Flint:
A Distributed Computation Engine (over Named Data Networking)

Jacob Zhi
zhi@cs.ucla.edu

Omar Elamri
omar@cs.ucla.edu

Paul Serafimescu
pserafim@cs.ucla.edu

Abstract

Distributed computing for big data is often achieved through inter-machine communication. The vast majority of distributed computing systems, such as Spark, use the TCP (or UDP) and IP protocols to achieve inter-machine communication. However, this results in an additional layer of indirection, where data cannot be directly located as there is little correspondence between data and machine name. To showcase the benefits and practicality of networking over data names for a distributed computing system, we present Flint, a distributed computation engine modeled after Spark and utilizing the Named Data Networking architecture. By exploiting features such as multicast data delivery over names, caching, and data security, we demonstrate the feasibility of a data-centric paradigm and its potential performance advantages in the context of cluster computing over a dataset.

1 Introduction

Distributed applications process and ingest thousands of petabytes worth of data each second. Developers turn to distributed computation frameworks to process this data. MapReduce, created by Google and Jeffrey Dean in 2004, is one of the first such systems to try to process data at a large scale [2]. Inspired by this work, Doug Cutting and Mike Cafarella created Apache Hadoop, which uses a distributed file system to perform the MapReduce computation model with fault-tolerance and scalability built in. Since then, big data systems have always been designed as traditional applications atop the standard TCP/IP stack. These systems are now ubiquitous, especially as the primary enabler in the era of machine learning [3]. However, conventional TCP/IP networking must add a level of indirection between the location of data and a client request, or in other words, the name of the machine. Current solutions, such as DNS map names to machine addresses rather than the data itself, requiring additional overhead at the application level to use a service. An example of this is the NameNode in the Hadoop Distributed File System (HDFS) [8], which is used to report the locations of data on various DataNodes. Often, exploiting data locality, load balancing, and caching in traditional big data systems require the engineering of explicit mechanisms and subsystems, increasing overhead and complexity of the overall framework.

We propose Flint, a novel distributed data processing engine built using the NDN (Named Domain Networking) architecture to potentially decrease or eliminate this and other possible overhead by moving request-response semantics common in big data systems to the network layer. Flint utilizes a familiar programming interface similar to big data systems like Spark [11], providing ease-of-use regardless of the underlying paradigm.

Flint is available as open-source software. The source code and documentation are available at https://github.com/UCLA-IRL/flint.

2 Background

2.1 Spark

Flint takes heavy inspiration from Apache Spark [11], in particular the concept of lineage. Spark stores DataFrames as resilient distributed datasets (RDDs) which are lazily evaluated, and potentially cached when evaluated through a series of transformations. In Spark, lineage is defined as a directed acyclic graph (DAG) representing the relationship between RDDs originating from some base datasets, where nodes represent RDDs and edges are transformations.

These RDDs are read-only (adhering to the functional programming model), distributed and partitioned across a cluster, and lazily transformed. For example, if an RDD is lost, then using the lineage graph, Spark can recompute the lost RDD.

The programming interface is also simple. The developer writes a driver program which launches operations in parallel. The results of computation are eventually collected at the driver. Spark provides the developer with flexibility in controlling when RDDs are computed and cached, which can

also be handled automatically due to the lazy evaluation of RDDs.

2.2 Named Data Networking

The Named Data Networking (NDN) [1] network architecture provides functionality to be used in place of the security, transport, and network delivery layers of the traditional TCP/IP architecture. In the place of machine names are names for individual, immutable data packets. These packets are requested directly by name in an Interest packet, sent from a consumer to any forwarder through the underlying medium. As needed, interests are routed by the network towards producers, who reply to an Interest with signed Data. Following this pattern, one can observe that request-response semantics (such as those provided by protocols such as an SQL connection or HTTP) are moved into the lower levels of the network.

NDN names are hierarchical and semantically meaningful, where each name consists of several name components with a type and value. This philosophy allows for routing over name *prefixes*, where over all prefixes in the forwarding information base, the longest prefix match of a descendant name is used as a next hop. This also allows NDN data names to align closely to application semantics–the paradigm for data organization and querying often itself follows a hierarchy. Additionally, semantically-meaningful names allow for obtaining data without any indirection through name lookup–consumers can be implemented in a fashion where application semantics can be directly translated into a meaningful name to put in an Interest.

NDN routing [10] follows a stateful, closed-loop design, integrating caching and multicast data delivery, thus improving network performance. When a forwarder receives an Interest, it first checks whether the Interest can be fulfilled by Data already present in its content store (cache), resulting in an immediate reply with data. Otherwise, the forwarder puts the interest in its pending interest table (PIT). While maintaining the PIT requires maintaining state, it allows for greater efficiency and a more robust control plane. For example, the forwarder will avoid sending duplicate interests for the same data through the same interface if it already exists in PIT. After the interest is forwarded to the next hop recursively and data is returned, the data is sent back through all interfaces with an active PIT entry, allowing the same data response to serve multiple consumers. A popular package for NDN forwarding is NFD (the Named Data Networking Forwarding Daemon) [4], which is utilized in the Flint cluster.

Security [12] is a fundamental design principle in Named Data Networking. All data is required to be signed, including keys stored as named data. Thus, named data which holds a key serves the purpose of a certificate. Similarly to other PKI (public key infrastructure) designs, verification of NDN data is a recursive process where the signature of the original data followed by the signature of the certificate that signed it is obtained and verified, and so on until a trust anchor key known *a priori* is reached. By securing the data and not the communication channels (as done traditionally in HTTPS and TLS), the provenance of the data need not be the location it was obtained, enabling data to be more efficiently cached and distributed.

3 Design

Flint is designed as a distributed computation engine that performs computation and analysis over sharded datasets using a cluster of nodes. These datasets are similar to a traditional tabular dataset. More concretely, Flint processes *transformations* over a dataset, turning it into a new dataset to which further transformations can be recursively applied, forming a lineage. Such a computation over data can effectively express various big data tasks, similarly to Spark. Each node does not necessarily have the complete data due to its size; the principle of locality guides the design such that each node should perform requested computations over some subset of the data it has. Each dataset is divided into shards, where each shard's size is on the order of dozens of megabytes. A key principle for such a distributed computing cluster is that any available worker should respond to a request for computation provided they have the ability to complete the request (e.g., sufficient hardware resources or the correct shard of data).

3.1 Programming Model

A developer with access to a Flint cluster's client can write a program that represents a parallel data task performed across the client.

The basis of this model is a Dataset object, which represents a certain view of data stored across the cluster and its lineage through various transformations. Though the developer may hold a Dataset object, the object may not yet have been materialized or computed across the cluster–the design adheres to the principle of lazy evaluation.

A Dataset object representing a descendant point in the lineage can be obtained by calling a *transformation* on a Dataset object. A transformation call is supplied a callable (closure) which specