to adoption. Instead, Jumpgate allows clients to specify the preferred transport in the request. While this may conflict with the NCAs that are able to be used for their request, this choice allows clients to more easily adopt Jumpgate. Jumpgate is transport angostic to enable experimentation with new transports in the future.

3.3.2 Scope and Assumptions

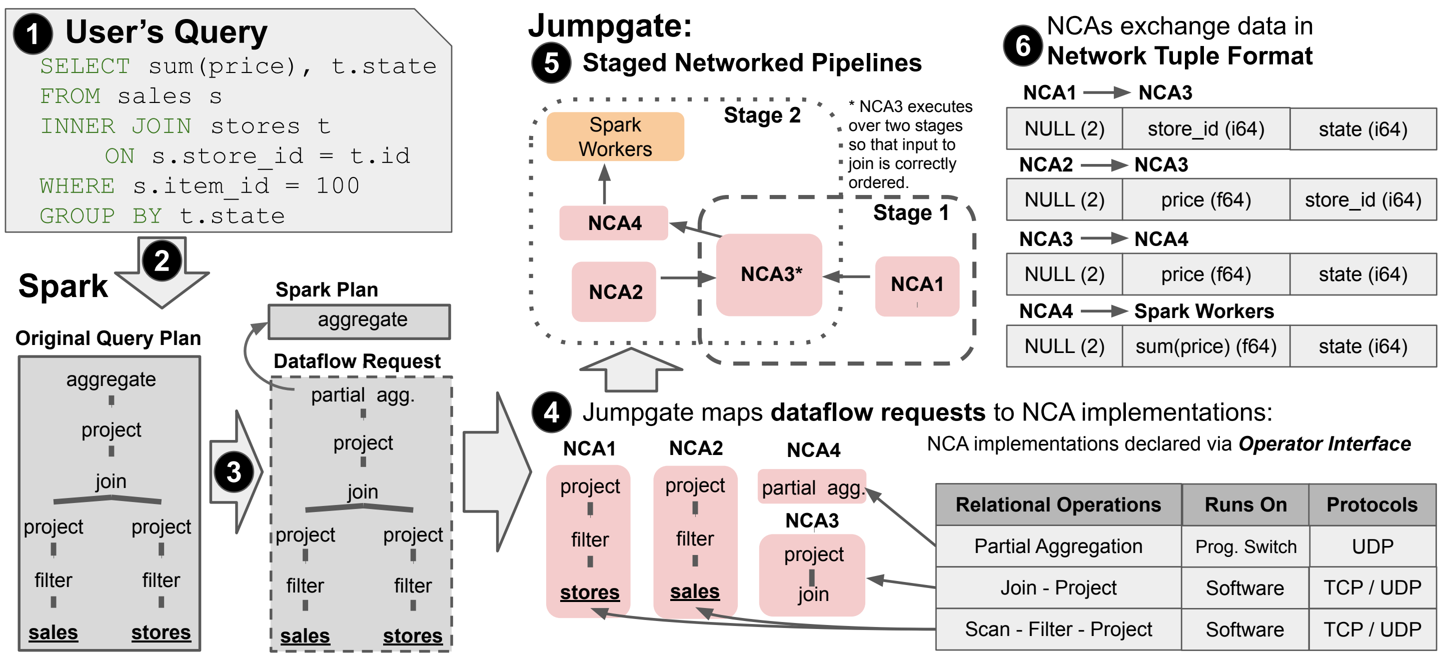
The current design of Jumpgate presented in this chapter focuses on enabling execution of client dataflow requests on NCAs. We do not yet investigate how to operate Jumpgate at data-centre scales, or how to efficiently allocate resources with respect to network topology, data locality, or resource consumption.

We assume there are NCA designers who can implement analytics operations for a particular device. But, we do not expect there is a single implementation that can work for *all* queries. Instead, we expect these implementations may need to be configured and/or compiled on a per-query basis. So, Jumpgate provides APIs – the operator and life-cycle interfaces – that NCA designers implement to invoke device-specific techniques to configure, compile, and control their execution.

We make this assumption because prior work has shown that there are already promising query-specific implementations of analytics operations for NCAs, such as joins [111, 175], aggregation [166] and filtering [175]. Prior work has also demonstrated several techniques to generate code for software [140], programmable NICs [151, 154], and programmable switches [172]. For example Cheetah [175] re-configures a filter implementation that has been pre-deployed to a programmable switch. While there are few analytics-specific configuration or code generation techniques in the literature for in-network processors, I hope Jumpgate inspires more. We implemented several software-based NCAs that follow this assumption to generate query-specific NCAs from templates we wrote (§ 3.4).

We assume it is possible to allocate devices to run generated NCA implementations using a resource allocation system like Kubernetes [100] or others [73, 168]. NCA designers can invoke device-specific allocation and management when Jumpgate calls implementations of the life-cycle interface (§3.3.4). Since Jumpgate allocates NCAs on demand, serverless platforms [11, 24] could be used if/when they support sockets.

Jumpgate presumes the client will retry failed queries. Our Spark integration does this. Themis [159] and Presto [169] note that failure recovery is expensive with little benefit, even at ≈ 1000 nodes, when job times are under a few hours.



```
SELECT sum(price), t.state
FROM sales s
INNER JOIN stores t
     ON s.store_id = t.id
WHERE s.item_id = 100
GROUP BY t.state
```

NCA1 → NCA3

NULL (2)	store_id (i64)	state (i64)

NCA2 → NCA3

NULL (2)	price (f64)	store_id (i64)

NCA3 → NCA4

NULL (2)	price (f64)	state (i64)

NCA4 → Spark Workers

NULL (2)	sum(price) (f64)	state (i64)

Relational Operations	Runs On	Protocols
Partial Aggregation	Prog. Switch	UDP
Join - Project	Software	TCP / UDP
Scan - Filter - Project	Software	TCP / UDP

Figure 3.2: How Jumpgate interacts with client analytics systems and accelerators to deliver processed data to client endpoints. (1) – (2) Users submit queries to the analytics system. (3) The client analytics system (i.e. Apache Spark) decides to offload part of the query to Jumpgate. (4) Jumpgate maps client requests to available NCAs, (5) generates code for NCAs and orchestrates execution, sending processed data in network tuple format (6) towards waiting client endpoints.

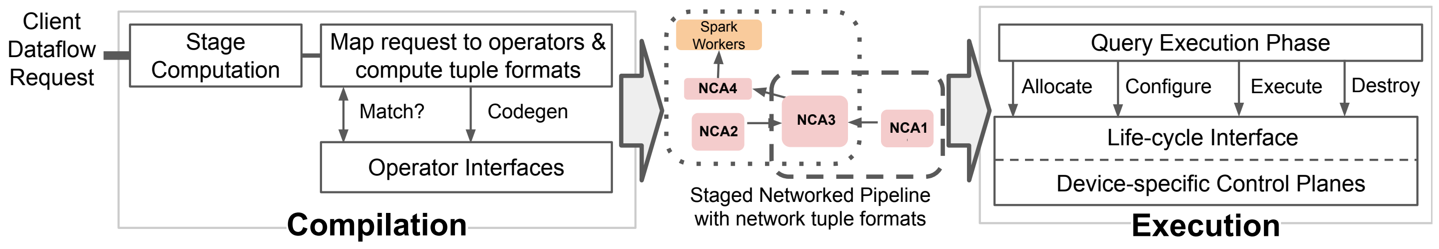


Figure 3.3: Compilation and execution phases within Jumpgate.

3.3.3 Jumpgate Overview

Figure 3.2 illustrates an end-to-end example in which Jumpgate is used by Spark to run a user query. We use this running example for the remainder of the paper. Figure 3.3 shows Jumpgate's compilation and execution phases. ❶ The user submits a SQL query to Spark to calculate total sales from each store for a given item, grouped by the store's state. ❷ 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. Without Jumpgate, Spark would execute this plan on worker nodes. ❸ To use Jumpgate, Spark splits the plan into two: a plan for its workers and a **dataflow request** for Jumpgate that will send data to its workers. We detail the client interface and how splitting is done in §3.3.4. In this case, Spark offloads most of the plan to Jumpgate, but splits the aggregation operation into a partial aggregation to be done by Jumpgate and a full aggregation to be finished by Spark's workers. In this example, Spark's request tells Jumpgate to send data over UDP (not shown) so that programmable switch aggregation can be used.

❹ Spark submits the request to Jumpgate. Jumpgate maps the logical dataflow request to a set of NCA implementations that can run the requested operations. Mapping is detailed in §3.3.5. Jumpgate essentially has a library of NCA implementations, and queries each implementation (using the **operator interface**) to see if it is capable of running any requested operations and communicating with adjacent NCAs. In this case, Jumpgate chooses 4 NCA implementations, **NCA1 – 4**, from the available operator implementations shown in the figure. The full set of operators is presented in §3.4. Jumpgate calls a compilation function for each chosen NCA implementations to specialize each one for the specific operations they will run and the data format(s) they will receive and send. For instance, Jumpgate's software operators generate and compile C code that implements the needed operations. For this work, we assume there are device-specific ways to allocate resources for each NCA (§3.3.2). The interfaces are detailed in §3.3.4.

❺ Jumpgate coordinates execution using staged networked pipelines to address the storage limitations of NCAs. Jumpgate organizes the set of selected NCAs into stages to ensure that most NCAs do not need to store intermediate data by making sure their consumers are running and ready to receive data. **NCA3** executes over two stages because it executes a join operation, which needs to receive all data from one side of the join before the other side can begin sending. In stage 1, **NCA1** reads the store table and sends data to **NCA3** to build an in-memory hash-table 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. Jumpgate initializes instances of each operator on the allocated resources and controls their execution using the **life-cycle interface** which bridges between Jumpgate and device-specific allocation and management control planes (not shown). The phases of computing an SNP and execution are detailed in §3.3.5.

❻ As Jumpgate signals NCAs to execute, producer NCAs send data to consumer NCAs in **network tuple formats (NTFs)**, over the requested transport (UDP). NTFs address the problem of NCAs only being able to work on simple data formats. NCAs can communicate using any transport, and Jumpgate ensures both sides support it. Jumpgate generates a precise definition of each tuple format that is passed – via the operator interface – to code written by NCA designers, so they can generate code or configuration for their NCA to efficiently parse tuples. The record format includes a null vector with at least 1 bit per field and supports fixed- or variable-length fields (§3.3.5). Jumpgate returns a specification of the output format to Spark so it can generate code to efficiently receive data (§3.5.2).

Summary– This overview highlights the importance of each of the novel abstractions in Jumpgate. **1. Dataflow requests** allow easy interfacing with analytics systems. **2. 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. **3. Network Tuple Formats** solve the per-packet read limitations of NCAs by simplifying the data format transmitted between NCAs, allowing software and hardware-based NCAs to interact in executing a user's query. **4. The operator and life-cycle interfaces** allow Jumpgate to query the capabilities and limitations of an NCA and to control heterogeneous NCAs.

3.3.4 Jumpgate's Interfaces

We now describe Jumpgate's client and operator interfaces that are used by the compilation and execution components (Figure 3.3). In this section, first we describe the mechanisms that Jumpgate implements, and then we describe how they address the challenges and limitations to using NCAs from existing analytics systems.

```
sales = read("/path/to/sales")
  .filter("item_id == 100")
  .project(["store_id", "price"])
stores = read("/path/to/stores")
  .project(["store_id", "state"])
agg = sales.join(stores, on="store_id")
  .project(["price", "state"])
  .aggregate("sum(price)", groupby=["state"])
s = agg.shuffle(keys="state", 2)
s[0].send(ip="10.0.0.1" port=1234, "TCP")
s[1].send(ip="10.0.0.2" port=4567, "TCP")
```

Figure 3.4: Jumpgate Dataflow Request example, derived from the user query in Figure 3.2. This builds a dataflow graph that performs the offloaded part of the query, and sends results to two client machines waiting for data.

Client Interface: Dataflow Requests

Job Request Flow. Clients construct a dataflow request as JSON data, and submit it to Jumpgate over HTTP. Jumpgate has endpoints for job compilation, job execution, and job status. When clients submit their job, Jumpgate returns a job ID, a description of the chosen NCAs, and the network tuple format the client will receive. When the client's endpoints are ready to receive data, the client submits the job ID to Jumpgate to begin execution, and Jumpgate internally signals the different NCAs to begin working and sending data to the client.

Dataflow Request API. While Jumpgate jobs are specified in JSON, they are better explained using a functional-style interface. Figure 3.4 shows a request that corresponds to the offloaded part of the job from Figure 3.2. The complete job

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, num_destinations*	Send records to different destinations. 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 3.1: Jumpgate's Client API: supported operations and their parameters.

specification is shown in Appendix A.1.1. The client needs to provide the location of the input data, the operations to perform, and the IP and ports where data should be transmitted. We omit the full JSON request for brevity – it is meant to be machine-generated.

Table 3.1 summarizes the operations Jumpgate supports. Most of these will be familiar to users of data analytics, but Jumpgate adds one that is important for returning data to the client: *send*. *Send* specifies how and where to transmit the input data, including address, port, and transport protocol the client is expecting. Clients can insert a *shuffle* operation to direct results to multiple receive nodes.

Schemas and Expressions. In data processing systems, each operation has an output schema that maps from names to types. Subsequent operations reference these names. Jumpgate needs the client to explicitly name the results of expressions, so there is no ambiguity. For instance, in Figure 3.4, Jumpgate needs the client's name for subsequent references to `sum(price)`.

Why a dataflow interface? Prior work requires each client analytics system to understand each NCA that is available and to be able to generate a low-level request to use the NCA. Instead, with Jumpgate, this request flow allows clients to submit operations to Jumpgate in operations it already understands (i.e. a dataflow graph). The client does not need to know which NCAs are available to Jumpgate, but it can inspect the response to determine if the NCAs selected by Jumpgate are desirable. In the future, Jumpgate could return predicted throughput that the client could use to decide if executing the request using Jumpgate is worthwhile. We have not yet explored these interactions yet.

Prior work was able to show that NCAs were promising for analytics tasks, but some, like NetAccel [111], manually offloaded queries to their NCAs and others, like Cheetah [175], aimed to only process almost-final results and presents results from 9 queries. **These approaches inherently limited the number of queries these systems are able to effectively process.** Using a dataflow interface allows clients to offload queries *bottom-up* from storage which is often where the bulk of data processing work lies, enabling more opportunities to use NCAs. Since clients only need to communicate *what operations* they want to offload without knowledge of the underlying NCAs, the dataflow interface means that analytics clients can automatically offload requests to NCAs, pushing the responsibility of finding appropriate NCAs to Jumpgate.

The current dataflow API mirrors the dataflow operations that analytics systems use to execute SQL queries. The interface could be extended with other operations (i.e. general map and reduce) to support more flexible NCAs in the future.

Operator Interfaces

Jumpgate uses the *operator interface* to first find NCAs that can implement logical operations in the dataflow request, and then replace each operator with an NCA instance. §3.3.5 describes this replacement. Table 3.2 summarizes the API that NCA designers implement. `match_input` and `match_operations` inspect the input the NCA would receive, the logical operation's given expressions, and any subsequent operations that can be fused. `match_input` accepts a description of the NTF layout, and a opaque string describing the transport the data will be received on. `match_operations` accepts a node from the dataflow graph, and can inspect this node's expression AST to determine any further applicability. `match_operations` can also inspect subsequent nodes of the dataflow graph if the NCA can implement operator fusion. `match_output` returns the NTF the NCA would emit and the transport that would be used. Appendix A.1.2 lists the pseudo-code for the operator interface of the JSON to NTF conversion/filtering/projection NCA for the running example.

To ensure sufficient resources are available, the operator interface specifies a resource manager. During compilation, Jumpgate ensures the manager has free resources for the given NCA before deciding to use it. During execution, the NCA's life-cycle interface accesses the reserved resources.

Why the operator interface? Since prior work only targets a single NCA directly in the analytics system, they do not require something like the operator interface. However, in order to use multiple NCAs, we expect to have several NCAs implementations with varying applicability. For instance, some NCAs might be able to implement *any* join operation, but others may only be able to implement joins with specific characteristics, such as a single 32-bit join key. So, we need the operator interface to find applicable NCAs to use. Instead of trying to create a very complex interface that is challenging to implement or narrow interface that only works for some NCAs, we aim to give NCA designers freedom to implement any detailed applicability checks their device may need. Our implementation provides helper functions to perform common applicability checks and walk expression ASTs.

The operator interface gives Jumpgate the freedom to find candidate NCAs that can replace logical operations in the dataflow graph and explore the space of possible replacements, as described in §3.3.5. At the same time, the NCA designer can write functions that simply return whether or not the NCA can be used for a given operation.

Life-cycle Interface

Once a set of NCA instances has been selected (described in §3.3.5), Jumpgate uses the *life-cycle interface* to configure each instance for the query, connect them, and control execution. The life-cycle interface separates Jumpgate's control plane from the NCA data plane, allowing NCA implementations to be written in any language, framework or hardware device. To make an NCA work with Jumpgate, the designer implements the life-cycle interface for their NCA in Jumpgate that can communicate with the NCA implementation. Table 3.3 summarizes the life-cycle API. §3.3.5 describes how it it is called. Pseudo-code for the life-cycle interface of an NCA from our running example is shown in Appendix A.1.3.

Why the life-cycle interface? Again, prior work targets a single NCA, so they did not require a life-cycle interface. But, since we want to use multiple NCAs, we expect that each NCAs will have its own control plane that requires particular ways of allocating, starting, running and stopping execution. For instance, one can interact with programmable switches by sending RPCs, or one can execute software on allocated VMs using SSH. So, Jumpgate provides a set of interfaces that describe what Jumpgate needs to do, and relies on the NCA designer to implement the idiosyncrasies for their particular analytics implementation.

We designed the life-cycle interface so that each step could be executed on each NCA in parallel lock-step to help reduce overheads. In other words, the API is designed so that all NCAs could be running the `compile` phase concurrently

Name	Meaning
match_input	Return true when the NCA accepts the input NTF on a given transport.
match_operations	Return the operation and any subsequent operations if the NCA can implement them.
match_output	Return the NTF and transport the NCA would emit.

Table 3.2: 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 3.3: Life-cycle interface to control NCAs.

before proceeding to the next life-cycle phase.

The life-cycle interface also gives the freedom to implement NCAs that use multiple devices in ways that Jumpgate cannot directly handle, such as a join that uses software for the build phase to build index structures and then offloads the checks to accelerator for the probe phase. We did not explore these in this work, but we elaborate on this idea in §3.7.

Summary. These interfaces help abstract running tasks on heterogeneous NCAs, both for the client and for NCA designers. These interfaces are intentionally *simple*, not tied to specific hardware, and allow analytics systems and NCAs to easily interface with systems like Jumpgate. In §3.4 and §3.5.2, we evaluate the LoC needed to implement these interfaces.

3.3.5 Compiling Staged Networked Pipelines

This section describes how Jumpgate compiles dataflow requests to staged network pipelines and executes the SNP (Figure 3.3).

Stage Computation

Jumpgate computes stages so that input ordering is respected while NCAs run concurrently. This addresses the problem of NCAs having limited storage by ensuring that NCAs do not need to store lots of intermediate data, unless they implement joins which inherently requires storing data. Jumpgate joins are implemented as hash joins, where the hash table is first built (the build phase), and then it is probed to emit records (the probe phase). The challenge is to ensure that (1) NCAs that feed a `join`'s build phase run before *any* NCAs that feed the `join`'s probe phase, and (2) during the probe phase, the NCAs that send to and receive from `join` are both running.

To compute stages, Jumpgate adds all runnable operations to the current stage, marks nodes as executed, and starts a new stage. Jumpgate repeats this process until all operations are completed. Streaming operations are runnable when both their inputs *and* output destinations are runnable. Joins in the build stage are runnable when their build input is runnable, and in the probe stage when their probe input and outputs are runnable. This is a slight twist on topological sort: nodes become runnable as their inputs are satisfied, but with the extra constraint that receivers *must also be runnable.* Pseudocode for this algorithm is supplied in Appendix A.1.4. Staging is performed on logical operations, and the NCAs that implement these operations inherit their assigned stages.

Mapping Dataflow Operations to NCAs

This section describes how Jumpgate maps the operations in dataflow requests to NCAs. Jumpgate has a library of available NCAs declared by designers via the operator interface. The operator interface API determines when an NCA can be used to implement a logical operation in the graph (§3.3.4). Jumpgate iteratively transforms a dataflow request into a graph of NCA instances by repeatedly picking an operation to replace, and calling each NCA's operator interface. Jumpgate will

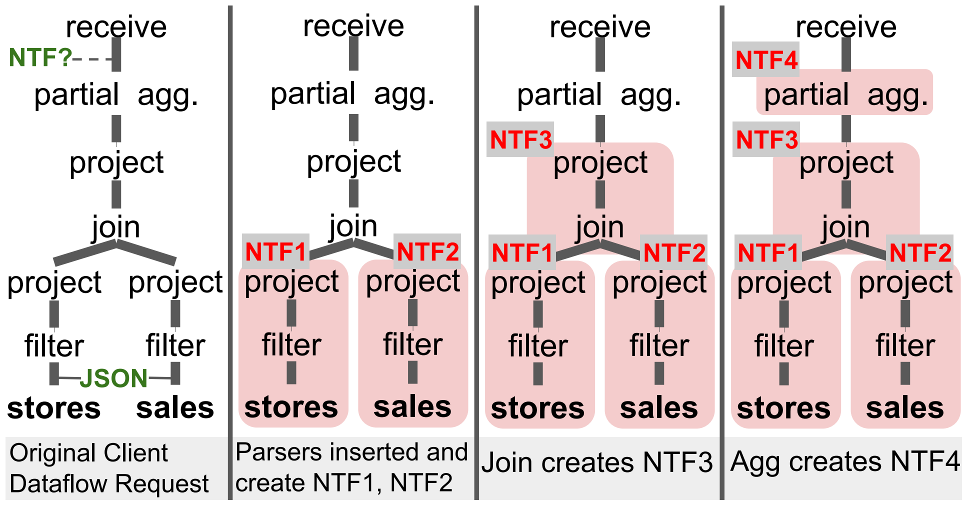


Figure 3.5: Compilation steps when compiling the example request from Figure 3.2. Compilation proceeds from left to right. `NTF?` denotes the client will receive data in NTF. `NTF#` denotes specific NTFs that are output by NCAs.

not insert an NCA unless it replaces at least one operation from the original graph.

The challenge is to insert NCAs that convert the input data into NTFs, and to ensure the client can receive on the transport used by the final NCA. When `match_input` is called, the given transport/format is found by following the input edges towards the source, until a previously inserted NCA or the source is reached. When `match_output` is called the returned transport/format is checked against the receivers `match_input` function, or the client's desired transport. When a candidate NCA passes both the input, output and operator checks, it can replace the logical operation(s).

Figure 3.5 walks through compiling the running example. Again, the example has three available NCA implementations (Figure 3.2): an NCA that implements scan with optional project and filter, an NCA that implements join with optional project, and an NCA that implements partial aggregation. Jumpgate walks over each node in the dataflow request, and tries to replace it with a matching NCA.

The first column of Figure 3.5 shows the original request, with the input data format (JSON) and the client's desired output format, shown as `NTF?` since the client doesn't know the specific NTF that will be returned. *Specific* NTFs are generated when NCAs replace operations, shown here as `NTF#`. At first the only NCA that matches is the JSON parser, because it is the only one that accepts JSON and generate NTFs. After parsers are inserted, NCAs that accept NTFs as input can match. The two right-most columns show Join and Aggregation matching. Jumpgate could replace Join and Aggregation in any order, but we show bottom-up ordering here. At the end, Jumpgate has computed the Staged Network Pipeline shown in Figure 3.2. Pseudo-code for this algorithm is supplied in Appendix A.1.5.

This iterative compilation algorithm per se is not a contribution of this work; other approaches could be taken here. Our contribution is to show how to use the interfaces to map a dataflow request to a graph of NCAs. There is a breadth of research in query plan optimization that could be applied, such as rules-based replacement instead of search. We chose iterative replacement because it produces many candidate graphs so Jumpgate can pick the best one based on predicted throughput, device availability, or some future metric. Currently, Jumpgate greedily picks the smallest graph so that operations are fused together as much as possible.

Generating Network Tuple Formats

NTFs remove the need to (de)serialize data during runtime, and NCAs and clients can *generate code* ahead of time to operate on them. NTFs address the issue of NCAs only being able to operate on a limited amount of data and not being able to deserialize complex formats. As they are inserted, NCA implementations generate a network tuple format (NTF) derived from the output schema of the operations they perform. 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. *NTFs are query-specific*. When a client submits a job, Jumpgate returns the NTF layout the client will receive, encoded as JSON, so the client can prepare its workers. Each NCA implementation configures itself to send and receive the given NTFs, via a call to `compile` on the life-cycle interface. While mapping the dataflow request to NCAs, if the NTF sent by a candidate NCA cannot be consumed by a receiving NCA, then the candidate will not be inserted. If NTFs seem straight-forward, that is the idea: they are easy to compute but remove complex (de)serialization from the data-path. We evaluate the benefit and burden of NTFs for both client and NCA implementations in §3.5.

Prior work has created NCA-specific data formats, relying on software-based converters to convert to and from this format, and they assume that there is only one NCA in use. For instance, Cheetah [175] has a format specific to their programmable switch implementation of filtering that only supports fixed-length values, omits a null vector, and must include the number of columns in each packet. Instead, NTFs are query-specific and implementation independent. NTFs can carry all data needed for the query to subsequent NCAs, and are designed to allow NCAs to pass tuples directly between one-another without conversion unless determined by the operation. Jumpgate checks NTF compatibility at compile time to ensure all NCAs can read their input.

Jumpgate generates NTFs with 8-byte-aligned fields for ease of configuring hardware devices. NCAs that have specific fixed output formats, such as our programmable switch aggregator, output their NTF specification directly. Figure 3.2 shows the generated NTFs for our running example. For Jumpgate's one-pass NTF generation algorithm see Appendix A.1.6.

Executing Staged Network Pipelines

After compilation, Jumpgate has a staged network pipeline ready for execution. Jumpgate executes staged network pipelines using the life-cycle interface of each NCA instance (§3.3.4). The SNP is set up by calling `compile`, `allocate`, `configure` on all NCA instances. Jumpgate proceeds to execute each stage, calling `execute` on NCAs that run in the given stage, and waiting for the NCAs to signal `done` before proceeding. The client receivers listen on sockets to receive data from the network. Execution errors are caught and reported to the client (i.e., Apache Spark) via the HTTP endpoint. Jumpgate relies on the client to retry the job.

In the running example (Figure 3.2), Jumpgate would first call execute on NCA1 and NCA3, to populate NCA3's hash-table. Once NCA1 and NCA3 signal "done", Jumpgate executes NCA2 and NCA4, and NCA3 again to execute its probe stage. During the second stage, Spark workers will receive data. At the end, NCA2-4 signal to Jumpgate that they are done. NCAs finish when they have completed parsing their files or their input connections are closed. When NCAs communicate via UDP, they send zero-length packets.

Summary In this section we explained our design contributions and illustrated how a dataflow request from an analytics system is compiled to an SNP and executed on NCAs. The interfaces, SNPs, and NTFs come together to automate previously manual processes to use NCAs.

3.4 Implementation

Jumpgate is designed to implement the compilation and orchestration mechanisms needed to use NCAs and get them to work together. Since Jumpgate is not on the data-path – i.e. it does not directly touch data moved between NCAs – we are free to implement Jumpgate in any language without imposing much run-time overheads. This design allows a implementation in a high-level language of ones choice. For instance, I implemented Jumpgate in about 5,500 LoC of Python. This chapter focuses on the novel contributions of Jumpgate, and does not detail all of Jumpgate's features. Jumpgate includes logic to measure operator throughput, scale instances of NCAs based on historical throughput (based on DS2 [91]), and utilities to help NCA implementations with code generation.

Other Clients – In addition to Spark, there are two other Jumpgate clients: a Python client to aid in testing Jumpgate without requiring Spark, and a preliminary Presto [169] client that can submit jobs and receive data from Jumpgate, but does not yet compute jobs from it's internal query representation.

NCA Implementations – We developed and integrated an aggregator NCA written in P4 for the Barefoot Network's Tofino switch ASIC to demonstrate that Jumpgate works with programmable switches, detailed further in §3.5.6. To emulate the operation of other accelerators, Jumpgate uses software-based NCAs written in C that can support all operations that Spark offloads when running TPC-DS (§3.5). Jumpgate uses simdjson [104] to parse JSON, and the ORC C++ Library to parse ORC [25]. Jumpgate deploys and executes software-NCAs on remote machines using SCP/SSH. Table 3.4 summarizes the NCA implementations, the operations they support, and the LoC to implement and integrate them into Jumpgate.

		Lines of Code		
Name	**Supported Ops**	**NCA**	**API**	**CG**
JSON	scan-[agg]	505	283	175
ORC	scan-[agg]	586	188	313
Join	join-[project]	766	343	266
Agg	[partial] aggregation	475	245	99
Shuffle	shuffle	321	146	62
PS-Agg	partial aggregation	700	186	-

Table 3.4: Jumpgate's current NCA implementations, the operations they support and the lines of code to implement: the NCA, the operator and life-cycle interfaces (API), per-query code generation (CG). PS-Agg denotes the programmable switch aggregator, the other NCAs are software-based. Square brackets denote optional operations the NCA can support. *Scan* includes read with optional filter and project.

Scaling NCAs – Jumpgate scales NCAs two ways: *internally* – to set threads counts for software NCAs – and *externally* – to create new instances of an NCA and link it into the existing SNP. Jumpgate determines how much to scale NCA using reported metrics and a technique adapted from DS2 [91]. DS2 computes the needed parallelism of all streaming operators in a graph to meet a target processing rate. The parallelism is derived using the each operator's throughput and the time each waits to send or receive data: too-slow operators are scaled up and too-fast operators are scaled down. Jumpgate uses DS2 to compute the parallelism needed to saturate storage bandwidth across the cluster, and then scales NCAs either internally or externally.

Currently, Jumpgate estimates an NCAs throughput by computing the mean throughput of all past instances. In the future, throughput could be more accurately estimated based on relevant factors: e.g., operation, size of each record, number of filtered columns, aggregation expressions. During our evaluation, we found that externally scaling parsers across machines was critical to take advantage of available storage throughput. But, internally scaling other NCAs was sufficient for our evaluation on small clusters. In the future, we plan to experiment with scaling operations on larger cluster sizes.

3.5 Evaluation

We evaluate the costs of using Jumpgate, including: overheads, ease of use, and factors affecting end-to-end performance. Prior work demonstrated that NCAs can be fast (§3.2), so it is not our goal to evaluate NCA implementations. **Instead, our goal is to validate our design decisions and explore the behaviour of pipelines of NCAs.** We show that: (1) using dataflow requests let analytics clients offload a significant number of operations with relative ease, (2) the overhead of Jumpgate is acceptable when processing high volumes of data, (3) Jumpgate can reduce data sent to client analytics systems by orders of magnitude, (4) and this can translate into improved query performance when NCAs can process data faster than the baseline analytics system.

3.5.1 Experimental Setup

Workloads We use the TPC-DS benchmark [136, 153], which consists of 99 parameterized SQL queries over simulated retail store data representing analytics queries used for decision-making. We use spark-sql-perf [47] to execute TPC-DS queries in Spark; spark-sql-perf breaks up the 99 canonical TPC-DS queries into 104 individual ones. When Spark uses Jumpgate, it transforms parts of these queries into Jumpgate jobs, described further in §3.5.2. Spark hits resources 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 ($\approx 370 GB$ uncompressed) and ORC format at SF=1000 ($\approx$ 387 GB compressed).

Hardware Setup We run Jumpgate on one machine, which receives jobs, compiles them, and then *sends and executes compiled binaries on other machines in the cluster.* Our Spark master runs on the same machine as Jumpgate, and Spark worker nodes run on the other machines in the cluster. Input data is stored on each machine. We explain our machine details in the relevant sections.

Metrics We measure query execution time using spark-sql-perf's built-in benchmarking, which measures the end-to-end time of each query. One way NCAs can improve analytics 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 (without comments) using `cloc` [14].

3.5.2 Client Integration: Apache Spark

Apache Spark 2.4.4 integration with Jumpgate was done in 2,200 LoC – a small amount compared to Spark's SQL modules, which comprise around 100,000 LoC. This result suggests that adding Jumpgate offload to client systems will not be onerous. Our changes can be broken down into four parts: offloading operators, execution coordination, receiving data, and the offload threshold heuristic. Our modifications were small, because Spark *already includes* code to manipulate expressions and query plans.

Query planning (1,100 LoC). Operations are offloaded from Spark's query planner [30] in a 'bottom-up' manner, starting at the table scan: supported operations are offloaded if they receive from an offloaded operator, until an unsupported operator is reached. *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, so the job request includes listening network endpoints of the workers.

Receiving Data (550 LoC). We added code to receive data over TCP and UDP sockets, as Spark workers read files (150 LoC). To receive data as fast as possible, we used Spark's code-generator to generate a parser for NTFs (400 LoC).

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

$$Overhead_{Jumpgate} + T/Throughput_{Jumpgate} \leq T/Throughput_{Spark}.$$

$Throughput_{Jumpgate}$ and $Throughput_{Spark}$ are determined experimentally. T is best computed *per-core*, so that as Spark is given more cores, it will use them when needed. Spark's current value of T is $700MB$ to offset a worst-case ≈ 6 seconds of overhead – a precise value is not too important because nearby values

	Operations allocated to:		
Offload	**Spark**	**Jumpgate**	**Total**
None	5,135	0	5,135
F	4,528	1,322	5,850
F,P	3,928	1,999	5,927
F,P,A	3,908	2,038	5,946
F,P,J	2,322	2,982	5,304
F,P,J,S,A	1,852	3,299	5,151

Table 3.5: Number of operations executed by Spark and Jumpgate when Spark offloads different combinations of: **F**ilter, **P**roject, **J**oin, **S**huffle, and **A**ggregate to Jumpgate. The total is not always equal to the original total because Spark performs redundancy eliminations; offloading removes some re-use opportunities.

will produce similar runtime for both systems. The heuristic offloads operations when the offloaded job would process at least one dataset of at least $T \times numCores$ in size.

3.5.3 TPC-DS Study

We now look at characteristics of operations Spark is able to offload to Jumpgate.

Total Offloaded Operations To measure offloaded operations, we ran TPC-DS queries on Spark and observed the jobs it sends to Jumpgate. Table 3.5 summarizes the split of operations between Spark and Jumpgate as we vary operation types to offload. With all supported operations offloaded and the offload threshold disabled, Spark generates 3299 operations ($\approx 60\%$ of total Spark operations) across 853 jobs – the offload strategy can produce multiple jobs per query. This highlights that the client interface can help to offload a significant amount of operations for acceleration using NCAs.

Properties of NTFs: NCAs like programmable switches are limited in the amount of data they can read from each packet [37, 166, 171], so we look at the size of the network tuple formats sent to each operation. Fixed length NTFs make

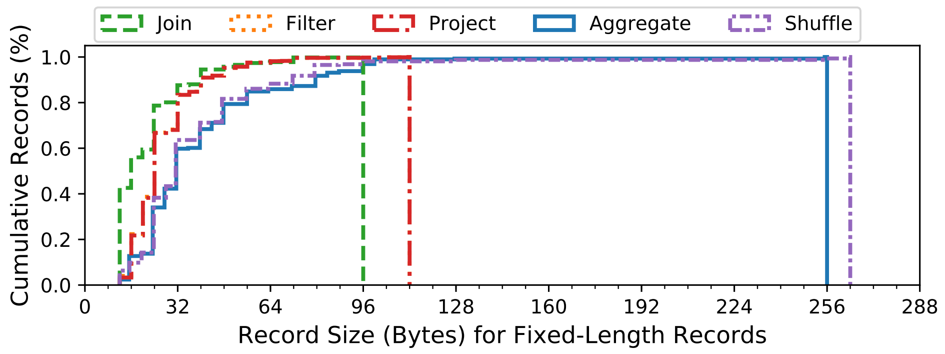


Figure 3.6: Cumulative distribution of the size of fixed-length network tuple format for operations offloaded from TPC-DS. Filter and Project have the same distribution and overlap.

up 60% of all NTFs (i.e., these NTFs have no strings). Figure 3.6 shows the cumulative size for fixed length NTFs. 98% of fixed length NTFs are under 96 bytes. *This indicates that NTFs make it feasible to use programmable switches to process many queries.* NTFs with variable-length sections make up 40% of all NTFs and the fixed length portion follows a similar distribution. This mix of fixed-length and string data motivates developing NCAs that can operate on both fixed-length *and* variable-length data. Further breakdowns of the field types that each operation references and the number of strings in each NTF are given in Appendix A.1.7.

3.5.4 Jumpgate Overhead

This section looks at the setup and dynamic overheads of Jumpgate. We find Jumpgate has (1) high set-up overhead due to its compile-everything approach, but (2) low dynamic overheads during execution, so it will not get in the way of high throughput operators.

To measure dynamic overheads, we configure Spark to offload all eligible operations from TPC-DS to Jumpgate (disabling offload threshold) 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

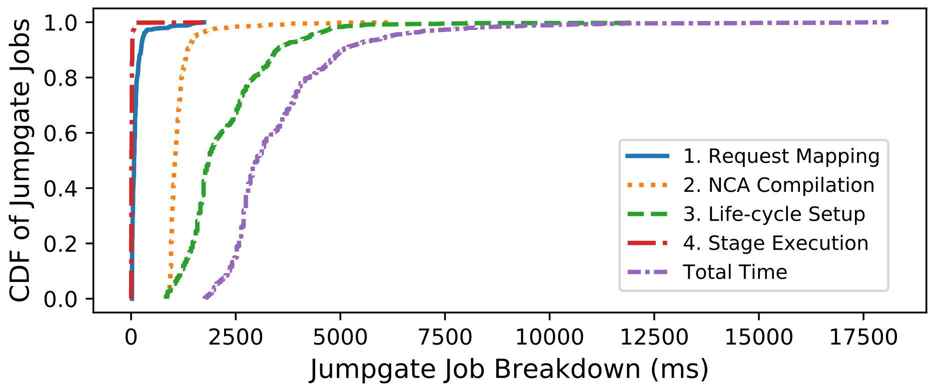


Figure 3.7: CDF of breakdowns of static and dynamic overheads in Jumpgate. Execution time is broken into four parts. The first three are static overheads. (1) Request Mapping, the time to map a request to staged network pipeline. (2) NCA compilation, the time for the NCAs in a job to generate query code. (3) Life-cycle Setup, the time to upload and start binaries on worker machines and to call allocate and configure to get NCA endpoints. (4) Stage Execution, the time to actually run the query, but since practically no data is processed for this test, it is dominated by the overhead of Jumpgate calling `execute` and receiving done callbacks.

machine to run Jumpgate, and deploy compiled NCAs to 4 machines.

Figure 3.7 shows a CDF breakdown of execution time for all 1205 offloaded jobs in this setup. Spark without Jumpgate takes 11-950ms for the same test. Overall, 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, when each NCA is compiled in parallel. Life-cycle Setup takes 1.5-5.8s, because of using SSH to transfer binaries and start processes 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 shows that *Jumpgate can get out of the way during execution, so it is possible to benefit from high performance NCAs.* We discuss how to reduce static overheads in §3.6, but even if static overheads are not reduced, the heuristic we added to Spark in §3.5.2 ensures static overheads are amortized.

3.5.5 Performance: Understanding NCA Behaviour on Real Queries

This section explores how Jumpgate and NCAs behave when executing queries and illustrates the main factors and bottlenecks that affect end-to-end performance of queries. Here, we use software NCAs to explore this question because we lack hardware implementations that can support every operation offloaded from TPC-DS.

Experimental Setup. For this evaluation, we ran experiments on Microsoft Azure LSv2 virtual machines as they provide NVMe drives to alleviate a bottleneck on input data. In these tests, the Jumpgate service and Spark master are run on one node, and Spark workers and software NCAs are deployed to other nodes in the cluster. This means that Spark and Jumpgate use the same compute resources. When Jumpgate runs out of nodes to allocate a Software NCA, it over-subscribes an existing node. This biases the findings towards Spark, as future Jumpgate deployments would use accelerators in addition to the existing cluster. One configuration uses four LSv2 nodes, with 8 cores each, that provides about 3.2Gbps inter-machine bandwidth. However, we found inter-VM networking bandwidth to be a bottleneck, and there was no way to allocate higher bandwidth network on the VM without also increasing core counts per node. Instead, we test on a configuration that uses a single 32 core LSv2 which could achieve 40 Gbps via loopback networking, and we restricted Spark nodes and software NCAs to use 8 cores to emulate the first configuration. At this time, no major clouds providers offer a combination of high bandwidth and low core counts, so this is the best we could do across cloud set-ups.

Understanding Jumpgate's execution model, bottlenecks, and potential. Figure 3.8 shows a timeline view of Jumpgate executing TPC-DS query 3. Q3 with and without Jumpgate has similar performance. To explain why, we first note that NCAs operate as a parallel pipeline, forwarding data to subsequent NCAs and forwarding to Spark. The fact that staged network pipelines are parallel pipelines means that we can draw upon prior work on understanding and modelling them. For instance, a pipeline's throughput is limited by the throughput of its slowest component [71, 91, 107]. Here, we see that since orc-2 spends most of its time processing (orange fraction) and all other NCAs spend most of their time waiting

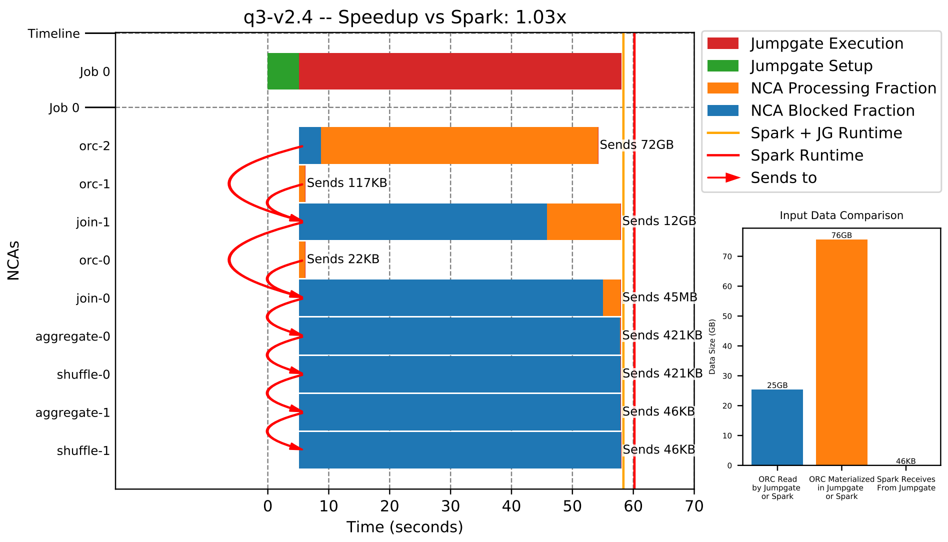


Figure 3.8: Visualization of the timeline of Jumpgate's execution of TPC-DS Query 3. Time goes from left to right. The top Timeline bars break down Jumpgate's set-up and execution time for the job, and below this are the NCAs used to execute the job. Red arrows show which NCAs send to one another. The NCA with no outgoing arrows (shuffle-1), is the NCA that transmits data to Spark. On each NCA bar, we show the fraction of time it spent processing data (orange) and the time it spent waiting to read or send data on the network (blue). The red vertical line shows the time that Spark took *without Jumpgate*, and the orange vertical line shows the time that Spark takes when *using Jumpgate*. Speed-up or slow-down over Spark is shown at the top. The inset bar chart at the bottom-right shows the data read from storage, materialized in memory, and transmitted back to Spark for this query.

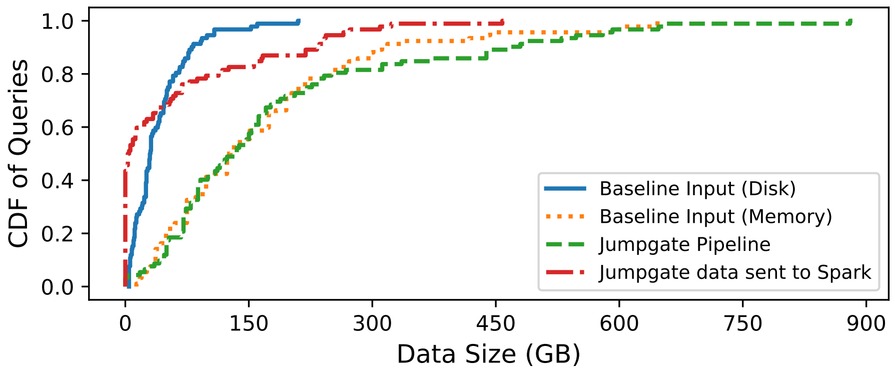
for data (blue fraction), that the ORC NCA was the bottleneck. The reason for the ORC NCA's performance is that while Jumpgate's ORC parser uses the C++ implementation of the Apache ORC parser, and so is almost twice as fast as the Java implementation used by Spark, the extra work of converting ORC parsed data to NTF data offsets this advantage.

Figure 3.8 also shows us the data sent by each NCA and read from storage. Here, we see that as data moves from the ORC NCAs towards spark (i.e. towards the bottom), the data sent decreases dramatically. The inset bar chart compares the data volumes read by this query and received by Spark. The ORC parser reads 25GB, which turns into 76GB after data is decompressed and materialized in memory. This volume of data would normally have to 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.

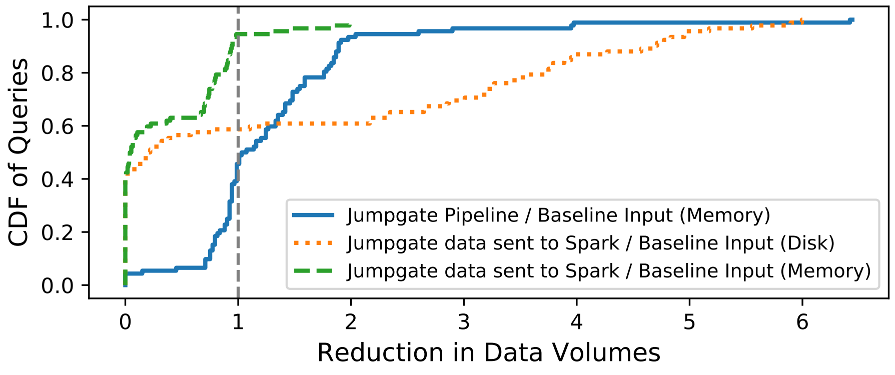
This example shows there are two factors at play in understanding when offloading will be successful: **(Factor 1)** *offloading will be successful when work is reduced for the client system.* In this case, this is reflected in how the volume of data transmitted to Spark is reduced significantly by "summative" or filtering properties that are inherent in filters, joins, and aggregations operations. But, **(Factor 2)** *benefiting from this reduction depends on the underlying NCA pipeline being able to operate on a large volume of data more quickly than the client system.* In this example, the software-based ORC parser was not quite quick enough, so only a small speed-up was seen. With a faster ORC to NTF parser, we would expect this query to see more speed-up. Overall, the first factor can show us the *potential* for offloading, while the second factor tells us if speed-up can be achieved *in practice*. Now, we zoom out to look at these factors for all of our studied queries.

Factor 1: The potential for data reduction in TPC-DS Figure 3.9a shows the CDF of data volume moved in all TPC-DS/ORC queries. *Baseline Input (Disk)* and *Baseline Input (Memory)* show how much data Spark reads from disk and materializes in memory, and reflect the first two bars on the inset chart in figure 3.8 for all queries. When offloading, Jumpgate would instead read/materialize *Baseline Input* data, and Spark would only receive and process *Jumpgate data sent to Spark*, but NCAs need to process data in *Jumpgate pipeline*.

Figure 3.9b computes the ratio of data volume change per-query: Comparing data received to compressed ORC data (orange dotted line) 60% of all queries have reduction in data received by Spark compared to compressed ORC data. 50% see a reduction greater than $4\times$, and 25% see a reduction over $2700\times$. Comparing data recevied to ORC materialized in memory (green dashed line), **94% of all queries see a reduction in data received by Spark** and 50% of queries see a reduction greater than $22\times$. *Reduction in data read and materialized by Spark translates*



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

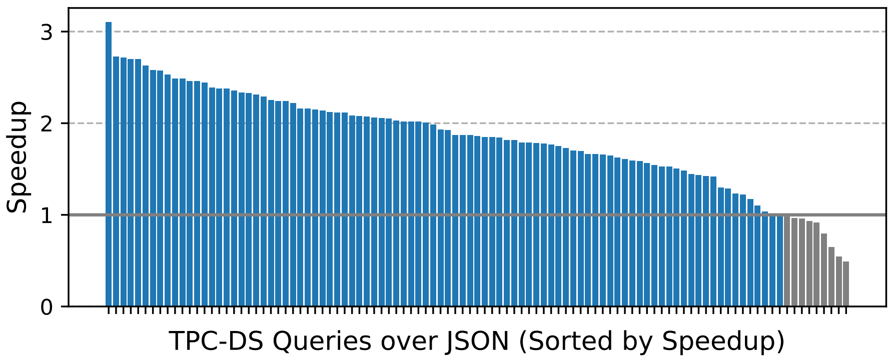


(b) CDF of data reduction of NTFs sent to Spark over reading ORC from disk or materialized in memory. **More queries lower than 1 on the x-axis is better.**

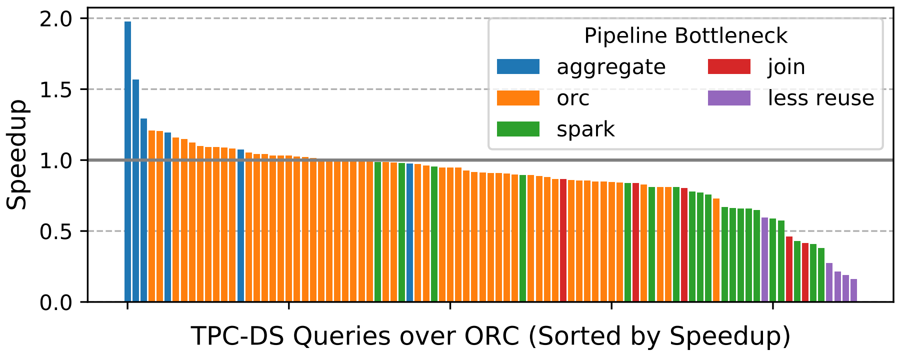
Figure 3.9: Data volume sent and received by Spark with and without Jumpgate.

into less work for Spark to do. Lastly, data volume of Jumpgate's NCA pipeline is within $1-2$x of the data materialized in memory (blue sold line), with inflation due to joins matching multiple tuples.

Offloading tasks to Jumpgate has potential to be a win when data received and processed by client can be reduced by orders of magnitude, and these results show that can happen frequently in TPC-DS, even with a limited set of database operations offloaded. These results also validate our decision to offload operations



(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 3.8). Higher is better.

Figure 3.10: Current performance of Software NCAs

'bottom-up' from storage, because we are able to capture a significant amount of work to do from the client system. The cost of this reduction is that NCAs must process this volume of data, and receive and send it over the network quickly. Figure 3.9a shows the overall volume of data materialized in memory is on-par with what would get written to the network when using NCAs. This gives a convenient rule of thumb for understanding the demands on the network if NCAs are in use.

Factor 2: The current performance when Spark uses Jumpgate for TPC-DS Figure 3.10a shows the speed-up of offloading TPC-DS queries over JSON to Jumpgate, using the 4-node experimental setup. We see that offloading to Jumpgate improves performance by $1.16\times - 3.1\times$ for 88 of the 104 queries, with the mean improvement of $1.8\times$. This is because Jumpgate's JSON parser is faster than Spark's due to Jumpgate's use of simdjson [104] over Spark's Java-based Jackson parser. Nonetheless, the pipeline remains bottlenecked on JSON parsing because it is generally quite slow.

Figure 3.10b shows the TPC-DS queries over the ORC format using the single node, 32 core experimental setup. ORC is optimized for analytics performance, and makes Spark more competitive with Jumpgate. Here, we used each query's timeline chart (see Figure 3.8) to determine the bottleneck of each query. Overall we see that there is less speed-up than with JSON, and many queries are bottlenecked on the ORC parser, as described earlier. Where we do see good speed-ups is on certain aggregations. These aggregations have many unique keys which causes many memory allocations, and the aggregate NCA uses a faster memory allocator (tcmalloc [63]) than Spark.

We see significant slowdowns attributed to Spark because in these queries Spark offloads only scan and filter operations and so the pipeline becomes bottlenecked on 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.* When we investigated the query plans, we found this was due to not offloading sort-merge joins from Spark which often preceded an aggregation which would have reduced data received by the client. The worst slowdowns are due to the Spark's current offload heuristics, which remove some opportunities for re-use from the query plan. In the future, Spark's offload heuristics should be revised to offload all joins, and to generally avoid offloading operations that won't bring an expected reduction in data size.

One thing we can learn from these experiments is that format parsing will be important to accelerate in the future. We found ORC parsing and Spark's NTF format parsing were frequent bottlenecks. There are a variety of approaches to improve format parsing, such as: changing the format altogether to eliminate parsing overheads, implementing format parsing accelerators in hardware, and optimizing the format parser in software.

Performance goals for future accelerators. Finally, to estimate how quickly future hardware accelerators would need 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 we assume all slowdown can be attributed to NCA processing speed, and not to Spark's NTF receive path. We found 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 (§3.5.6), 40 Gbps SmartNICs, and Barefoot Network's Tofino ASIC (6.4 Tbps). This is a promising result for the feasibility of future NCAs development.

3.5.6 Performance: Programmable Dataplane Switches

We now look at how a programmable dataplane switch can be integrated into Jumpgate, show that Jumpgate enables it to cooperate with software-based NCAs, and show how it can be used to accelerate queries.

Integration. We implemented a partial aggregation operator that operates on data in network tuple format received over UDP on a programmable dataplane switch in P4 [38]. This operator performs a partial group-by operation over 64-bit values, counting each unique occurrence, and producing a 32-bit or 64-bit summation. The aggregator does not support floating point arithmetic, a known limitation [167]. This limitation unfortunately means it cannot be used for TPC-DS queries as all TPC-DS aggregations work on floating point numbers.

The aggregator maintains a 64K entry hashtable using on-chip SRAM, and relies on an upstream software-aggregator to perform the final aggregation. In case of a hash collision, the operator forwards packets to the full aggregator. Our P4 implementation is $\approx$700 LoC. **We integrated this operator into Jumpgate using only 186 LoC,** including detecting applicable operations and modifying control plane rules in the switch.

Performance. *Experimental Setup –* Since no cloud provider offers programmable switches, we deploy this set-up in our lab. We deployed the partial aggregator using a 6.4Tbps switch equipped with the Barefoot Networks Tofino ASIC [32]. We use dual NUMA node, 2.4 GHz Intel Xeon E5-2407 v2 servers with 8 CPU cores, connected to the switch with Intel XL710 40GbE NICs. Since TPC-DS queries only aggregate decimal values and cannot use this NCA, this experiment uses a custom query: an aggregation over TPC-DS store_sales at SF=100 that counts unique items and sums up sale price as a 32-bit integer. Dataplane programmable switches *always* operate at network line-rate, and so the aggregator can run at the speed of the switch hardware: 6.4 Tbps. End-to-end performance is determined by *how fast data can be sent to and received from the aggregator.* To illustrate, we ran two versions of this experiment:

Test #1. Integrated with Jumpgate – We submit the job to Jumpgate using our Python client. Jumpgate automatically generates two software NCAs: one to parse ORC and send NTFs towards the switch and another to receive NTF data from the switch and perform a final aggregation. Jumpgate uses the lifecycle interfaces to direct the switch to send data to the the IP/port of final aggregator. Partial aggregation reduces the output data volume by 43× to 2.29% of the original input stream. But, the ORC parser sends each small packet via the `send` syscall, resulting in low throughput (0.288Gbps) and no runtime improvement. This result echos our findings in the prior section, the potential data reduction is high, but the limitations of the ORC parsing NCA prevent us from seeing speed-up.

Test #2. Preliminary DPDK Versions – To remove network and storage format bottlenecks, prior work uses DPDK to bypass the kernel to improve UDP send-rates [111, 175]. DPDK is not yet fully integrated into Jumpgate, so we manually updated the Jumpgate-generated network code for the ORC parser and aggregator to use DPDK. We compare the software version of the NCA, where the parser and aggregator run in software with the version where the aggregation is done on the switch. In the software version, the sender transmits pre-recorded NTF packets at up to 27Gbps, but the receiver aggregates at 12Gbps, limiting the sending rate of the sender. The test completes in 1.5*s*. Using the switch to aggregate allows the sender to run faster, completing in 0.8 seconds (a throughput of 27Gbps) for a speedup of 1.8× over the software-only DPDK case.

Putting these results in the framework from the previous section reinforces how these two factors interact: Test #1 significantly reduced data sent to the client (factor 1) but data could not be sent fast enough to see a performance difference. Test #2 improved the send and processing rate (factor 2) so we could see a runtime benefit from reducing data received by the client.

Overall, this test shows that Jumpgate can successfully use programmable switches, but highlights the limitations of current hardware and the need to accelerate input data parsing to achieve performance improvements discussed in the prior section. Ideally, future aggregation NCAs would be able to handle floating point aggregations over integer and string data types so they could execute real queries.

3.6 Limitations and Discussion

Applicability of TPC-DS and Spark's offloading heuristics. We present results for TPC-DS, a standard analytics benchmark, but it only represents a particular industry, and not all users of SQL. Not all queries will be able to be run on all NCAs. We show how to characterize workloads to determine potential benefits from using NCAs. We present results of TPC-DS as offloaded by our current Spark heuristics. In the future, we could decide to offload more operations, such as sort-merge-joins.

Focus on Apache Spark. The above sections focused on how Apache Spark can use Jumpgate. However, Spark is not the fastest system at executing SQL queries [52]. We chose Spark because of its popularity which is attributed to its ability to mix general purpose computation with SQL queries and machine learning operations. When Spark uses Jumpgate, users can continue to benefit from Spark's flexibility while analytics operations are offloaded to NCAs. Our evaluation shows that Jumpgate can reduce data returned to *any* analytics system. Under the right conditions – with NCAs that are faster than the analytics systems, and fast enough software networking – other analytics systems could benefit from offloading to Jumpgate.

We believe other analytics systems could integrate just as well with Jumpgate. Presto [169] is an analytics system designed by Facebook for large scale data processing. We are working to integrate Presto with Jumpgate: Presto can currently submit hand-coded jobs and receive data, but does not yet automatically compute Jumpgate jobs. Presto has logical query plans similar to Spark and includes an extensible query optimizer, so work is ongoing to have Presto create jobs automatically.

Jumpgate's static overheads. *Reducing Compilation Time:* Software NCAs are compiled from scratch per-query to reduce runtime overhead. One way to reduce compilation time is to pre-compile most code and link in query-specific code produced by generating byte-code: HyPer [140] reduces compilation time more than $10\times$ using this technique. *Reducing Life-cycle Setup:* Jumpgate deploys and executes software NCAs with SSH. In the future we plan to take advantage of existing low-latency job execution systems.

Dependency on storage and network performance. Jumpgate relies on high storage and network data rates, and operators must ensure that storage and networking resources are adequate to support high NCA performance. Jumpgate's data reduction means that client systems do not need to have high bandwidth.

Software NCA Limitations. Our software NCA implementations are simple, but help to evaluate integration and to study behaviour of NCAs. Operator fusion [130] is a well known idea that can further improve Jumpgate's performance. A production system based on Jumpgate will need to carefully trade off fusing operations in software NCAs while utilizing hardware NCAs.

Network Transport Limitations. When using a programmable switch, NCAs send UDP packets. Prior work shows how to address UDP unreliability in this setting [85, 167, 175, 195]. Another limitation is the low performance of sending small packets, which can be improved with kernel-bypass or in-kernel networking (e.g., DPDK [50], XDP/eBPF [35, 74], Netmap [121]) or by developing NCAs that can operate on larger packets that can be efficiently sent from user-space.

3.7 Opportunities for Future Work

Jumpgate contributes a framework that enables client systems to use NCAs and allows NCAs to work together. These foundational capabilities open up a variety of potential future work. We detail some potential future research opportunities in this area that Jumpgate can support.

Multi-device NCAs. Jumpgate does not restrict the implementation of NCAs aside from requiring input and output to be directed toward network endpoints. We see an opportunity for developing NCAs that mix software and hardware devices at lower levels, while still communicating with other NCAs as if it was a single NCA. For instance, a multi-device Join NCA might perform the build phase using software to generate index structures (e.g., Bloom filters, B-Trees) and then deploy these structures to hardware accelerators for the probe phase, which is often much larger. This behaviour could be done transparently to Jumpgate while still presenting as a single NCA.

Using Jumpgate to Explore Memory Grants for Disaggregated Systems. Jumpgate makes very little assumptions about the transports used by NCAs. Jumpgate treats the transport protocols and connection endpoint information used by NCAs as opaque strings. This allows one to use Jumpgate to experiment with and develop new alternative data transport protocols while using Jumpgate and its clients to drive analytics workloads. Disaggregated data centre designs that decouple memory from compute have recently been proposed [18, 170, 192, 193]. Some of this work has discussed the possibility of a Rack-level MMU that could transfer remote page ownership between devices, called memory grants [18, 192]. NCAs that exchange data via memory grants could be added to Jumpgate, and Jumpgate could generate code and orchestrate execution as was shown above. This proposal is compelling for two reasons: First, the proposed memory grant protocol exchanges memory pages via RDMA, which could could help reduce the overheads of software sending data on the network. Second, the access patterns of inter-NCA communication are linear, so they could be a good match for disaggregated memory which suffers from high latency on un-prefetchable requests [193].

Low-latency Allocation and Scheduling. Jumpgate needs to reserve and allocate NCAs, and ideally they are allocated with respect to how they will communicate, localizing communication, likely to a single rack when possible. Choosing the best set of NCAs from a larger pool to optimize communication will be an important issue. The challenge is that Jumpgate needs to reserve and allocate devices quickly because it is on the critical path for query processing.

Job Merging and Cooperative Scanning. We noticed that many queries often produced several Jumpgate jobs with scans against the same table with different filter or aggregation criteria. But, Spark's current Jumpgate offload strategy submits one job at a time. Prior work shows that feeding several queries with a single scan can be beneficial, but requires complex I/O scheduling [197]. However, because NCAs are connected via the network, we see an opportunity to simply broadcast the output of a scan to multiple NCAs. We envision experimenting with submitting *all* jobs for a query simultaneously, to give Jumpgate an opportunity to merge SNPs together and reduce the number of scans done against storage.

Re-writing queries for better applicability to NCAs. Jumpgate could re-write jobs to be more appropriate for a given NCA. For instance, to enable the P4-based aggregation to work over string keys, Jumpgate could re-write the job to dictionary-encode each string, then aggregate over the dictionary ID column, and then join the output against the dictionary. ACCORDA [57] showed how encoding data into an intermediate form can make it much more appropriate for hardware accelerators. In addition, clients could also potentially re-write their queries to be more applicable, such as referencing alternate columns where possible.

Hardware acceleration for file formats. Jumpgate, like many systems, relies on being able to read data quickly from storage, and we saw that performance can suffer if data cannot be put on the network quickly enough. We also see an opportunity to evaluate and improve how quickly analytics file formats can be read and written using hardware accelerators or simplified software. For instance, Albis [178] showed how compressed formats like ORC take significant software overhead to decompress and thus designed a simpler format that sacrificed more storage space for faster software parsing.

Serverless Analytics Performing analytics on serverless platforms is attractive because they offer vast computational resources on demand. However, serverless analytics tasks communicate by writing intermediate data to storage, which can limit scalability [97]. There is an opportunity to adapt techniques presented here to serverless settings. Jumpgate's software NCAs are good candidates for running on serverless platforms because they generate native code that has few outside dependencies and only maintain state while they run. The Staged Networked Pipeline execution model provides a way for analytics tasks to communicate directly without the need for intermediate storage, provided a serverless platform supports listening sockets and can guarantee concurrent execution. Recent work has lamented how current serverless platforms provide little in the way of performance isolation [182], and a platform for allocating hardware NCAs could help.

3.8 Conclusion

This chapter presented the design and evaluation of Jumpgate, a system that solves the extensive integration problem in using NCAs for data analytics. Jumpgate's interfaces enable integration of existing clients and NCAs. Spark integrates Jumpgate in 2,200 lines of code and Jumpgate accelerates Spark by 1.12-3$\times$.

3.8.1 War Stories

I want to conclude this section by discussing overall lessons learned while working on Jumpgate, and specifically what changed between Jumpgate as proposed in Chapter 2 and Jumpgate as developed in Chapter 3.

Jumpgate began as an ambitious project to deliver on the potential for network connected acceleration, inspired by earlier work that promised large speed-ups. My early plans for the system included plans for such features as: (1) A compiler that could target several different accelerators and automatically generate code for each system without a lot of engineering work. (2) Topology aware scheduling of NCAs that could optimize for throughput and data locality, perhaps through the use of an SMT solvers. (3) Addressing the limited read window of programmable dataplane switches by using a constraint solver to ensure that all required data fields of a packet fell within the needed window. (4) Evaluating and developing new transport protocols that could enable better interoperation with programmable switches and software.

The first goal had to be reconsidered as access to hardware became a challenge, and it became clear that each device could have very unique capabilities and would require very device-specific ways of deploying the operations. Additionally, the semantics of SQL do not map trivially to other languages. Special-case code had to be produced for C, for example, to handle null values. Thus, the operator and life-cycle interfaces were born to help abstract away this complexity and to allow me to search for coauthors who could implement these separate devices. This trade-off actually paid off, because it allowed me to split the work for developing

NCAs between several people. I am grateful for Niloo Gharavi who helped on the JSON parser, Joel Ahn who helped on the ORC parser, and Swati Goswami who developed the aggregator for the programmable switch.

The second goal had to be reconsidered as having limited access to accelerator hardware made the practice topology issues relatively straight-forward to address – there simply wasn't a big enough problem to solve yet. I also made an early decision to connect NCAs using the IP layer, so that I could test many NCAs locally on a single machine, and also deploy NCAs in any environment without worrying about low level network connectivity. The topology allocation question remains open, as described in future work.

The third goal was eventually dropped as I evaluated the first outputs from NTF on TPC-DS. I discovered that most NTFs came far under the packet read window on the Barefooot Tofino switch, so this was not a problem as I anticipated.

The fourth goal was cut short for lack of time. The programmable switch tests in this paper use raw UDP, and other work has introduced simple reliability mechanisms on top of this to address the rare dropped packet [167, 175, 195]. I believe that such transports will still be important in the future and are worthwhile to look into, especially as network connected accelerators begin to gain traction and show further promise.

Chapter 4

Practical Cross Program Memoization with KeyChain

Previous chapters presented techniques for accelerating data analytics that compiled a client request to staged networked pipelines of NCAs. This section presents another compiler-assisted technique for accelerating analytics that re-uses previously computed results.

Cross program memoization (CPM) reduces resource utilization and improves response times by enabling data processing systems to re-use previously computed results between programs. An under-explored requirement to implementing CPM in general purpose data processing systems like Apache Spark is computing identifiers for results of user-defined functions that are valid between programs while avoiding degrading system performance when sharing is not possible. In this chapter we describe and evaluate a technique, called KeyChain, that computes keys for intermediate and final results of programs with user-defined functions. We use KeyChain to implement CPM in Apache Spark, and show that KeyChain's simple design means it can be easily added to relevant systems, incurs low runtime overheads, and enables heuristic detection of equivalent programs so that CPM can be added to more systems and useful results can be more widely re-used.

This chapter was accepted for publication at the 2018 IEEE International Conference on Big Data, and was awarded best paper [134].

4.1 Introduction

Data intensive programs can be long running and expensive to execute, so it can be quite beneficial to share previously computed intermediate and final results to speed up future computations. For example, if Alice and Bobbie are exploring the same data-set on the same infrastructure and Alice performs a join query that takes a long time, it would be ideal if the results of Alice's query are re-used when Bobbie runs the query. Similarly, if programs written by Alice and Bobbie perform the same pre-processing steps, they should also share their results. In general, it is desirable to share many previously computed results as widely as possible to improve performance and reduce resource usage. Since users of these systems may not even know if and when sharing can occur, the job of exploiting sharing opportunities falls to the execution system. **Cross Program Memoization (CPM) is our name for optimizations that enable programs run on shared infrastructure to re-use computation performed by other programs.**

In distributed data processing systems, cross program memoization (CPM) can yield significant gains. Nectar [68] reports a reduction in cluster execution time of up to 50% using CPM, SQLShare [83] estimates that 37% of their system's total execution time could be saved and BigSubs [87] realizes up to 40% machine time reduction on production clusters at Microsoft. Despite these successes, a core part of CPM implementations has remained under-explored in the literature: *low-overhead techniques to compute identifiers (keys) for the results of final and intermediate computations with user-defined functions (UDFs).* Such keys are needed to search for and look up results in a cache and to track metadata about previous computations, such as if a result *should* be cached next time it is computed.

Low-overhead key computation and CPM implementations are important because the potential to share results can vary widely between deployments. Ren et al [160] study a few deployments shared by researchers and find that different users share less than 1% of files, probably because each user was working on separate projects. Nectar's evaluation of automatic caching in production clusters shows some deployments experience less than 10% reduction in runtime, but others experience up to 50%. SQLShare [83] ran a long-term study on usage patterns of SQL-as-a-service and found that CPM could reduce total execution time by 37%, but most queries could either significantly benefit from sharing with a 90% reduction in execution time, or benefit very little, less than 10%.

Deployment-dependent variation in sharing potential presents an awkward choice to either require every deployment to somehow analyze benefits before enabling CPM, or leave it disabled and potentially miss out on beneficial opportunities. To break out of this dichotomy, **we need key computation techniques and, more generally, CPM implementations that have low overheads so that sharing can be enabled by default** in order to put *all* deployments in a position to exploit data sharing when it occurs. Despite this need, Nectar did not report overhead of their key generation or CPM implementation and Incoop [36], an incremental processing system, can increase an application's running time by 5-22%, a high price when no benefit is to be had[1].

Data processing systems like Apache Spark support UDFs so that users can perform arbitrary computations on data (§4.2.1). To increase sharing potential, a system would ideally detect if UDFs used in different programs are equivalent[2] so that it could more widely re-use results produced by UDFs. Systems that execute SQL are able to do this by analyzing query syntax [103, 108, 109, 164]. In contrast, UDFs are often written in the same general purpose language as the execution layer, so it is not immediately clear how the execution layer can detect UDF equivalence without implementing its own parser and compiler.

4.1.1 Contributions

This chapter describes and evaluates a technique, called KeyChain, that computes keys for intermediate and final results of programs. We show this core component of CPM can be easily added to an existing system, incurs low runtime overheads, and enables heuristic detection of equivalent programs so data can be widely reused. Figure 4.1 shows how KeyChain sits within a generic execution layer. KeyChain generates string-based keys that represent the intermediate and final results of data processing programs. Keys are valid across programs, so that different

[1]Incremental processing systems are usually applied on a per-application basis when the utility is known a-priori, so they are more tolerant to overheads on the initial execution.

[2]Since program equivalence is undecidable [177], the system would take a heuristic approach to keep overheads low.

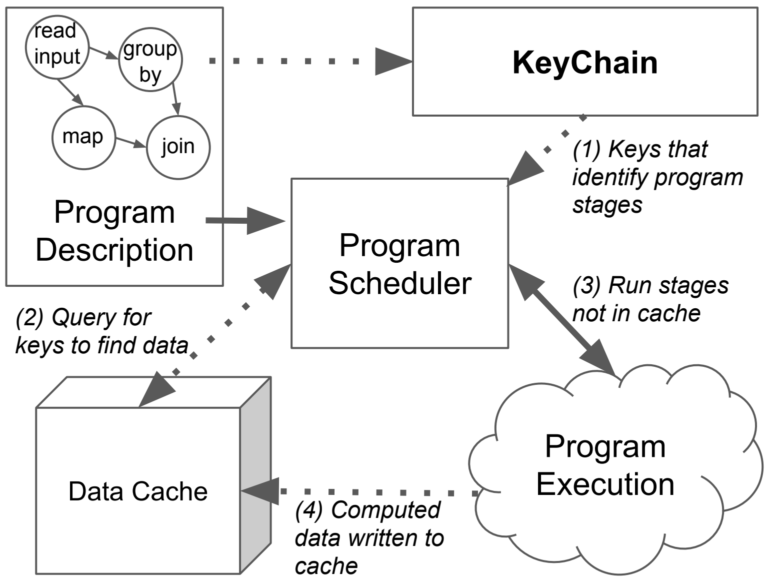

read input
group by
map
join
Program Description
KeyChain
(1) Keys that identify program stages
Program Scheduler
(2) Query for keys to find data
(3) Run stages not in cache
Data Cache
Program Execution
(4) Computed data written to cache

Figure 4.1: Example execution layer augmented with KeyChain. Dotted lines show the generation and use of KeyChain keys in the system.

programs can re-use other program's results. KeyChain generates keys *before* the program is run by hashing together representations of input metadata, communication patterns, and the UDF bytecode. The execution layer of the data processing system can use these keys to find and store re-usable data in any storage system.

KeyChain addresses the challenge of computing a key that identifies the results of computations. We use KeyChain to implement a simple CPM in Apache Spark that enables sharing of previously cached results. We do not focus on determining *which* data to cache to achieve the best cost-benefit trade-offs as this important aspect has been covered extensively by prior work in memoization and automatic materialized view selection [17, 68, 87, 103, 108, 109, 164, 188] (§4.6). KeyChain compliments this prior work by enabling systems to find more reusable data.

By presenting an evaluation of the feasibility and simplicity of KeyChain, we hope to encourage the wider adoption of CPM so that systems can share more data, reduce operating costs, and improve user experience. To this end, we make the following novel contributions:

Simple Design. KeyChain is designed (§4.3) to be simple but effective so that CPM can be easily added to existing data processing systems that implement the DAG computational model (§4.2.1). We added KeyChain to Apache Spark's caching layer (§4.2.2) in under 2100 lines (§4.4) that enables CPM for its existing in-memory cache (§4.5.2). The UDF hashing library and Spark implementation are open-source so that others can re-use and extend our work.

Low Overhead. KeyChain uses low overhead techniques to compute keys and the overheads grow with program size, *not* the data-set size. For example, prior work [36, 68] uses a hash of the input data as the input representation, requiring significant I/O costs to compute a key. Instead, KeyChain shows how to use input meta-data (e.g. a filename or query string) while still ensuring the computed keys reflect the most up-to-date input data. Our tests show KeyChain hashes UDFs in under 350ms (avg. 2ms) and our evaluation on TPC-DS [136, 153] shows no significant impact on query times when no sharing occurs. Low overhead means CPM can always be enabled because we pay almost nothing when sharing does not occur, but benefit significantly when sharing does occur (§4.5.4).

Evaluation of compiler-assisted UDF equivalence. Prior work [76, 95, 150] has shown it is possible to leverage existing compilers to heuristically detect functionally equivalent programs that are syntactically different. KeyChain's UDF hashing (§4.3.2) shows that we can use compiler-assisted equivalence detection to share data between some functionally equivalent UDFs, even if their syntax differs. To measure how well this effect works in Spark and motivate its use in other systems, we contribute a new benchmark suite to evaluate this property, called syntactic resilience, for different compilers (§4.5.3).

4.2 Background and Motivation

4.2.1 DAG Computation Model

Many popular distributed data processing systems represent computations as a directed acyclic graph (DAG), such as MapReduce [48] and Apache Spark [191]. In a DAG, each node represents an operation that produces data, and an edge from node x to y means that the output of x is used as input to y. DAGs enable an execution layer to understand data dependencies, which aids fault tolerance in the event of data loss, as well enabling optimization and scheduling decisions to be made before the program is executed.

For KeyChain, there are two important classes of DAG nodes: Input nodes and Transformation Nodes. *Input nodes* represent data read from some outside source, such as a network filesystem, database, or even randomly generated data. Input nodes have no incoming edges in the DAG, because they produce data from outside the computation model. *Transformation nodes* represent an operation applied to the node's input data to produce output data. Operations are defined as a combination of communication pattern (such as map, filter, reduce, and join) and a user defined function (UDF).

UDFs are an important part of the DAG computation model because they enable users to perform arbitrary computations on data, such as specialized transformations of input data objects, custom business logic, and development of new data mining and machine learning algorithms. To enable this flexibility, UDFs are written in a general purpose programming language, such as Java or Scala for Spark, and C/C++ for the original MapReduce [48]. UDFs provide such flexibility that Spark itself implements high level operations by combining UDFs with its lower-level operators such as building a join operation out of map and reduce.

As Spark and MapReduce do, this work assumes that UDFs are deterministic. Deterministic UDFs enable efficient fault tolerance mechanisms because any lost data can simply be re-computed using the information stored in the DAG. Neither Apache Spark or Hadoop MapReduce actually check UDFs for determinism, but we found this assumption to hold in all data processing programs we encountered.

4.2.2 Spark: A Motivating Example

Apache Spark is a distributed implementation of the DAG computation model[3]. In Spark, users use a simple API to compose together nodes. This DAG model also serves as the physical execution layer for higher level Spark APIs, such as Spark SQL and Dataframes [77]. In Spark, data in each node is not materialized until a user requests the data. At this time, the Spark execution layer will pipeline several transformations from related nodes together to produce an execution plan of stages to be executed.

[3]DAG nodes are called Resilient Distributed Datasets (RDDs) by the Spark authors

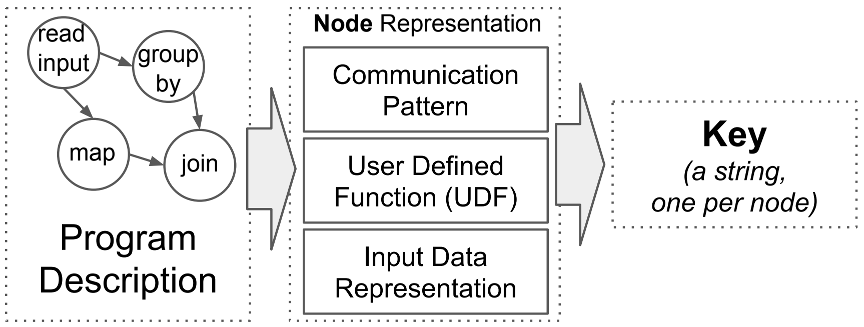


Figure 4.2: How a key is generated by KeyChain.

To improve performance, authors of Spark programs can materialize and cache data of an important node to speed up subsequent computations[4]. As fig. 4.1 illustrates, when a node is to be materialized, the execution layer checks for any cached data that could be used, and if found, incorporates the data into the execution plan. *However, data and computation duplication occurs* because Spark identifies cached data using increasing integer IDs assigned to each DAG node. Since different Spark programs assign the same IDs for different data, it is not possible to share data between programs, even when they have equivalent nodes[5].

To share cached data more widely, we use KeyChain keys to identify cached data so that equivalent nodes will be assigned the same key. As we will show in the evaluation, this helps to share data when the *same* program has multiple equivalent nodes **or** when *different* programs have equivalent nodes.

4.3 KeyChain

KeyChain generates strings that represent a DAG node's output, which we refer to as keys. Figure 4.2 shows an overview of how KeyChain computes a key: string representations of the communication pattern (§4.3.1), UDF (§4.3.2), and input data (§4.3.3) are combined into strings that represent each node. The string that KeyChain generates is: KeyChain(node) = '`<communication pattern> <UDF> <input metadata>`'. This key can be used to query any storage sys-

[4]Usually intermediate data is cached in memory, but it could also be serialized and written to an external storage system.

[5]Nodes are equivalent if they share the same input data, communication pattern, and UDF, thus producing the same output.

tem to find previously computed results that can speed up the computation.

Keys for all DAG nodes are computed before the DAG is executed so that the cache can be checked for prior results. Prior work can also use this key to identify results that are useful to cache (§4.6). Keys are computed starting from the input nodes, and the key of input nodes is used as the input data representation for transformation nodes (§4.3.3). The next sections describe the representations KeyChain uses for input, communication patterns and UDFs.

4.3.1 Communication Pattern Representation

As described in §4.2.1, a communication pattern is an operation such as map, group by, or reduce, and this name can be used as the string representation. For instance, the '`map`' string describes a map operation that applies a UDF to each element. If the communication pattern takes any extra parameters that affect the output, such as the hash key for a shuffle operation, those parameters should be included in the string. (§4.3.3).

4.3.2 UDF Hashing

A hash of the UDF is used as the UDF string in the key. Assuming the program is deterministic (§4.2.1), source code or any executable representation can be passed to the hash function. For the JVM (Java Virtual Machine), the best representation we found was the bytecode of the UDF because: (1) Overheads are low because bytecode is already produced during compilation, so KeyChain only needs to hash it. (2) Bytecode is less sensitive than program source to syntax variations like white-space, variable names, and comments, which means some functionally equivalent programs can share data, investigated further in §4.5.3. (3) Bytecode is accessible to programs running on the JVM (§4.4). To further reduce runtime overhead in the future, it is possible for compiler to hash the bytecode at compile time. For instance, we developed an LLVM pass that hashes each function at compile time and stores the hash for lookup via function pointer at runtime.

The hash should capture the entire behaviour of the UDF, so our implementation hashes the UDF and all its callees. If a function is common across all programs in the system (such as the Java standard library) it is possible to skip hashing these functions and just include the fully qualified method name since no information about program behaviour would be lost.

4.3.3 Input Representation

We break computing input representations into two cases: input originating from outside the DAG, and from other nodes.

Internal Data: Chaining

To represent input from other nodes in the DAG, KeyChain uses the key of the node that supplies the input, called *chaining*. For example, if node x takes input from node y, then Key(x) = '`<communication pattern> <UDF> Key(y)`'. Since the key of input nodes is computed before other nodes, Key(y) will be known when Key(x) is computed. Chaining enables computation of the key for all nodes in a large DAG without executing any parts of the DAG.

In the next section (§4.3.3), we describe a case where the system falls back to assigning DAG nodes integer based IDs (i.e. not KeyChain). All DAG nodes that receive input from nodes with integer IDs must also be assigned integer IDs and are not able to use KeyChain because integer IDs are not valid across programs (§4.2.2).

Outside Data: Origin Descriptors

The string representation of input data needs to encode when the input data last changed, so that out-of-date cached data will not be referenced. For instance, when a file is cached and the file contents change, a different key should be used to search for file so that old data is not referenced. Prior systems such as Nectar and Incoop accomplish this by using a hash of the data as the representation, which ensures the representation reflects the latest data, but significant I/O costs will be incurred to read large inputs (although the filesystem can be modified to compute this hash, as in Incoop).

Instead, KeyChain uses an origin descriptor (OD) to represent outside data. ODs should: (1) *describe where the data came from* (i.e. the file path) and (2) *include when the data was last changed, or the timeframe it is valid for* (i.e. the file's last modified time). Since the OD is computed whenever the key is computed, the key will reference the up to date data.

The exact way ODs are computed depends on the metadata available for a particular storage system. When the storage system does not have metadata about when data was last modified, the user could specify a time window for acceptably up-to-date data. For instance, a database query of total revenue might be acceptably up to date so long as it includes yesterday's sales, and this could be reflected in the OD as: '`<query string> <current day/month/year>`'. If users of different programs read the same data, but disagree on the acceptably up-to-date value, their programs will compute different keys and not share data.

If neither of the above approaches are possible because the storage system doesn't provide a last modified time and the program must read the most up-to-date data, this is a sign that the program and data it produces might not be suitable for memoization. In this case, we should also avoid hashing input. When KeyChain detects that there is no OD, it falls back to simple integer IDs as described in §4.2.2.

The one advantage that hashing data brings over the above techniques is that hashing data can ensure programs will share data even when they read from logically different places, such as duplicate files. Incoop relies on the storage system hashing stored data. KeyChain does not require that data is hashed, but could use a hash as the origin descriptor, if present.

Random Input Data

Random data can be generated two ways: by input nodes, or by a UDF in a transformation node. Generating random data in a UDF introduces non-determinism and is usually avoided when writing data processing programs (§4.2.1,4.3.4). However, random data is often generated by an input node, such as when initializing machine learning algorithms or generating test data. Since pseudo random number generators are deterministic with respect to their parameters [185], KeyChain assigns these nodes an OD that includes those parameters, such as the name of the algorithm, the seed, and the size of data generated.

4.3.4 KeyChain Assumptions and Limitations

Some limitations of KeyChain stem from limitations of UDF hashing: (1) Programs compiled by different compilers that produce different output will never match. (2) There is no guarantee that functionally equivalent programs will produce the same hash. If a compiler produces different output for equivalent programs that have minor syntax differences, then data can only be shared between UDFs that have exactly the same syntax. These limitations don't cause fundamental issues for CPM (sharing is still possible), but it is ideal to share data between programs that are *functionally equivalent* even if they are not syntactically identical. We investigate this further in §4.5.3.

KeyChain assumes that it is possible to access a hash of UDF bytecode. Bytecode is easily available at runtime from the JVM, but we are not aware of this feature in other languages/runtimes aside from the experimental pass we developed for LLVM. We hope that KeyChain leads to interest in adding such capability to other systems.

As discussed in §4.2.1, KeyChain assumes that UDFs are deterministic: that a program's output is determined by its source code and input. KeyChain does not add any program requirements, because this assumption is already made by Spark, MapReduce and others to enable fault tolerance.

Security – KeyChain does not enforce access control. An attacker could compute or guess a key for a file path they cannot access and try to read the data from a cache. Enforcing access control on cached data is challenging as access control is very deployment dependent. One way to enforce access control to cached data would be to ensure it can only be read by users that can access the original data. If a storage system is unable to enforce such access control, the cached data could be encrypted, and the encryption key(s) made available to those with the proper access rights.

4.4 KeyChain in Apache Spark

This section describes our implementation of KeyChain in Apache Spark 2.2. Since one contribution of this work is showing the simplicity of a CPM implementation, we describe our implementation, some challenges we encountered, and what made it simple. Overall, we wrote under 2100 lines of code, including a UDF hashing library for the JVM (<1000 lines) (§4.4.1) and modifications to Spark's execution layer to enable KeyChain (§4.4.2) and transfer data between Spark contexts (§4.4.3) (<1100 lines). None of the changes require modifications to existing Spark programs. Our changes to Spark are publicly available[6].

4.4.1 UDF Hashing Library

We wrote a UDF hashing library in Scala, and have made it publicly available[7].
Usage. The library provides `Hash(func)`, which accepts a reference to a JVM function and returns a string containing its hash. When a node is added to the DAG, Spark passes the node's UDF to `Hash` and stores the result.
Implementation. `Hash` uses the ASM analysis framework [144] to read the bytecode of the function and its callees. Class reflection [142] is used to find and hash all referenced variables. `Hash` applies filtering to remove unnecessary details from ASM's output (below), and passes the remainder to a cryptographic hash (SHA256). To improve performance, `Hash` caches previously hashed function bytecode (but not variable values).
Challenges. The largest challenge in using the UDF hashing library is ensuring that hashes don't needlessly vary. One way this happens is that Spark assigns a random UUID [186] to each JVM and tags all SQL expressions with it, making otherwise equivalent UDFs produce different hashes. To find such problems, `Hash` can output a JSON trace and our library includes tools to compare traces. We used traces to identify the UUID issue, and added filters to exclude hashing the UUID.

4.4.2 Execution Layer Modifications

Key computation. Since Spark implements the DAG model, it was simple to add KeyChain to the execution layer. Spark already has the information needed for KeyChain to compute keys and the program structure to chain keys from input nodes. *Communication pattern* strings were taken directly from Spark's names for DAG operations. *Origin Descriptors* were added for input nodes that read from

[6]https://github.com/craiig/spark-keychain/tree/branch-2.2-keychain
[7]https://github.com/craiig/keychain-tools/tree/master/udf-hash

HDFS and local files. *UDFs* were hashed using our library.
Using keys to identify data. As described in §4.2.2 and shown in fig. 4.1, Spark's execution layer indexes the cache using the integer ID of each DAG node to be materialized. Instead, we modified Spark to use keys generated by KeyChain. We retained the ability to use numeric IDs for cases where we could not compute or did not implement an origin descriptor (§4.3.3). Spark computes the IDs for every DAG node once so it does not add lots of extra computation to call KeyChain to generate keys. Overall, to support the entirety of TPC-DS, we added KeyChain to 26 different DAG node types in Spark, with each change averaging 14 lines.
Handling complex operations Our treatment of Spark's shuffle operation highlights how simple adding KeyChain can be, even with complex code. Shuffle is an all-to-all communication primitive used to implement reduce, group by, and join operations. Shuffle requires the movement of many data blocks between worker nodes, and each block is tracked using a numeric ID. We could have changed the shuffle implementation to use KeyChain keys for shuffle block IDs, but this was not necessary because only the final result of the shuffle is useful to cache. Instead of deeply modifying the shuffle implementation, we just used KeyChain to compute a key for the final output.

4.4.3 Cross Instance Sharing

Users often share a *single* Spark context, allowing them to share data that is cached in the same process, but users also use *separate* contexts for resource isolation. To enable sharing between separate contexts, we modified Spark so contexts can query one-another for cached data. So that applications do not inadvertently share data identified by numeric IDs, only KeyChain keys are requested from other contexts. (§4.2.2). In our current implementation, contexts are manually linked via an API call and no access control checks are performed when sending cached data to another context. Sharing between contexts requires extra computation to serialize and transfer the data to another context (§4.4.4), evaluated in §4.5.2.

4.4.4 KeyChain in Spark Limitations

The largest issue with adding CPM (via KeyChain) to Spark is that the lifetime of cached data can be short. Data in Spark's in-memory cache is managed by an application (driver in Spark terminology), so cached data is lost when an application shuts down. However, applications can be long lived when they are used interactively, such as when Spark is used in a shared notebook system like Jupyter [156] or Zeppelin [29]. To extend the lifetime, cached data could be stored on external storage such as HDFS [22] or Alluxio [16] with KeyChain being used to compute the filenames, so data can be found by later applications.

Like Spark itself, we rely on the programmer (or a higher level library) to mark useful results to cache. Prior work has shown how to automatically decide what to cache, and even how to re-write programs to achieve better sharing, so we did not investigate these issues in this work (§4.6).

The main drawback of reading cached data from other Spark instances or from a network file system is that serialization and deserialization overheads can greatly impact the speedup achievable from sharing (§4.5.2). Serialization overhead in Spark has been investigated by Ousterhout et al. [143] and Neutrino [188], and Skyway [141] presents a way to reduce JVM serialization overheads.

4.5 Evaluation

First, we show a situation where data sharing occurs to show how CPM (as implemented by KeyChain) can help to achieve more data sharing and high speedups thanks to the re-use of cached data (§4.5.2). Next, we show how UDF hashing can identify some functionally equivalent programs, so that sharing can occur in more situations (§4.5.3). Finally, we show that the overheads of KeyChain are negligible so it can be safely enabled even when data sharing may not occur and systems and users can benefit from CPM when sharing does occur (§4.5.4).

4.5.1 Experimental Setup

Test machines. We ran the tests in §4.5.2 and 4.5.4 on a machine with two 2.6 Ghz AMD Six Core Opteron 2435 processor with 32GB of RAM with a 160GB 7200rpm HDD used to store Spark shuffle data. For our larger scale tests in §4.5.4, we used Microsoft Azure, described later.

Spark. We tested an unmodified version of Spark 2.2 and our KeyChain implementation in Spark. For brevity in this section, we call our modified version KeyChain. We setup Spark to use all cores and all available memory. For our interprocess tests in §4.5.2, each instance used half the available memory.

Benchmarks and Test Data. For a source of realistic data and queries, our tests use TPC-DS [136, 153], a benchmark designed to represent modern decision support tasks, including ad-hoc and reporting queries. TPC-DS includes query and data generators. We used the TPC-DS benchmarking framework from Databricks [47] configured to generate and use Parquet [26] format data.

Avoiding JVM bias. Most of our tests ran on the JVM, which has a just-in-time compiler (JIT) that optimizes the program as it runs and a garbage collector that runs across allocated memory. To ensure consistent results (but not necessarily the best performance), we restart the JVM after each test so that later tests do not benefit from prior tests warming up the JIT optimizer [33, 116] and so that later tests are not hindered by the garbage collector cleaning up memory allocations performed by earlier tests. For our TPC-DS tests, we restarted the JVM after each full run of all queries.

4.5.2 How KeyChain Enables More Sharing

To highlight the gains that CPM can achieve through sharing data more widely, we perform a series of experiments to mimic an interactive use of Apache Spark in a context where data sharing occurs. Returning to the example from the introduction, we simulate Alice and Bobbie beginning to explore a data-set and running the same tasks: (1) *scanning* the TPC-DS `store_sales` table and caching it in memory for faster access, and (2) *joining* `store_sales` to the `customer` table and then caching those results in memory. Both tasks are written using the Spark Dataframe API. Our tests were run on a 2GB dataset on a Spark instance with 13GB in-memory cache.

Performance improvements of caching. Table 4.1 shows the overall performance of Alice and Bobbie running both tasks. Alice runs her query first and the data is loaded and cached in memory (Cache Miss in table 4.1). When Bobbie runs the same tasks, they re-use the data loaded by Alice (Cache Hit) leading to a 189x and 281x speedup for each task.

Table 4.1: Performance of cache hits and misses for both example queries where the data is shared within a process.

Shared Process	**Scan**	**Join**
Cache Miss	39±4.8s	101±2s
Cache Hit	0.21±0.02s	0.36±0.02s
Speedup	189x	281x

Table 4.2: Performance of cache hits and misses for both example queries where the data is *shared between processes*.

Interprocess (Serialized)	**Scan (s)**	**Join (s)**
Cache Miss	151±6s	98±3s
Cache Hit	56±2	22±1
Cache Hit w/ Data Xfer	69±6s	27s±1.1s
Hit Speedup	2.6x	4.4x
Hit w/ Data Xfer Speedup	2.2x	3.6x

Table 4.3: Cache hits and miss achieved in different sharing scenarios.

Same Code	Spark	Hit	Hit
	KeyChain	Hit	Hit
Same Process	Spark	Miss	Miss
	KeyChain	**Hit**	**Hit**
Diff. Process	Spark	Miss	Miss
	KeyChain	**Hit**	**Hit**

The above speedups are high because data is re-used from the same process without serialization or data transfer overhead. Table 4.2 shows the performance when data is serialized in memory and transferred to another process on the same machine. When serialized data is read from the same process, speedups drop to 2.6x-4.4x, and to 2.2x-3.6x when serialized data is transferred to another process.

High serialization overheads are endemic to Spark, and not due to anything added by KeyChain. Not all systems will have such high interprocess overhead.

KeyChain's increased sharing potential. Now, we look at **if** sharing will occur when Alice and Bobbie run their tasks in different sharing scenarios, and how CPM (as implemented by KeyChain) can achieve more cache hits than unmodified Spark. One way that these scenarios occur in practice is when users are using interactive notebooks like Jupyter [156] or Zeppelin [29], which allows users to work collaboratively on the same code or just share a connection to a single Spark context. Table 4.3 shows the results. **Same Code** is when Alice and Bobbie are collaborating to write the same program and can reference each other's variables. *This is the only case in which unmodified Spark can make use of cached data.* **Same Process** and **Diff. Process** are when Alice and Bobbie write completely separate code, but that code is run in either the same or different process. **KeyChain achieves more cache hits than Spark because KeyChain enables cross program memoization.**

4.5.3 Syntactic Resilience Evaluation

This section investigates how compilers transform syntactically different but functionally equivalent input programs into the same output, a property we call *syntactic resilience*. If the compiler produces the same output for equivalent programs, then the UDF hash is the same, and computation can be shared *across* equivalent programs written by different users. Our goals in studying syntactic resilience is to: (RQ1) justify our choice of hashing UDF bytecode versus using program source, (RQ2) understand the limitations of using compiler output to detect program equivalence, and (RQ3) measure the current syntactic resilience of different compilers.

Methodology

One way to measure syntactic resilience is to collect real-world programs, manually verify (or create) equivalent UDFs, and then check if a compiler produces identical outputs. However, this does not help predict whether or not this effect will generalize to other UDFs that are not in the test set. Therefore, we developed a test suite to test compiler's resilience to specific syntactic variations to better understand the limitations of each compiler and syntactic resilience in general.

To create our test suite, we surveyed several sources (described below) to create a corpus of 64 syntactic *variations* that can be applied to change a program's syntax without changing its functionality. For each variation, we derived a test case of at least two code snippets, where the variation has been applied to the first to create the second. Table 4.4 lists some of these variations and example test cases. We have made our benchmark publicly available[8].

We do not claim our set of variations is exhaustive, but we do cover a larger variety of non-functional syntactic differences than prior work. Prior evaluations of syntactic resilience [76, 95, 150, 165] tested variations that *change* a programs functionality, such as adding a postfix increment operator to every variable reference and biased their results by repeatedly testing the same variation sometimes thousands of times.

We briefly describe our sources, why we chose them, and how we derived variants from each. We have made our full methodology and source data available, with direct links provided as footnotes on each source.

Source 1: TPC-DS[9] (7 test cases). To identify variations that can appear in data processing programs, we surveyed all expressions from TPC-DS and considered ways to re-write them while still retaining functional equivalence, and then added those variations to the corpus. For instance, if an expression included a commutative binary operator, then the operands could be swapped, which resulted in us adding the `Swap Operands` variation to the corpus.

Source 2: Compiler Documentation[10] (41 test cases). Since syntactic resilience is dependent on a compiler's optimizations, we exhaustively surveyed LLVM and GCC's optimizations and canonicalizations [62, 120]. When an optimization would transform equivalent code into the same output, such as LLVM and GCC's tree reassociation pass that canonicalizes mathematical expressions, we included a variation to test that optimization. Compiler optimizations are not guaranteed to produce the *same* output as a hand optimized version, but we find many do. We chose not to include some optimizations when the resulting test case would simply compare an unoptimized version to a hand optimized version that we felt was unrealistic for

[8] https://github.com/craiig/keychain-tools/tree/master/resilience_eval
[9] https://github.com/craiig/keychain-tools/tree/master/tpcds_analysis
[10] https://github.com/craiig/keychain-tools/tree/master/compiler_analysis

a programmer to write, such as auto-vectorization or jump-threading.

Source 3: Related Work[11] (11 test cases). TCE [150] tested against a set of manually verified equivalent programs taken from real world code. We grouped these equivalent programs into 11 test cases that tested distinct variations. We excluded TCE test cases that overlapped with existing tests in our corpus, but kept tests that were more complex than our prior test cases to evaluate real-world examples.

Source 4: Miscellaneous. The remaining 5 tests in our suite include a test for resilience to whitespace, `if(x!=0)` equivalence to `if(x)`, associative constant folding, pointer dereference equivalence to array indexing, and a for-loop with a single iteration.

Studied Compilers. We evaluate several compilers because while Java and Scala are used by Apache Spark, this is not the only choice for data processing systems. Data processing systems that compile to native code can outperform JVM-based systems [53, 140], so we test GCC and Clang (LLVM) to see if they are a better choice for equivalence checking. To ensure the code snippets were consistent in each compiler, we ported tests to each language with template-based code generation. Scala lacks a post- and prefix increment operators, so our Scala test skips those tests. To measure resilience, we compile each variant with each compiler and hash the output. For Java and Scala we use the UDF hashing library from KeyChain (§4.4.1). For Clang/GCC output we hash the assembly output.

Results

We classify how well a compiler successfully performs on each test based on the number of unique outputs relative to other compilers. **Pass (P)** is when a compiler produces *one* unique output from a number of unique input programs. **Qualified Pass (QP)** is when a compiler produces more than one unique output, but no other compiler produces fewer. Qualified passes ensures we don't penalize compilers for difficult tests when no other compilers can do better. **Partial Fail (PF)** is when a compiler produces fewer unique outputs than it has inputs, but does not achieve a pass or qualified pass. **Fail (F)** is when a compiler produces the same number of unique outputs as unique inputs.

[11] https://github.com/craiig/keychain-tools/tree/master/tce_analysis

Name and Description	Example(s)	Scala 2.12	Java 1.8	Clang 7	GCC 7
		Optimizations Off / On (where applicable)			
Whitespace Add, change, remove whitespace, comments, variable names	(x+y) (x + y) (x + /* hello */ y) (w+z)	**P/P**	**P**	**P/P**	**P/P**
Swap Operands Swap the operands of a binary commutative operator	x+y −> y+x	F/F	F	F/**P**	F/F
Logical Operand Swap Swap the operands of a logical comparison. Short circuiting in language specifications disallow treating these statements as equivalent.	x || y y || x	F/F	F	F/F	F/F
Constant Folding Replace a constant with an equivalent expression	x+2 x+1+1 x+(1+1) 1+1+x	PF/PF	PF	PF/**P**	**P/P**
Algebraic Tree Reassociation Reassociate commutative expressions to promote better constant propagation [120]	(x+y)−(z+v) ((x+y)−z)−v (x+y−z)−v x+y−z−v	PF/PF	PF	PF/**P**	PF/PF
Comparison Invert Apply De Morgan's laws to a comparison operation	(x==1) && (y==2) !((x!=1) || (y!=2))	**P/P**	**P**	PF/**P**	**P/P**

Comparison Swap
Invert the comparison statement and swap the basic blocks of an if-statement (*Passes on GCC 4.9 -O3)
	if (x){return y;} else{return x;}
	if (!x){return x;} else{return y;}
		F/F	F	F/**P**	F/F*

Loop Invariant Hoisting
Move a loop invariant from the inside of a loop. All tests are qualified pass as no compilers canonicalize to our hand hoisted version.
	int a = 0; for (...) { if (x){ a=y; } ... }
	
	int a = 0; if (x and loop_condition){ a = y;}
	for (...) { ... }
		QP	QP	QP	QP

Table 4.4: Selected results from syntactic resilience benchmarks.

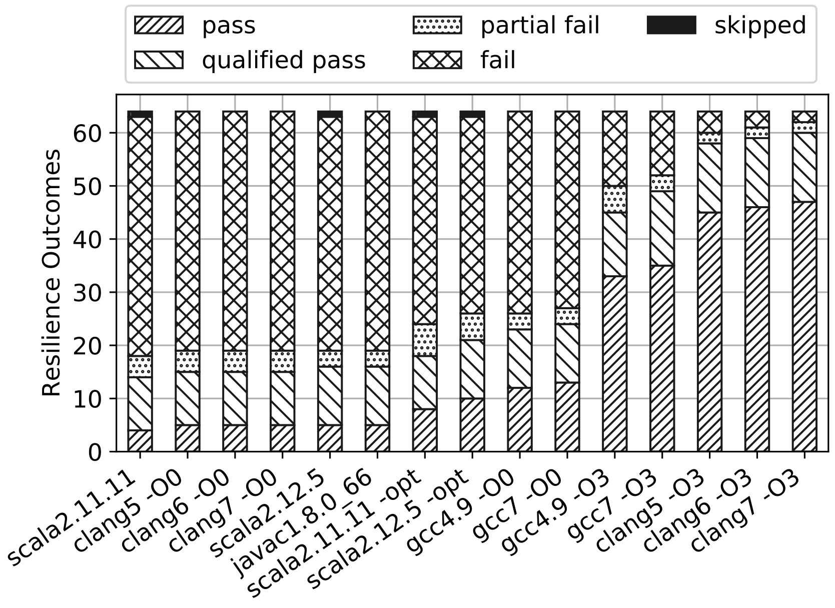


Figure 4.3: Overall Syntactic Resilience benchmark outcome.

RQ1: Is hashing bytecode an improvement over program source? Table 4.4 presents selected results from our benchmark to highlight common program variations. The **Whitespace** row shows *all compilers can provide basic resilience to whitespace, comments and variable name differences, an improvement over using program source.*

RQ2: What are the limits of compiler-assisted equivalence? All compilers fail Logical Operand Swap (LOS) because, strictly speaking, this variation is not functionally equivalent. Short circuit evaluation for logical-or means operand ordering changes execution behaviour, even though the order may be irrelevant. LOS highlights the limits of syntactic resilience: *when syntax variations imply changes in execution behaviour, these differences are not optimized away by the compiler.* This applies to memory allocations, side effects, early exit conditions, and more.

RQ3: What is the syntactic resilience of different compilers? Fig. 4.3 shows the aggregate results for each compiler. Compilers that perform more ahead of time optimizations are better at canonicalizing functionally equivalent code. Clang and GCC include many ahead of time optimizations and are the state-of-the-art for syntactic resilience. Scala and Java have less ahead of time optimizations because they rely on JIT optimization , but Scala with ahead of time optimizations (`-opt`) performs better than without. We also found GCC and Clang could be improved: most failures were due to variations in label names, register allocation, and ordering of independent instructions.

Overall our tests show that compiler canonicalization and optimization passes can be used to detect some equivalent programs. KeyChain leverages this effect to achieve more data sharing. While the syntactic resilience of Scala and Java lags behind GCC and Clang, finding equivalent programs by hashing JVM bytecode is more likely than using program source or simple integers (§4.2.2) to identify UDF computations, and our test suite can help guide future improvements.

4.5.4 Overheads

Now we examine the overheads of using KeyChain to implement CPM in Spark. We first measure the standalone performance of the UDF hashing library (§4.5.4) and then the overall impact on TPC-DS when there is no potential for sharing (§4.5.4).

UDF Hashing

Fig 4.4 shows the time to hash each of our Java and Scala syntactic resilience tests (§4.5.3) . The time to hash a UDF is determined by the number of bytes and functions hashed. The largest UDF takes under 350 milliseconds, while the smallest takes under 150 milliseconds. This shows UDF hashing performance in the conservative case, because each test is hashed using a newly created JVM with no cached results.

Table 4.5 shows UDF[12] hashing time for TPC-DS queries during the tests in (§4.5.4). We also show performance when debug tracing is active and logging all hashing traces to disk, which is only used to debug UDF hashing (§4.4.1). It

[12]The UDFs in this case are Spark's database operators.

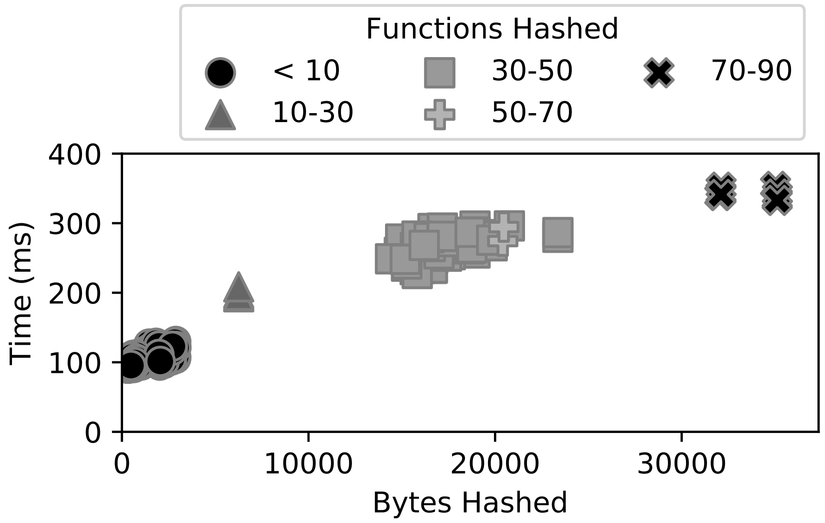


Figure 4.4: Overall time to hash Java and Scala functions from our resilience benchmark using the UDF hashing library.

Table 4.5: UDF Hashing Overhead on TPC-DS. During *Normal* operation, and *Debug* when UDF tracing is enabled for debugging.

Mode	**Max**	**Avg**	**Min**	**Sum**	**Total**
Normal	265ms	2ms	0.07ms	18s	9,345
Debug Miss	322ms	65ms	4ms	1.1s	17
Debug Hit	1.5s	11ms	0.1ms	108s	9,328

takes 18 seconds to hash all UDFs used in TPC-DS, for an average of 2ms per UDF. There are over 9000 UDFs because Spark's operators are implemented as UDFs (§4.2.1). Compared to our prior results with the UDF hashing library, UDF hashing in a realistic scenario is much faster due to caching previously hashed functions and a warmed-up JVM.

End to End Spark Performance

To show that KeyChain allows CPM to be enabled in all deployments without incurring high overheads, we compare the performance of TPC-DS with and without

Table 4.6: Machine and TPC-DS Configuration for end to end tests.

Name	**Small**	**Large**
TPC-DS Scale Factor	10	1000
Partitions	50	160
Masters:Workers	1:3	1:20
Instance Type	D4S_v3	E8_v3
vCPUs per node	4	8
Memory per node	16GB	60GB
Storage per node	32GB	200GB

Table 4.7: Performance of TPC-DS. Average Total Completion Time (*ATCT*) is the average total runtime of each TPC-DS run.

	Small		Large	
	Spark	KeyChain	Spark	KeyChain
Iterations	5	5	3	3
ATCT (s)	3628	3684	34,328	33,732
Std.Dev.(s)	75	83	1002	857
	2%	2.2%	2.9%	2.5%

KeyChain when there is no sharing. In both cases, the empty cache is checked for relevant results. In the original Spark, the cache is indexed with integers. In the KeyChain version, the cache is indexed with KeyChain keys. Tests were run on large and small scale clusters on Microsoft Azure. Machine details and TPC-DS parameters are in table 4.6. HDFS was used to store data. TPC-DS includes 104 queries, but we disabled query 72 on Large due to too much shuffle data for some nodes, and 77 due to a memory leak when over 40,000 tasks were generated.

Table 4.7 shows the results of our tests. Overall, we find that KeyChain has a negligible impact when there is no sharing, because the performance is within the standard deviation of the original Spark. We estimate that sources of system jitter like garbage collection [122], JIT compilation [33, 116], and stragglers [48, 190] make a larger impact than KeyChain. While our CPM implementation is decidedly simple, we have shown that KeyChain can implement CPM without incurring high run-time overheads when sharing is not possible.

4.6 Related Work

Memoization

Early work on memoization investigated how caching can help *single programs* [131] and what kinds of programs are suitable for caching [132]. KeyChain applies memoization *between* data processing programs.

Explicit Sharing and Incremental Processing Systems

When programs are *known* to benefit from prior results, programmers will write them to explicitly share data. For instance, Spark users often use Alluxio [16][13], an in-memory cache for distributed file systems, to store data they know can be shared. If a program can re-use its own results, incremental processing frameworks can be used. For instance, Incoop reports overheads of 5-22% for initial runs with no cached data, but this pays off with 3-1000x run time improvement on subsequent runs [36]. However, explicit sharing or incremental processing depends on individual programmers knowing about a sharing opportunity ahead of time. Low-overhead techniques to identify results of computation like KeyChain are useful when opportunities for sharing are *not guaranteed* or might be difficult for programmers to identify.

Nectar [68] uses similar techniques to KeyChain and reported a 20-50% reduction in computation time on Microsoft's clusters due to CPM. Nectar hashes C# bytecode but also *hashes input data*, a source of overhead that KeyChain avoids. Nectar does not report overhead, implementation complexity, or syntactic resilience of the hashing technique and their implementation is not publicly available. KeyChain contributes an improved design with negligible overhead so that CPM can be deployed more widely, and a evaluation of syntactic resilience that tests the limits of UDF hashing.

Materialized view selection and cache placement algorithms

Materialized view selection techniques analyze past SQL queries to determine subexpressions to cache for the benefit of future queries, but do not consider UDFs [87,

[13]Formerly Tachyon [112]

103, 164] or if equivalent UDFs can be shared between programs [108, 109]. Spark's SQL caching layer will search for relevant cached SQL expressions, but not UDFs and does not work across Spark instances. KeyChain computes identifiers for the results of UDF computations that these techniques can use to decide what is beneficial to cache [17, 188]. For instance, [87] shows that they can save up to 40% of machine hours in their cluster. However, these benefits can only be realized once the common subexpressions can be identified.

Value Numbering in Compilers

Value Numbering (VN) [179] is a compiler pass that eliminates redundant expressions by computing identifiers for expressions in a basic block by hashing operands and operators. KeyChain can be seen as applying ideas from VN to the data processing setting.

Program Equivalence Detection

UDF hashing is a heuristic equivalence detection technique. Program equivalence is undecidable [177] so feasible approaches are either provers or heuristics. Provers, such as Cosette [43], search for a series of valid transformations to transform one program to another, but are not ideal for low overhead equivalence checking because they cannot guarantee termination and would need pair-wise comparisons over all programs with cached data. Clone detection [165] detects *similar* programs, but considers programs with *different functionality* to be similar, which is not suitable for CPM. Trivial Compiler Equivalence (TCE) [76, 95, 150] eliminates redundant test cases by compiling and removing test cases with duplicate outputs. KeyChain uses this same effect to gain wider sharing potential, and our evaluation tests only functionally equivalent variations (§4.5.3).

4.7 Conclusion

This work showed that KeyChain enables cross program memoization implementations to achieve significant performance benefits when sharing occurs, while incurring negligible overheads when sharing is not possible. We hope these results encourage the addition of CPM to more data processing systems.

Chapter 5

Applying Compilation to Systems: Lessons Learned

My work takes a broad view of compilation. This chapter situates my work in the broad space of domain-specific compilation, and presents lessons learned from designing compilation-assisted techniques.

5.1 Compilation Construed Broadly

Often I have found many researchers and engineers think of compilation as being done once, ahead of time, where source code written in a high level language is mapped to machine-specific code. After this step, the resulting machine-specific code must deal with all complexities of the problem in question. I call this the *classical* approach to compilation. Programs built for classical compilation are loosely structured by necessity to handle various special cases, instead of being organized as simple 'straight-line' code. Such loose coupling limits the optimizations that classical compilers can perform because static analysis cannot often infer runtime behaviour. This approach leads to machine code that consists of many branches and additional code to handle special cases, restricting opportunities for optimization. Another well-known approach is just-in-time (JIT) compilation of general purpose languages, such as Java, where intermediate byte-code is compiled to machine-specific code on the fly and specialized for observed behaviour.

Modern database systems take a different approach where ahead-of-time compilation is *mixed* with execution. There are at least two forms of compilation in modern database systems: As described in §3.2, a user's query (often written in SQL) is first compiled into a logical query plan, which is then compiled into a physical query plan of implementations the database system knows how to execute. This kind of compilation is usually called *query planning*. It can be challenging to find the best physical plan, because the interactions of hardware, database software, and properties of the input data are difficult to estimate and can produce surprising performance outcomes.

After selecting the physical plan, many modern database systems employ another form of compilation: *code generation.* The query engine generates and compiles specialized 'straight-line' code for a specific query, resulting in increased performance [30, 51, 130, 140]. While this approach is similar in spirit to JIT compilers used by interpreted languages, modern database systems usually compile code *once* to a lower-level target, instead of iteratively refining running code as JIT compilers do. Some database systems produce byte-code that is evaluated by an intermediate VM [180]. SQLite targets its own VM [7], while Spark SQL generates Java code that is executed on the JVM [30]. Databases like HyPer [140] directly target LLVM's intermediate representation (IR). MemSQL [6, 51] targets a custom intermediate representation that is then compiled to LLVM IR. For a more in-depth overview of query compilation, Menon et al. [130] provide a great overview with relevant citations.

Both query planning and code generation exploit more domain knowledge than is available in classical compilation, but must also deal with domain-specific challenges. Query planning integrates knowledge of the behaviour of database operations and the properties of the *particular datasets* to be queried, but leveraging this knowledge well can be complex. For instance, one important job of query planners is to pick the best join ordering, but picking the best ordering is highly dependent on accurate cardinality estimates [110]. Code generation exploits the precise structure of a query, and knowledge of how the semantics of relational operations can be physically realized on a particular machine architecture. But, the best implementation on a particular machine can be query specific. Kersten et al. [93] show that some queries do better using vector or scalar processing and vice versa.

Halide [157, 158] builds domain specific languages to address this problem, splitting the definition of an algorithm from a particular implementation of it (called the schedule), but leaves the schedule to experts. TVM [40] uses machine-learning approaches that learn the best way to map model training and inference to available hardware implementations. I am also not alone in noting similarities and opportunities between classical compilation and database systems. Pirk et al. [152] note that there there is an under-exploited opportunity for data-analytics optimizations to be exported to general purpose programs via the compiler.

Jumpgate and KeyChain fit amongst this work because they also exploit the additional domain knowledge available in data analytics systems. They also take on more domain-specific challenges.

Jumpgate shows how existing query plans can be exported from a client system, and used to select and configure suitable NCA implementations. Jumpgate also provides a new opportunity to exploit domain knowledge for hardware design. Instead of asking hardware designers to support general purpose operations, Jumpgate provides more precise specification: a modest set of relational operations and the expressions that occur in them.

Jumpgate also takes on more complexity than even database compilation approaches have had to. Jumpgate considers currently available devices before choosing how to compile a query to a logical plan. And, Jumpgate gives the NCA implementor the opportunity to further specialize and configure the device for the query in question. Production systems based on Jumpgate can be aware of what kinds of devices are *currently available at the moment* rather than just targeting a general device that we can assume to be available.

KeyChain uses the data-dependency semantics of analytics programs: data produced by a node in the DAG only changes when its input changes. KeyChain applies a decades old memoization technique [131, 132] in a more challenging setting. Prior work employed memoization in situations where it is known to be effective so any overhead from memoization was always worthwhile. But, potential for data re-use is harder to predict in multi-user environments that KeyChain is designed for. Therefore, techniques like KeyChain need to be low-overhead so they can always be enabled. KeyChain also leverages existing compiler techniques to canonicalize input, benefiting from decades of work on compiler optimization.

5.2 Considerations When Designing a Compilation Approach.

This section presents an overall approach to designing compilation-assisted techniques to solve systems issues. My hope is that this section helps inspire future compilation-assisted work. We consider a few preliminary questions to determine if a problem is suitable for a compilation approach, discuss why prior techniques have been so effective, and relate these questions to the design decisions of Jumpgate, KeyChain, and systems in the literature.

In this section I will use ‘query’ as general short-hand for any user-specified task, not just a query submitted to a database system. This could be a command-line invocation of a program, a series of API requests made over some transport (e.g., REST, RPC), or via buttons clicked in a UI.

How hard is a general solution? How hard is a specific solution? In general, domain specific compilation techniques are effective at increasing performance because they consider several *precise* problems in lieu of chasing a single general purpose solution. In many cases it is difficult to create a single piece of machine-code that can execute *all* queries quickly when queries are diverse and data-dependent. Compilation approaches solve a fundamentally easier problem: given a *particular* query and *particular* input data, they create code to efficiently answer *that* query. The first clue that a problem might require a compilation approach is that individual queries can have efficient hand-implemented versions, but a general version is even harder to write than the hand-implemented version, and perhaps would not execute as quickly.

This is especially true of data analytics, where each query is a complex program submitted by the user and it is well known that hand-written queries can perform well [140]. I also encountered this with Jumpgate – researchers were successfully executing *individual* queries on programmable switches, but their implementations could not be readily applied to other queries, because each implementation made query-specific assumptions. At the same time, the implementation itself was complex, so it is important to make that effort count across many queries. KeyChain relies on the host execution to take a compilation approach so there is an opportunity to rewrite the query to re-use existing results.

How will you offset extra compilation time? Compilation approaches fundamentally trade off additional start-up time for better performance during execution. This means a successful compilation approach will spend more time executing generated code than generating code. It is important to consciously make this trade-off and design a system that ensures this will be true. In designing Jumpgate, I knew it would have relatively high start-up overhead, but that it would not be too much of a problem because Jumpgate is intended for processing lots of data. KeyChain is designed to have very little overhead to avoid significantly increasing compilation time. Sometimes compilation overhead is higher than execution time, but if queries are similar to one another, there can also be opportunities to memoize and re-use generated code. One can take advantage of the generality vs specificity spectrum: it is possible to generate a *mostly* specialized version that can be re-used, instead of a *very* specialized version that cannot.

How well-defined is the query? Data analytics are a good fit for compilation because SQL queries are a relational algebra, and they have very precisely defined semantics over well defined input data. This provided an opportunity for a compiler to understand the *intent* behind a user's query and create code that embodies it well. Imperative APIs such as filesystems `open/read/write` or key-value stores `get/put` interfaces do not have such rich amount of information. The type of data being stored, and even the access pattern can be often opaque to the underlying system. It is no coincidence that data analytics tasks can often be expressed in functional style APIs, as in Apache Spark [191] and Weld [146, 148].

Jumpgate and KeyChain both take advantage of the well-defined nature of data analytics queries to accelerate them. For KeyChain, since queries are read-only, and the output of each relational operator is determined by the inputs, it is correct to rewrite the query to reference saved data. For Jumpgate, we can similarly rewrite the query plan to be executed by NCAs, changing how the query is physically executed, but in a way that guarantees the results are still correct.

Constructing well-defined queries. If the query is not *yet* well defined, perhaps you can define a simple domain specific language. Work such as Green-marl [75] has done so for graphs, Halide [157] did so for image processing, Tensorflow [9] did so for machine learning. One helpful technique is to use a host language to implement an API that can describe the query. Many domain specific compilers and languages let users write code in a host language, such as Python, that when executed, builds an internal domain-specific AST. This internal AST can then be passed to a domain specific compiler, often implemented in the host-language. This is very similar to lazy evaluation. This trick helps eliminate some of the more rote tasks of defining a DSL, such as defining a grammar and implementing a parser, allowing implementors to skip directly to solving the compilation challenges of their domain. Examples include Spark's Dataframe API, machine learning frameworks like Tensorflow [9], and Legno [10]. Legno defines dynamical systems in Python, and compiles them to analog devices, a very complex execution environment. Jumpgate has a similar API for the Python client that helped drive initial development.

What will be the execution target(s) of the compiler? When designing a compilation approach it is important to carefully consider what problems you must take on, and which issues you can externalize. As we've discussed, applying a compiler approach to a system is fundamentally accepting more challenges, so it is important to be selective. One way to control which problems you take on is by carefully choosing the *compilation target*. Many compilers have several intermediate phases, each with different targets. For a given problem, it can be useful to define new targets.

For Jumpgate, our eventual target was programmable switches and software-based NCAs. It was not clear at first how to compile a user request to these devices: *Should we generate P4 code directly for programmable switches? What if a query can't be supported? What data format will they communicate with?* It helped greatly to define two intermediate targets: staged networked pipelines (SNPs), and network tuple formats (NTFs). SNPs simplified the problem of mapping a client request to NCAs, and provided a high-level interface for each NCA backend. NTFs precisely define the query-specific data format, making it easier to generate code. SNPs also helped to abstract the code generation problem: generating code became a mostly separate problem from mapping the query to NCAs. To me, these abstractions will be the lasting impact of Jumpgate.

For code generation, Jumpgate's software NCA code generation targeted C++, relying on the underlying compiler to perform optimizations (e.g., dead code elimination, constant propagation) to produce efficient machine code. This made the development of our code-generation library much easier, as I could implement a simple template-based code generator that relied on constants to activate/deactivate specialized code branches. We forewent generating P4 code in this iteration of Jumpgate. The choice of SNP allowed us to simply parameterize the aggregation implementation to send to the right IP/port, and Jumpgate's operator interface will use the switch aggregator whenever it can apply.

Wrap-up and Related Work These questions are some of the first things I look for when considering taking a compilation approach. Instead of re-iterating the work of others, I leave you with some related work that inspired these research directions.

Flare [53] and its Light-weight Modular Staging (LMS) [161] approach showed me that many problems could be cast as compilation problems. LMS is not just a great technique but a nice *reasoning* tool that can be used to understand if and how programs can be specialized. LMS taught me to think of programs in terms of *stages* of progressive compilation.

There is a breadth of work that applies optimizations to computational DAGs to solve a variety of challenges. Overall, these work showed me that DAGs are a powerful program representation that can be useful in many surprising places. Resilient Distributed Datasets [191] and Spark SQL first showed me the power of DAGs for fault tolerance. Fouladi et al. [60] extract DAGs from existing build systems by inserting stubs for arbitrary commands, and make use of memoization and high levels of parallelism to quickly compile large programs. Conclave transforms computational DAGs to guarantee privacy [181] in secure multi-party communications. Dandelion [163] demonstrated how DAG computations can be mapped to heterogeneous devices. These works showed me how the DAG model of computation could be effective to solve a diverse set of problems.

During my two years at Oracle Labs while I worked on my PhD, I worked on compiler tooling, particularly on GCC internals [3] to optimize GCC's output for a novel computer architecture [12, 31, 65]. This work taught me that production compilers are just a big collection of heuristics and optimizers joined together with intermediate representations and clever interfaces. Specifically, understanding RTL [5] and how it was used to map logical instructions to machine instructions [4] was fundamental to helping me develop compilation approaches for the research presented here.

Chapter 6

Conclusion

This thesis presented two systems to improve the performance of data analytics: KeyChain and Jumpgate. Both systems employ compilation-based techniques to achieve performance improvements for data analytics systems.

KeyChain helps analytics systems re-use data generated by multiple users with low overheads. KeyChain takes advantage of the data-flow graph used by modern data analytics systems to compile and execute queries. KeyChain re-writes data-flow graphs to use prior results. KeyChain contributes a simple but effective heuristic for detecting equivalent but syntactically different programs, and contributes a benchmark to evaluate future compilers for syntactic resilience.

Jumpgate introduces a novel execution paradigm and interfaces that enables networked accelerators to be used as part of data analytics pipeline. As systems designers increasingly turn to domain specific architectures to improve performance, the fundamental contributions of Jumpgate will enable these devices to be more easily integrated into analytics systems. I look forward to seeing and working on new hardware designs that are built assuming that systems like Jumpgate can coordinate their execution.

6.1 Discussion

6.1.1 Jumpgate

Jumpgate's lasting contribution will be that it showed how clients and NCAs could made to work together, thanks to network tuple formats (NTFs) and staged network pipelines (SNPs). This establishes a broad space for exploration of future accelerator designs. Now that we have a feasible execution model and working prototype, we can begin answering questions about the benefits and costs to network connected acceleration for analytics in light of future trends.

Jumpgate itself provides an experimental platform to answer these questions, and the simplicity of staged network pipelines allows for modelling behaviour analytically to derive new insights about using accelerators. To illustrate, this section presents several future questions that can be investigated with Jumpgate and describes approaches to answer them.

When will Jumpgate be a win?

A key question for future work to pursue is when NPaaS/Jumpgate will be a win on given hardware accelerators. This question deserves its own section. As we saw in the evaluation, Jumpgate is a win when the accelerators used can outperform the client system on the offloaded operations and the data can be transferred back quickly enough for the client to complete the remainder of the query. While the software-based NCAs and switch example showed some wins, we were limited by access to available hardware accelerators and could not show clear wins in all circumstances due to our software implementations. We expect even better performance will come with future implementations of analytics operations on hardware accelerators, as indicated by prior work (§3.2). Future work should therefore seek to develop and evaluate new accelerators, but also to develop a model that can be used to estimate impact from new opportunities as networking improves and new devices are introduced. Jumpgate can help in both these directions.

One can evaluate new accelerators experimentally on real queries, as we did with the Jumpgate prototype system in § 3.5.5. Here, Jumpgate helps because it provides framework to support a diverse set of accelerators and integration into client systems. Before Jumpgate, performing such an evaluation would have required the development of format parsers, data senders, client integration, and potentially hand orchestrating query execution in lieu of an execution model. Thanks to Jumpgate, answering this question experimentally becomes a matter of implementing the analytics operations on the target accelerator and adding it to Jumpgate, rather than the tougher integration problem that prior work had to wrestle with [111, 166, 175]. While Jumpgate is not a panacea because analytics implementations can be challenging in themselves, Jumpgate helps researchers focus on the problem of implementing analytics operations on the accelerator being studied by providing a framework to work within. A promising direction is integration with SmartNICs, discussed below.

Before implementation, it is beneficial to analytically model the expected behaviour of an accelerator to determine if it is worth implementing, or to evaluate designs that are not yet feasible to implement. Here, Jumpgate's contributes a simple execution model that works for many queries and can be modelled analytically: Staged network pipelines are essentially parallel pipelines, and there is extensive prior work that models and reasons about pipeline parallelism in computer architecture [71] and the software community [101, 102, 107]. There is also fundamental work that helps to model communication costs of applications, such as LogP and it's refinements [15, 44, 126]. Future work could create analytical models of staged networked pipelines based on real queries, and explore how the performance and costs change as new accelerators are added and parameters are adjusted.

For example, one question that modelling needs to help answer is: **Is it better to combine all operations into one device, or distribute them across many devices?** Should operations be merged together into a single NCA (localized) or split apart across many NCAs (disaggregated)? This fundamental question has implications for how NCAs are designed overall, and dictates whether we should aim to perform analytics operations strictly on single nodes, or distribute them throughout the network. This question also has implications for how much network bandwidth will be used by the query.

To begin answering this question analytically, we can consider the trade-off between communicating the tuple or just performing the operation locally. We can break the time for an NCA to process a tuple into two parts (1) the time to read and write each tuple to the network (or to a buffer that will be read/written in parallel by the NIC), and (2) the time to process the tuple through the NCAs assigned operations. These parts are derived from LogP's overhead parameter [44] and reflects the time that a processor will be occupied working on a given tuple, and thus determine the throughput of the pipeline. When it takes less time to perform the operation locally (part 2) than to send the tuple on the network (part 1), we should favor localizing operations as this will reduce per-stage latency and thus improve pipeline throughput. However, when part 1 drops below part 2, it becomes beneficial to disaggregate operations across NCAs to improve throughput and gain parallelism from pipelining. Disaggregation means we can overlap the operation with prior and subsequent operations on other tuples, and means that we can start designing per-operation accelerators for specific activities rather than general purpose devices.

To get a specific answer as to whether disaggregation is worthwhile, we can compare these factors across different devices. Programmable switches with fixed latency pipelines have the same per-tuple latency regardless of the operations performed and can emit a packet every cycle, so localization and disaggregation are equivalent in terms of performance[1]. On general purpose processors, a hash-table lookup for join or aggregation operations incurs a worst case 1-2 cache misses, plus additional cycles for instructions to perform the operation. Given the rough 100ns rule of thumb for a cache miss, this gives roughly 200ns for join and aggregation operations. Prior work from network functions [149] reported a packet can be received and forwarded in 40ns with a state of the art framework that uses kernel bypass[2]. However, this is with an extensively optimized network path. Zilberman et al. [196] measured request-reply latency with kernel bypass at ≈ 946ns with a contribution of ≈ 572ns from the interconnect, leaving 374ns for the round-trip that we attribute to software overheads and giving roughly 187ns for a single direction[3]. This implies disaggregation could also be effective on general purpose

[1]One should pack as many operations into each programmable switch to avoid squandering this expensive accelerator.

[2]Based on the experiment in figure 4 of [149] showing NetBricks achieved almost 25 Mpps using a single core.

[3]The disparity between these two results could be explained by the NF case measuring overall packet throughput that amortizes buffer read/write costs while Zilberman et al. measure single packet

CPUs using efficient kernel bypass networking. There are also proposals that integrate a NIC directly with a CPU [79] that would allow a CPU to read a packet directly from the register file, giving single cycle access to a packet received by the NIC. In this case, disaggregation could be a clearer win as it would be equivalent to classical operator fusion in databases that exchanges data between operations using registers [130].

This brief analysis make several assumptions that future work will need to reconsider. It is biased towards disaggregation because it assumes the network has infinite capacity and extra devices, and that connectivity is free. It doesn't consider data distribution: some aggregations or joins require no cache misses because their key-space is small enough to fit into a processors cache. It doesn't consider selectivity of the operations: some operations reduce the amount of data to be transmitted and so localizing can reduce the expected output network cost. It assumes that the bottleneck is the overall pipeline throughput, and not a particular operation. If one operation bottlenecks the pipeline, localization or disaggregation will not help without first relieving the bottleneck via acceleration, scaling out the operation using a technique like DS2 [91], or even using integer linear programming to perform the optimal assignment of operations to devices to balance load [102]. Future work should build more thorough models to answer this question and many more that require modelling the behaviour of the processing pipeline, but this example shows how the assumption of a staged network pipeline enables us to perform such reasoning.

To the playground! – I expect deciding if network connected acceleration is worthwhile to be an evergreen question as networks continue to get faster, devices become more optimized for network communication, and accelerators become more mainstream. Considering the large number of complex factors in play and how costs such as bandwidth, physical connections, rack space, etc. can often be context-specific and completely opaque to academia, a concrete project that would be worthwhile is an *Accelerator Playground* that allows users to explore this question as they change parameters to reflect their particular environments. This playground would allow users to see the performance and costs of different accelerators assuming a given workload, and allow them to tune the model and understand

latency.

the outcome using their own cost functions. As future trends change the equation, the playground could continue to keep users apprised of opportunities for network connected acceleration in their workloads.

More Future Work and Question for Jumpgate

How should Jumpgate integrate SmartNICS? SmartNICs are network interface cards with programmable components such as FPGAs or even general purpose cores [117]. SmartNICs provide a unique combination of low latency host and network communication, and can help bridge the gap between the two. SmartNICs can also help to implement many more analytics operations since they are not as limited as today's programmable switches. The interesting questions for future work to investigate here are all the potential roles of SmartNICs. There are three potential roles SmartNICs could excel at: *Storage acceleration*, where data is parsed from an analytics format and written to the network quickly. Here, data from storage could be passed to the SmartNIC through host memory. *In-network acceleration*, where NTF data is received and processed over the network. Here, a host may not be required whatsoever unless extra memory is needed. *Receive-side acceleration*, in some cases we saw that Spark could not receive a large volume of NTF data quickly. SmartNICs could help to alleviate this receive bottleneck and write data directly into the address space of the receiving process. The answers to these questions will depend on how well the programmable units of the SmartNICs are able to deal with the challenges presented by each role. Future work should attempt implementations of storage parsing, joins, aggregations, and NTF receive to determine their suitability to each. One side-affect of this work is the potential to reconsider the tuple-oriented NTF format that was required to use programmable switches, since SmartNICs could process data in larger blocks, perhaps similarly to ORC's row-group format. Again, Jumpgate can help provide an evaluation framework for this work and can be adapted to generate a new form of NTF if needed.

How should Jumpgate handle network topologies? Jumpgate NCAs communicate at the transport level so they do not need to be topology aware and this eliminates the need for accelerators to be on the data-path between storage and client systems. But, topology-aware placement and scheduling still matters, especially since the bandwidth requirements of queries can be high when all operations are disaggregated. Work such as network-aware virtual data center allocation [34] could be applied to allocate a set of NCAs in a way that optimizes for bandwidth and minimizes latency between communicating NCAs and the final host system. Future work here would need to look at how to trade off allocation overhead for good performance. Since Jumpgate is designed for large jobs, improvements in throughput could help offset the time needed to compute an ideal allocation.

How should analytics systems be re-designed for Jumpgate? In this work, one goal was to integrate NCAs with existing analytics systems with minimal code changes to the existing analytics system. On the other hand, it could be beneficial to make further changes to existing systems to make them more amenable to use NPaaS systems like Jumpgate. One example is designing the analytics system to receive streaming data. Since Spark was designed as a batch processing system, many parts of Spark are built to assume that buffers of intermediate data are always available to be read on demand. This required our Spark integration to buffer data from Jumpgate before allowing Spark to continue execution. While we found this was often equivalent to Spark's original query plan for these operations, it is possible to do better here, if Spark's execution were to be adapted to receive streaming data. Presto [169] is designed to receive streaming data from other nodes, and so we expect it to potentially be more amenable to receiving data from Jumpgate.

Combining Jumpgate and KeyChain Jumpgate and KeyChain are complimentary techniques. Jumpgate improves performance by accelerating queries, and KeyChain improves performance by avoiding re-executing portions of queries. Modern databases use similar techniques to improve query execution time. In the future, it would be beneficial to attempt to integrate support for materialization of sub-queries into Jumpgate. NCAs could direct results towards storage devices to be stored for re-use future queries. KeyChain and other techniques that select subqueries to materialize (§4.6) could be applied to detect the best subqueries to re-use.

6.1.2 KeyChain

KeyChain's lasting contribution will be to shed light on the usefulness of low overhead techniques that can be enabled without knowledge of whether data sharing occurs. These techniques enable users to work faster and be more productive, without requiring them to constantly coordinate with their team while exploring data-sets. Detecting semantically equivalent queries was shown to be challenging when using the JVM because of it's lack of optimization passes. As future analytics frameworks emerge that eschew the JVM for performance of native code generation [52], the techniques presented by KeyChain will become even more effective.

Future work for KeyChain should improve the detection of semantically equivalent code. One angle is to improve the detector by adding better canonicalization and analysis passes. However, the low overhead requirement needs to kept in mind, and proof-based models may continue to not have the low-overhead required, though new developments could make these more feasible. Another angle could be to improve the user-interface around exploring available data-sets and deriving new ones. Collaborative query editing environments could be used to enable analysts to work more closely together, rather than 'heads-down' and separated. For instance, users could be presented with existing materialized data-sets that are not exact matches for their program, but are closer to the users request. The collaborative editing environment could encourage users to curate materialized data-sets together to help get their work done.

Bibliography

[1] AWS Lambda: Run code without thinking about servers. https://aws.amazon.com/lambda/. → page 8

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, . → page 3

[3] GCC Internals. https://gcc.gnu.org/onlinedocs/gccint/, . → page 95

[4] GCC Instruction Patterns. https://gcc.gnu.org/onlinedocs/gccint/Patterns.html#Patterns, . [Online; accessed 12-June-2020]. → page 95

[5] GCC RTL Representation. https://gcc.gnu.org/onlinedocs/gccint/RTL.html#RTL, . [Online; accessed 12-June-2020]. → page 95

[6] MemSQL. https://www.memsql.com/. → page 90

[7] The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html. → page 90

[8] Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. URL https://www.usenix.org/conference/atc20/presentation/tsai. → page 25

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale

machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org. → pages 7, 13, 93, 94

[10] S. Achour and M. Rinard. Noise-aware dynamical system compilation for analog devices with legno. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 20, page 149166, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi:10.1145/3373376.3378449. URL https://doi.org/10.1145/3373376.3378449. → page 94

[11] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*, 2020. → page 27

[12] S. R. Agrawal, S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal, and E. Sedlar. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 245–258, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4952-9. doi:10.1145/3123939.3123985. URL http://doi.acm.org/10.1145/3123939.3123985. → pages 7, 9, 95

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015. → page 13

[14] AL Danial. cloc Github repository. https://github.com/AlDanial/cloc. → page 45

[15] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: Incorporating long messages into the logp modelone step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, page 95105, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917170. doi:10.1145/215399.215427. URL https://doi.org/10.1145/215399.215427. → page 99

[16] Alluxio Open Foundation. Allluxio: Open Source Memory Speed Virtual Distributed Storage. http://www.alluxio.org/. → pages 76, 87

[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228326. → pages 66, 88

[18] S. Angel, M. Nanavati, and S. Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. URL https://www.usenix.org/conference/hotcloud20/presentation/angel. → page 59

[19] Apache ORC. ORC Specification v1. https://orc.apache.org/specification/ORCv1/. → pages 12, 26

[20] Apache Software Foundation. Apache Arrow. http://arrow.apache.org/, . → pages 12, 26

[21] Apache Software Foundation. Flink. https://flink.apache.org/, . → page 25

[22] Apache Software Foundation. Hadoop. http://hadoop.apache.org/, . → pages 25, 76

[23] Apache Software Foundation. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, . [Online; accessed 10-April-2017]. → page 13

[24] Apache Software Foundation. Apache OpenWhisk. https://openwhisk.apache.org/, . → page 27

[25] Apache Software Foundation. Apache ORC Core C++. https://orc.apache.org/docs/core-cpp.html, . → page 42

[26] Apache Software Foundation. Apache Parquet. http://parquet.apache.org/, . → pages 12, 26, 77

[27] Apache Software Foundation. Apache Spark. http://spark.apache.org/, . → pages 6, 24, 25

[28] Apache Software Foundation. Apache Spark: JSON Files. https://spark.apache.org/docs/latest/sql-data-sources-json.html, . [Online; accessed 18-Dec-2018]. → page 13

[29] Apache Software Foundation. Zeppelin: A web-based notebook that enables interactive data analytics. https://zeppelin.apache.org/, . → pages 76, 79

[30] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015. → pages 21, 45, 90

[31] C. Balkesen, N. Kunal, G. Giannikis, P. Fender, S. Sundara, F. Schmidt, J. Wen, S. Agrawal, A. Raghavan, V. Varadarajan, A. Viswanathan, B. Chandrasekaran, S. Idicula, N. Agarwal, and E. Sedlar. Rapid: In-memory analytical query processing engine with extreme performance per watt. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1407–1419, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4703-7. doi:10.1145/3183713.3190655. URL http://doi.acm.org/10.1145/3183713.3190655. → pages 9, 95

[32] Barefoot Networks. Barefoot Tofino Switches. https://www.barefootnetworks.com/products/brief-tofino-2/. → page 56

[33] E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt. Virtual machine warmup blows hot and cold. *CoRR*, abs/1602.00602, 2016. URL http://arxiv.org/abs/1602.00602. → pages 77, 86

[34] S. Bayless, N. Kodirov, I. Beschastnikh, H. H. Hoos, and A. J. Hu. Scalable constraint-based virtual data center allocation. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 546–554. AAAI Press, 2017. ISBN 978-0-9992411-0-3. URL http://dl.acm.org/citation.cfm?id=3171642.3171722. → page 103

[35] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM*, 1999. → page 58

[36] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York,

NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9.
doi:10.1145/2038916.2038923. URL
http://doi.acm.org/10.1145/2038916.2038923. → pages 65, 67, 87

[37] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013. → pages 5, 7, 12, 21, 46

[38] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. ISSN 0146-4833. doi:10.1145/2656877.2656890. URL http://doi.acm.org/10.1145/2656877.2656890. → pages 7, 11, 21, 24, 55

[39] Cavium. Liquidio ii network appliance smart nics. https://www.cavium.com/liquidio-II-network-appliance-adapters.html. → page 5

[40] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL http://dl.acm.org/citation.cfm?id=3291168.3291211. → page 91

[41] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356. ACM, 2018. → page 11

[42] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. DRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017. → pages 21, 24

[43] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: Proving query rewrites with univalent sql semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 510–524, New York, NY, USA, 2017.

ACM. ISBN 978-1-4503-4988-8. doi:10.1145/3062341.3062348. URL http://doi.acm.org/10.1145/3062341.3062348. → page 88

[44] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, page 112, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915895. doi:10.1145/155332.155333. URL https://doi.org/10.1145/155332.155333. → pages 99, 100

[45] W. J. Dally, Y. Turakhia, and S. Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):4857, June 2020. ISSN 0001-0782. doi:10.1145/3361682. URL https://doi.org/10.1145/3361682. → page 1

[46] Databricks. Apache Spark as a Compiler. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html, 2016. → page 10

[47] Databricks. Spark SQL Performance Tests. https://github.com/databricks/spark-sql-perf, 2018. → pages 44, 77

[48] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. → pages 67, 68, 86

[49] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi:10.1145/2463676.2465295. URL http://doi.acm.org/10.1145/2463676.2465295. → pages 9, 25

[50] DPDK Project. Data plane development kit (dpdk). https://www.dpdk.org/. → pages 5, 26, 58

[51] Drew Paroski. Code Generation: The Inner Sanctum Of Database Performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html. → page 90

[52] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing Apache Spark for Scale-Up Architectures and Medium-Size Data. In *OSDI*, 2018. → pages 57, 104

[53] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Native compilation for heterogeneous workloads in apache spark. *CoRR*, abs/1703.08219, 2017. URL http://arxiv.org/abs/1703.08219. → pages 81, 95

[54] Facebook Inc. Bryce Canyon Storage Specification. https://www.opencompute.org/contributions?query=bryce%20canyon%20spec, . [Online; accessed 20-Dec-2018]. → page 8

[55] Facebook Inc. Mono Lake Server Specification. https://www.opencompute.org/contributions?query=mono%20lake, . [Online; accessed 20-Dec-2018]. → page 8

[56] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. Udp: A programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 55–68, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4952-9. doi:10.1145/3123939.3123983. URL http://doi.acm.org/10.1145/3123939.3123983. → page 12

[57] Y. Fang, C. Zou, and A. A. Chien. Accelerating raw data analysis with the accorda software and hardware architecture. *Proc. VLDB Endow.*, 12(11): 15681582, July 2019. ISSN 2150-8097. doi:10.14778/3342263.3342634. URL https://doi.org/10.14778/3342263.3342634. → page 60

[58] D. Firestone. Vfp: A virtual switch platform for host sdn in the public cloud. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 315–328, Berkeley, CA, USA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL http://dl.acm.org/citation.cfm?id=3154630.3154656. → page 14

[59] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association. ISBN 978-1-931971-43-0. URL https://www.usenix.org/conference/nsdi18/presentation/firestone. → pages 5, 7

[60] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/fouladi. → page 95

[61] GCC, the GNU Compiler Collection. Canonicalization of Instructions. https://gcc.gnu.org/onlinedocs/gccint/Insn-Canonicalizations.html, . → page 4

[62] GCC, the GNU Compiler Collection. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, . → page 80

[63] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2009. → page 54

[64] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*, page 2. ACM, 2015. → pages 13, 24

[65] V. Govindaraju, S. Idicula, S. Agrawal, V. Vardarajan, A. Raghavan, J. Wen, C. Balkesen, G. Giannikis, N. Agarwal, and E. Sedlar. Big data processing: Scalability with extreme single-node performance. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 129–136, June 2017. doi:10.1109/BigDataCongress.2017.26. → pages 9, 95

[66] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, COM-HPC '16, pages 1–10, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-5090-3829-9. doi:10.1109/COM-HPC.2016.6. URL https://doi.org/10.1109/COM-HPC.2016.6. → pages 6, 10, 14, 24

[67] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for

near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8947-1. doi:10.1109/ISCA.2016.23. URL https://doi.org/10.1109/ISCA.2016.23. → pages 9, 25

[68] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924949. → pages 64, 66, 67, 87

[69] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 357–371, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5567-4. doi:10.1145/3230543.3230555. URL http://doi.acm.org/10.1145/3230543.3230555. → pages 7, 10, 21, 24

[70] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back, 2018. → page 8

[71] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X. → pages 49, 99

[72] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):4860, Jan. 2019. ISSN 0001-0782. doi:10.1145/3282307. URL https://doi.org/10.1145/3282307. → page 1

[73] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011. → page 27

[74] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet

processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 54–66, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6080-7. doi:10.1145/3281411.3281443. URL http://doi.acm.org/10.1145/3281411.3281443. → pages 5, 58

[75] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 349362, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307598. doi:10.1145/2150976.2151013. URL https://doi.org/10.1145/2150976.2151013. → page 93

[76] M. Houshmand and S. Paydar. *TCE+: An Extension of the TCE Method for Detecting Equivalent Mutants in Java Programs*, pages 164–179. Springer International Publishing, Cham, 2017. ISBN 978-3-319-68972-2. doi:10.1007/978-3-319-68972-2_11. URL https://doi.org/10.1007/978-3-319-68972-2_11. → pages 67, 80, 88

[77] Y. Huai. A Deep Dive into Spark SQL's Catalyst Optimizer. https://www.slideshare.net/databricks/a-deep-dive-into-spark-sqls-catalyst-optimizer-with-yin-huai. → page 68

[78] Ian Ward. Documentation for the JSON Lines text file format. http://jsonlines.org/. [Online; accessed 18-Dec-2018]. → page 13

[79] S. Ibanez, M. Shahbaz, and N. McKeown. The case for a network fast path to the cpu. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets 19, page 5259, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370202. doi:10.1145/3365609.3365851. URL https://doi.org/10.1145/3365609.3365851. → pages 24, 101

[80] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *Proc. VLDB Endow.*, 9(3):216–227, Nov. 2015. ISSN 2150-8097. doi:10.14778/2850583.2850595. URL http://dx.doi.org/10.14778/2850583.2850595. → page 15

[81] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS*

operating systems review, volume 41, pages 59–72. ACM, 2007. → pages 13, 25

[82] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, Aug. 2017. ISSN 2150-8097. doi:10.14778/3137628.3137632. URL https://doi.org/10.14778/3137628.3137632. → pages 9, 25

[83] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 281–293, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi:10.1145/2882903.2882957. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/2882903.2882957. → page 64

[84] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, pages 21–28, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6710-3. doi:10.1145/3314148.3314356. URL http://doi.acm.org/10.1145/3314148.3314356. → pages 19, 21, 26

[85] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi:10.1145/3132747.3132764. URL http://doi.acm.org/10.1145/3132747.3132764. → pages 6, 7, 24, 26, 58

[86] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association. ISBN 978-1-931971-43-0. URL https://www.usenix.org/conference/nsdi18/presentation/jin. → pages 6, 7, 24

[87] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proc. VLDB Endow.*, 11(7):800–812, Mar. 2018. ISSN 2150-8097. doi:10.14778/3192965.3192971. → pages 64, 66, 87, 88

[88] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong. Yoursql: A high-performance database system leveraging

in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, Aug. 2016. ISSN 2150-8097. doi:10.14778/2994509.2994512. URL http://dx.doi.org/10.14778/2994509.2994512. → pages 9, 25

[89] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5028-0. doi:10.1145/3127479.3128601. URL http://doi.acm.org/10.1145/3127479.3128601. → page 8

[90] Julien Le Dem. The striping and assembly algorithms from the Dremel paper. https://github.com/julienledem/redelm/wiki/The-striping-and-assembly-algorithms-from-the-Dremel-paper. [Online; accessed 18-Dec-2018]. → pages 12, 13

[91] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. 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*, OSDI18, page 783798, USA, 2018. USENIX Association. ISBN 9781931971478. → pages 42, 43, 49, 101

[92] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi:10.1145/2872362.2872367. URL http://doi.acm.org/10.1145/2872362.2872367. → page 25

[93] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):22092222, Sept. 2018. ISSN 2150-8097. → page 90

[94] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL

https://www.usenix.org/conference/osdi18/presentation/khawaja. → page 7

[95] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering*, 2017. → pages 67, 80, 88

[96] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4240-7. doi:10.1145/2901318.2901337. URL http://doi.acm.org/10.1145/2901318.2901337. → page 8

[97] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/klimovic-serverless. → pages 8, 60

[98] E. Kohler, M. Handley, and S. Floyd. Designing dccp: Congestion control without reliability. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 27–38, New York, NY, USA, 2006. ACM. ISBN 1-59593-308-5. doi:10.1145/1159913.1159918. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/1159913.1159918. → page 12

[99] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4952-9. doi:10.1145/3123939.3124553. URL http://doi.acm.org/10.1145/3123939.3124553. → pages 9, 25

[100] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/. → page 27

[101] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114124, June 2008. ISSN

0362-1340. doi:10.1145/1379022.1375596. URL https://doi.org/10.1145/1379022.1375596. → page 99

[102] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114124, June 2008. ISSN 0362-1340. doi:10.1145/1379022.1375596. URL https://doi.org/10.1145/1379022.1375596. → pages 99, 101

[103] M. Kunjir, B. Fain, K. Munagala, and S. Babu. Robus: Fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 219–234, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi:10.1145/3035918.3064018. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/3035918.3064018. → pages 65, 66, 88

[104] G. Langdale and D. Lemire. Parsing gigabytes of JSON per second. *CoRR*, abs/1902.08318, 2019. URL http://arxiv.org/abs/1902.08318. → pages 10, 12, 25, 26, 42, 54

[105] J. Larus and G. Hunt. The singularity system. *Commun. ACM*, 53(8): 72–79, Aug. 2010. ISSN 0001-0782. doi:10.1145/1787234.1787253. URL http://doi.acm.org/10.1145/1787234.1787253. → page 14

[106] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673. → page 3

[107] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, Z. Zhang, and J. Sukha. On-the-fly pipeline parallelism. *ACM Trans. Parallel Comput.*, 2(3), Sept. 2015. ISSN 2329-4949. doi:10.1145/2809808. URL https://doi-org.ezproxy.library.ubc.ca/10.1145/2809808. → pages 49, 99

[108] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Miso: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1591–1602. ACM, 2014. → pages 65, 66, 88

[109] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 851–862. ACM, 2014. → pages 65, 66, 88

[110] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204215, Nov. 2015. ISSN 2150-8097. doi:10.14778/2850583.2850594. URL https://doi.org/10.14778/2850583.2850594. → page 90

[111] A. Lerner, R. Hussein, and P. Cudre-Mauroux. The Case for Network-Accelerated Query Processing. CIDR 2019, 2019. → pages 6, 7, 8, 10, 11, 12, 14, 19, 21, 24, 26, 27, 34, 56, 99

[112] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. doi:10.1145/2670979.2670985. URL http://doi.acm.org/10.1145/2670979.2670985. → page 87

[113] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 467–483, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026914. → pages 6, 7, 24

[114] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi:10.1145/3132747.3132751. URL http://doi.acm.org/10.1145/3132747.3132751. → pages 6, 7

[115] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast json parser for data analytics. *Proc. VLDB Endow.*, 10(10):1118–1129, June 2017. ISSN 2150-8097. doi:10.14778/3115404.3115416. URL https://doi.org/10.14778/3115404.3115416. → pages 10, 12, 26

[116] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm

warm-up overhead in data-parallel systems. In *OSDI*, pages 383–400, 2016. → pages 77, 86

[117] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5956-6. doi:10.1145/3341302.3342079. URL http://doi.acm.org/10.1145/3341302.3342079. → pages 25, 102

[118] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/liu-ming. → page 25

[119] LLVM Project. Instcombine. https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions, . → page 4

[120] LLVM Project. LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html, . → pages 80, 82

[121] Luigi Rizzo. netmap - the fast packet I/O framework. http://info.iet.unipi.it/~luigi/netmap/. → page 58

[122] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 1–1, Berkeley, CA, USA, 2015. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2831090.2831091. → page 86

[123] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, Dec. 2002. ISSN 0163-5980. doi:10.1145/844128.844142. URL http://doi.acm.org/10.1145/844128.844142. → pages 6, 10

[124] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*,

CoNEXT ’14, pages 249–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3279-8. doi:10.1145/2674005.2674996. URL http://doi.acm.org/10.1145/2674005.2674996. → pages 6, 10, 14

[125] Mario Baldi. Exposing Data Plane Programmability - Mario Baldi. https://www.inf.usi.ch/faculty/soule/baldi.pdf. [Online; accessed 16-Jan-2019]. → page 14

[126] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *SIGARCH Comput. Archit. News*, 25(2):8597, May 1997. ISSN 0163-5964. doi:10.1145/384286.264146. URL https://doi.org/10.1145/384286.264146. → page 99

[127] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, 2015. → page 8

[128] Mellanox Technologies. Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). http://www.mellanox.com/page/products_dyn?product_family=261&mtag=sharp. [Online; accessed 7-Jan-2019]. → page 10

[129] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010. ISSN 2150-8097. doi:10.14778/1920841.1920886. URL http://dx.doi.org/10.14778/1920841.1920886. → pages 12, 26

[130] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):113, Sept. 2017. ISSN 2150-8097. doi:10.14778/3151113.3151114. URL https://doi.org/10.14778/3151113.3151114. → pages 58, 90, 101

[131] D. MICHIE. Memo functions and machine learning. *Nature*, 218(5136): 19–22, 04 1968. URL http://dx.doi.org/10.1038/218019a0. → pages 87, 91

[132] J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI’85, pages 165–172, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. ISBN 0-934613-02-8. URL http://dl.acm.org/citation.cfm?id=1625135.1625165. → pages 87, 91

[133] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522738. URL http://doi.acm.org/10.1145/2517349.2522738. → pages 13, 24

[134] C. Mustard and A. Fedorova. Practical cross program memoization with keychain. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 262–271, 2018. → page 63

[135] C. Mustard, F. Ruffy, A. Gakhokidze, I. Beschastnikh, and A. Fedorova. Jumpgate: In-network processing as a service for data analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, 2019. USENIX Association. → page 5

[136] R. O. Nambiar and M. Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006. → pages 44, 67, 77

[137] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, pages 17–33, 2017. → page 8

[138] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 85–98, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4653-5. doi:10.1145/3098822.3098829. URL http://doi.acm.org/10.1145/3098822.3098829. → pages 7, 10

[139] Netronome. Smartnics overview. https://www.netronome.com/products/smartnic/overview/. → page 5

[140] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. ISSN 2150-8097. doi:10.14778/2002938.2002940. URL http://dx.doi.org/10.14778/2002938.2002940. → pages 27, 58, 81, 90, 92

[141] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 56–69, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6.

doi:10.1145/3173162.3173200. URL
http://doi.acm.org/10.1145/3173162.3173200. → page 76

[142] Oracle. java.lang.Class API Documentation.
https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html. → page 74

[143] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL
http://dl.acm.org/citation.cfm?id=2789770.2789791. → pages 7, 76

[144] OW2 Consortium. Asm. http://asm.ow2.org/, 2009. [Online; accessed 14-April-2017]. → page 74

[145] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi:10.1145/2815400.2815423. URL
http://doi.acm.org/10.1145/2815400.2815423. → page 6

[146] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017. → pages 13, 14, 24, 93

[147] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment*, 11(11), 2018. → pages 12, 26

[148] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *Proc. VLDB Endow.*, 11(9):1002–1015, May 2018. ISSN 2150-8097. doi:10.14778/3213880.3213890. URL
https://doi-org.ezproxy.library.ubc.ca/10.14778/3213880.3213890. → pages 13, 14, 24, 93

[149] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16,

pages 203–216, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026894. → pages 6, 14, 100

[150] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 936–946, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818867. → pages 67, 80, 81, 88

[151] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. 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, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/phothilimthana. → pages 11, 25, 27

[152] H. Pirk, J. Giceva, and P. R. Pietzuch. Thriving in the no man's land between compilers and databases. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL http://cidrdb.org/cidr2019/papers/p91-pirk-cidr19.pdf. → page 91

[153] M. Poess, B. Smith, L. Kollar, and P. Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 582–587, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi:10.1145/564691.564759. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/564691.564759. → pages 44, 67, 77

[154] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 531–548, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-49-2. URL

https://www.usenix.org/conference/nsdi19/presentation/pontarelli. → pages 7, 27

[155] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 209–215, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6727-1. doi:10.1145/3317550.3321439. URL http://doi.acm.org/10.1145/3317550.3321439. → page 24

[156] Project Jupyer. Jupyter Notebook. http://jupyter.org/. → pages 76, 79

[157] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi:10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176. → pages 91, 93

[158] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1): 106115, Dec. 2017. ISSN 0001-0782. doi:10.1145/3150211. URL https://doi.org/10.1145/3150211. → page 91

[159] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC 12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450317610. doi:10.1145/2391229.2391242. URL https://doi.org/10.1145/2391229.2391242. → page 27

[160] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment*, 6(10):853–864, 2013. → page 64

[161] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi:10.1145/1868294.1868314. URL http://doi.acm.org/10.1145/1868294.1868314. → page 95

[162] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011. → page 7

[163] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522715. URL http://doi.acm.org/10.1145/2517349.2522715. → pages 14, 95

[164] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan. Don't trash your intermediate results, cache 'em. *CoRR*, cs.DB/0003005, 2000. URL http://arxiv.org/abs/cs.DB/0003005. → pages 65, 66, 88

[165] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi:10.1145/2884781.2884877. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/2884781.2884877. → pages 80, 88

[166] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5569-8. doi:10.1145/3152434.3152461. URL http://doi.acm.org/10.1145/3152434.3152461. → pages 6, 7, 10, 11, 12, 24, 26, 27, 46, 99

[167] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *CoRR*, abs/1903.06701, 2019. URL http://arxiv.org/abs/1903.06701. → pages 24, 26, 55, 58, 62

[168] O. Sefraoui, M. Aissaoui, and M. Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012. → page 27

[169] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, April 2019. doi:10.1109/ICDE.2019.00196. → pages 21, 25, 27, 42, 57, 103

[170] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/shan. → pages 25, 59

[171] N. K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 67–82, Berkeley, CA, USA, 2017. USENIX Association. ISBN 978-1-931971-37-9. URL http://dl.acm.org/citation.cfm?id=3154630.3154637. → pages 7, 12, 46

[172] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 15–28, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4193-6. doi:10.1145/2934872.2934900. URL http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/2934872.2934900. → pages 7, 27

[173] B. Stephens, A. Akella, and M. M. Swift. Your programmable nic should be a programmable switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 36–42. ACM, 2018. → page 5

[174] R. Stewart. Stream control transmission protocol, rfc 4960. Technical report, IETF, 2007. → page 12

[175] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, SIGCOMM Posters and Demos '19, pages 72–74, New York, NY, USA, 2019. ACM. ISBN

978-1-4503-6886-5. doi:10.1145/3342280.3342311. URL http://doi.acm.org/10.1145/3342280.3342311. → pages 19, 24, 25, 27, 34, 41, 56, 58, 62, 99

[176] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating Database Queries with Switch Pruning. *SIGMOD*, 2020. doi:10.1145/3342280.3342311. URL https://arxiv.org/abs/2004.05076. → pages 21, 25, 26

[177] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Proceedings of the USSR Academy of Sciences*, 70: 569–572, 1950. → pages 65, 88

[178] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler. Albis: High-performance file format for big data systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 615–630, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/trivedi. → pages 12, 26, 60

[179] T. VanDrunen and A. L. Hosking. Value-based partial redundancy elimination. In E. Duesterwald, editor, *Compiler Construction*, pages 167–184, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24723-4. → page 88

[180] S. D. Viglas. Just-in-time compilation for sql query processing. *Proc. VLDB Endow.*, 6(11):11901191, Aug. 2013. ISSN 2150-8097. doi:10.14778/2536222.2536254. URL https://doi.org/10.14778/2536222.2536254. → page 90

[181] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys 19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi:10.1145/3302424.3303982. URL https://doi.org/10.1145/3302424.3303982. → page 95

[182] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-13-7. URL

https://www.usenix.org/conference/nsdi20/presentation/vuppalapati. → page 61

[183] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006. → page 13

[184] Wikipedia. Tilera. https://en.wikipedia.org/wiki/Tilera. → page 5

[185] Wikipedia. Pseudorandom number generator. https://en.wikipedia.org/wiki/Pseudorandom_number_generator, 2018. [Online; accessed 9-May-2018]. → page 72

[186] Wikipedia. Universally unique identifier. https://en.wikipedia.org/wiki/Universally_unique_identifier, 2018. [Online; accessed 11-April-2018]. → page 74

[187] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 255–268, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi:10.1145/2541940.2541961. URL http://doi.acm.org/10.1145/2541940.2541961. → page 9

[188] E. Xu, M. Saxena, and L. Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *HotStorage*, 2016. → pages 66, 76, 88

[189] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, et al. Bluecache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment*, 10(4):301–312, 2016. → page 25

[190] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855744. → page 86

[191] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings*

of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2228298.2228301. → pages 7, 13, 14, 21, 67, 93, 95

[192] Q. Zhang, Y. Cai, S. Angel, V. Liu, A. Chen, and B. T. Loo. Rethinking data management systems for disaggregated data centers. In *CIDR*, 2020. → page 59

[193] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):15681581, May 2020. ISSN 2150-8097. doi:10.14778/3397230.3397249. URL https://doi.org/10.14778/3397230.3397249. → page 59

[194] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*, pages 26–31. ACM, 2016. → page 6

[195] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proc. VLDB Endow.*, 13(3):376389, Nov. 2019. ISSN 2150-8097. doi:10.14778/3368289.3368301. URL https://doi.org/10.14778/3368289.3368301. → pages 24, 26, 58, 62

[196] N. Zilberman, M. P. Grosvenor, D. A. Popescu, N. M. Bojan, G. Antichi, M. Wójcik, and A. W. Moore. Where has my time gone? In M. A. Kâafar, S. Uhlig, and J. Amann, editors, *Passive and Active Measurement - 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings*, volume 10176 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2017. doi:10.1007/978-3-319-54328-4_15. URL https://doi.org/10.1007/978-3-319-54328-4_15. → page 100

[197] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB 07, page 723734. VLDB Endowment, 2007. ISBN 9781595936493. → page 60

Appendix A

Supporting Materials

A.1 Appendix

This appendix details the key components in Jumpgate. We use Python-like pseudocode to illustrate these algorithms. (Figures A.2, A.3, A.4, A.5, A.7). Jumpgate is actually implemented in Python. This is not a performance concern because Jumpgate acts as a control plane to coordinate the computation over the data and is not on the data transfer path or the compute path. The example code matches the running example in chapter 3, corresponding to Figure 3.2.

A.1.1 Client Interface Example

Figure A.1 shows the JSON request the client would generate from the functional-style example shown in Figure 3.4. Each analytics operation is represented as a dictionary object, describing its kind, name, connectivity (`input_channels`), and any operation-specific parameters. The main difference between this representation and the functional-style example is how the client must explicitly name each column produced. This is important so that neither to client or Jumpgate system needs to guess the names of columns to use in subsequent

A.1.2 Operator Interface Example

Figure A.2 shows an example of the operator interface for the software NCA that matches against the read, filter, and project operations (NCA1 and NCA2) shown in the overview (Figure 3.2). This NCA can read JSON, filter and project it, and output a network tuple format over UDP or TCP.

Jumpgate checks that the NCA can implement the desired logical operation(s) (`match_operations`), send and receive data in the given format and transport (`match_input`, `match_output`). Also, there is a deployment manager that can be used to allocate the device (`match_deployment`). `match_output` is called with the *desired* output format from the sender, and returns the NTF that the NCA would send, based on the operations being performed.

The operator interface gives NCA implementors the freedom to implement specific matching rules, depending on how flexible their target hardware is. For instance, `match_operations` is free to inspect all expressions used in the operations, to determine if it can implement each one.

A.1.3 Operator Life-cycle Example

Figure A.3 shows an example of the life-cycle interface for NCA3 from Figure 3.2 that implements join. The implementation of the life-cycle interfaces for this NCA communicates with a running instance over stdin/stdout. `compile` configures the NCA for the query to be executed. In the example, it compiles a C program specialized with the join expression and output schema. Jumpgate operators communicate at the network transport level. Jumpgate calls `allocate` to get the host and port that the NCA will receive data on, which is known after the binary is executed and allocates a port. Jumpgate calls `configure` to tell the NCA where to send data to: other NCAs or the client endpoints. Jumpgate calls `execute` when the operator should run (after all operators have been configured), and Jumpgate supplies a callback to be notified when the NCA data-plane has processed all incoming data for this stage. In the example, since Join executes in two stages, execute will be called twice. Once for the first build stage, and once again for the probe stage.

A.1.4 Staging Algorithm

Figure A.4 illustrates Jumpgate's staging algorithm. The algorithm finds the set of stages required to ensure input to `Join` NCAs is properly ordered (e.g., build input is done before probe input starts), and that when an NCA will output data, there is another NCA that runs concurrently to receive the data. `compute_stages` is given a client request, and iterates over the operators in the request that are not yet completed. When operators are executed, the stage they executed during is saved. `can_run` is called to determine if an operation is currently runnable. Most operations can run if their inputs and output NCAs can run. Join can run in its build phase if its build input can run, and can run in its probe phase if its probe input and outputs can run. `execute` is called to mark an operation as executed in the given stage,

A.1.5 Dataflow to NCA Mapping Algorithm

Figure A.7 gives the pseudo-code for mapping logical operations from the dataflow request to available NCA implementations. Jumpgate iteratively transforms a dataflow request into a graph of NCA instances by repeatedly picking an operation to replace, and calling each NCA's operator interface. `map_request_to_ncas` is called with a graph of logical operations, picks an operation to replace and iterates over all NCAs calling `replace_op` to to check if the operation can be replaced. `replace_op` calls the *operator interface* on the given NCA to determine if it is a match.

A.1.6 Generating Network Tuple Formats

Figure A.5 shows Jumpgate's algorithm for generating NTFs. An NCA that wants to generate an NTF calls `generateNTF` with the output schema, and an NTF specification is returned as an array of objects that specify the precise memory layout of the NTF.

Jumpgate generates 8-byte aligned fields for ease of configuring hardware devices. One bit of the null vector is assigned to each field. String (variable-length) fields are represented as an offset and length where the field will appear in the

	BOOL	INT	BIGINT	DOUBLE	STRING
Aggregate	0.0	70.4	13.9	82.4	67.4
Filter	0.2	94.1	6.5	6.1	18.2
Join	0.0	96.8	5.9	0.0	2.8
Projection	0.0	95.4	8.0	41.6	33.5
Shuffle	0.0	58.7	3.8	4.8	55.2

Table A.1: Percent of operations that operate on at least one field of the given data type.

variable length section.

Schemas map from field names to types, but do not impose an ordering on fields. Jumpgate's NTF generation algorithm currently follows the same ordering as given in the schema – but there is no requirement to do so. For NCAs that have very specific output formats, such as our RMT-based aggregator, the NCA designer specifies the NTF by hand and does not need to call `generateNTF`.

A.1.7 Further TPC-DS Characteristics

We present further details on TPC-DS characteristics that are relevant to the design of future NCAs. In §3.5.3 we found that many NTFs were small enough to be processed by NCAs that can only look at a few 100 bytes of each packet. However, we found that $\approx 40\%$ of NTFs included strings. Overall, future NCAs will need to operate on fixed-length binary, string and floating point values. Figure A.6 shows the *number* of strings per NTF. Table A.1 shows the field types that each operations references. Many operations reference multiple strings and floating point values. Doubles appear because `spark-sql-perf` was configured to emit doubles, but the official TPC-DS specification is for fixed point decimal numbers. Joins operate mainly on integers, but may need to output variable length data. Aggregations use strings and integers as grouping keys, but mostly aggregate double values. Shuffles use the grouping key of aggregations, which is often strings.

```
{ "kind": "DataSource", "name": "read_sales"
  "file_path": "/path/to/sales",
  "schema": "`item_id` INT, `store_id` INT, `price` DOUBLE"
},
{ "kind": "Filter", "name": "filter_sales",
  "input_channel": "read_sales",
  "filter_expression": "item_id == 100"
},
{ "kind": "Project", "name": "project_sales"
  "input_channel": "",
  "projection_expressions": ["store_id", "price"],
  "schema": "`store_id` INT, `price` DOUBLE"
},

{ "kind": "DataSource", "name": "read_stores"
  "file_path": "/path/to/stores",
  "schema": "`store_id` INT, `state` STRING"
},
{ "kind": "Project", "name": "project_stores"
  "input_channel": "read_stores",
  "projection_expressions": ["store_id", "price"],
  "schema": "`store_id` INT, `price` DOUBLE"
},

{ "kind": "Join", "name": "join_sales_stores",
  "input_channels": {
    "build": "project_stores",
  "probe": "project_sales"
  },
  "build_keys": ["store_id"]
  "probe_keys": ["store_id"],
  "join_condition": ""
},
{ "kind": "Project", "name": "project_join"
  "input_channel": "join_sales_stores",
  "projection_expressions": ["price", "state"],
  "schema": "`price` DOUBLE, `state` STRING"
},
{ "kind": "Aggregate", "name": "agg",
  "input_channels": "project_join",
  "group_keys": ["state"],
  "group_schema": "`state` STRING",
  "aggregate_expressions": ["sum(price)"],
  "aggregate_schema": "`price` DOUBLE",
  "result_expressions": ["state", "price"],
  "schema": "`state` STRING, `total_price` DOUBLE"
},

{ "kind": "Shuffle", "name": "shuffle",
  "input_channels": "agg",
  "key_expression": "state",
  "num_destinations": 2
}
{ "kind": "Send", "name": "host1"
  "input_channels": "shuffle[0]",
  "transport": "TCP",
  "host": ["10.0.0.1", 1234]
}
{ "kind": "Send", "name": "host2"
  "input_channels": "shuffle[1]",
  "transport": "TCP",
  "host": ["10.0.0.2", 4567]
}
```

Figure A.1: Actual dataflow request generated from example in Figure 3.4.

```
def match_operations(operations):
  # Check if operations are supported.
  # This NCA supports Read operations on JSON, and can fuse
      subsequent Filter and Project
  matched_ops = match_operations(operations,
    [Read, Filter, Project])
  if matched_ops:
    if matched_ops[0].format == JSON:
      return matched_ops

def match_input(transport, format):
  # Check if operator is able to receive data on a given transport
       and a given format.
  # This NCA does not receive from other NCAs, so it checks for
      empty inputs.
  if format is None and transport is None:
    return True

def match_output(ops, transport, format):
  # Check if able to send data in format/transport
  project_schema = ops[2]
  transport_match = transport in [TCP, UDP]
  # check that the receiver accepts NTF
  format_match = (format == NTF)
  if format_match:
    # generate the specific NTF for the output
    ntf = generateNTF(project_schema)
  if transport_match and format_match:
    return transport_match, ntf

def match_deployment():
  # declare that this NCA needs to be deployed on software, with 8
       cores.
  return {"type":"software", "cores":8}
```

Figure A.2: Operator interface API and example for the Software NCA that matches against read-filter-project from Figure 3.2.

```
def compile(operations):
  # generate query specific code
  join = operations[0]
  project = operations[1]

  code = codegen(join.expressions,
      project.output_schema)
  binary = compile(code)
  return binary

def allocate(binary, deployment):
  # start compiled operator and return the host and port it
      receives data on
  process = deployment.start(binary)
  host = process.readline()
  port = process.readline()
  return (host, port)

def configure(destinations):
  # send the host/ports of output destinations to the accelerator
  build = destinations[0]
  probe = destinations[1]
  process.stdin.write(build.host)
  process.stdin.write(build.port)
  process.stdin.write(probe.host)
  process.stdin.write(probe.port)

def execute(stage_num, done_func, stage_num):
  # start sending/receiving data, and call done_func when operator
       has succeeded or failed
  if stage_num == 0:
    process.stdin.write("execute build")
  elif stage_num == 1:
    process.stdin.write("execute probe")
  while data = process.stdout.readline():
    if data == "done":
      done_func()

def destroy():
  # stop and clean up operator, due to failure or completion
  process.terminate()
```

Figure A.3: Operator Life-cycle API example for NCA3 from Figure 3.4 that implements join and project, and communicates with the NCA over stdin/stdout.

```
def can_run(op):
  # determine if an op can send and receive data during this stage
  inputs_ready = all_ready(op.inputs)
  if op.kind == 'join' and op.phase == 'build':
    # joins do not send output during build
    outputs_ready = True
  else:
    outputs_ready = all_ready(op.outputs)
  if inputs_ready and outputs_ready:
    return True
  else:
    return False

def execute(op):
  # simulate execution of the operation
  if op.kind == 'join':
    if op.phase == 'build':
      op.phase = 'probe'
    elif op.phase == 'probe':
      op.done = True
  else:
    op.done = True

  return op.done


def compute_stages(request):
  # create a copy of the array containing the operations in this
      request
  remaining_operations = request.operations.copy()
  stages = []
  stage_num = 0
  while len(remaining_operations) > 0:
    for op in remaining_operators:
      if can_run(op):
        # remember which stages this op ran in
        op.stages.append(stage_num)
        if execute(op):
          remaining_operations.remove(op)
    stage_num += 1
```

Figure A.4: Staging algorithm illustrated in Python-like pseudo-code. The staging algorithm simulates job execution to find the set of operations that need to run concurrently.

```
def generateNTF(schema):
  # every NTF starts with a null vector, large enough to store 1
     bit for each field
  layout = [ {'kind':'nullvector', 'size': 8} ]
  # NTFs need a variable length section if there are strings
  needs_variable_length = False
  # track which null bit to use for each field
  nullidx = 0
  for name, type in schema:
    if current_size(layout) % 8 != 0:
      # insert padding if not 8-byte-aligned
      pad_size = 8 - (current_size(layout) % 8)
      layout.append({'kind': 'padding',
              'size': pad_size})

    if type.variable:
      # Strings (aka variable-length) fields are represented as an
          offset and length that index into the variable length
         section.
      needs_variable_length = True
      layout.append({'kind': 'offset',
              'name': name,
              'size': 4,
              'nullidx': nullidx})
      layout.append({'kind': 'length',
              'name': name,
              'size': 4})
    else:
      # Fixed-length fields are stored verbatim in the NTF
      layout.append({'kind': 'field',
              'name': name,
              'size': type.size,
              'nullidx': nullidx
              })
    nulldx += 1
  # return the NTF specification
  return {
    'layout': layout,
    'total_size': current_size(layout),
    'needs_variable_length': needs_variable_length
  }
```

Figure A.5: Network Tuple Format generation algorithm illustrated in Python-like psuedo-code.

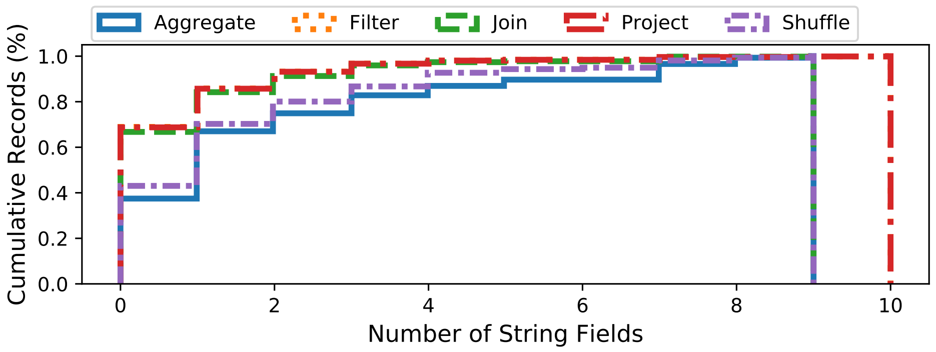


Figure A.6: Cumulative distribution of number of strings in each NTF with variable length sections.

```
def replace_op(op, nca, graph):
  # read input formats and transport from sender
  in_transports, in_formats = get_input(op, graph)
  # get desired input format and transport from receiver
  out_transports, out_formats = get_output(op, graph)

  # check that the NCA matches this operator
  input_match = nca.match_input(in_transports, in_formats)
  output_spec = nca.match_output(op, out_transports, out_formats):
  matched_operations = nca.match_operations(op, graph)

  # check that output of this NCA would be compatible with the
      receivers in the graph
  output_match = graph.get_receiver(nca).match_input(output_spec)

  # return failure if none of the above checks were successful
  if not (input_match and output_match and matched_operations):
    return None

  # match found, replace operations and return new candidate graph
  new_graph = graph.copy()
  graph.replace(matched_operations, nca.new_instance())
  return new_graph

def map_request_to_ncas(graph):
  ops_to_replace = find_logical_nodes(graph)
  all_NCAs = get_all_NCAs()
  while len(ops_to_replace) > 0:
    # pick an operation to replace
    op_to_replace = ops_to_replace[0]
    # track best replacement found for this operation
    best_metric = None
    best_graph = None
    # iterate over all NCAs and check for a match
    for nca in all_NCAs:
      new_graph = replace_op(op_to_replace, nca, best_graph)
      if new_graph:
        new_graph_metric = evaluate_graph(new_graph)
        if best_graph is None or best_metric > new_graph_metric:
          best_graph = new_graph
          best_metric = new_graph_metric

    # pick best graph found to replace the given operation
    graph = best_graph
    # update the operations to be replaced, at least one operation
        is always guaranteed to be replaced, so this will
       terminate.
    ops_to_replace = find_logical_nodes(graph)
```

Figure A.7: Jumpgate's operation to NCA mapping algorithm illustrated in Python-like pseudo-code. The mapping algorithm iteratively replaces nodes in the graph, greedily picking the preferred graph so far.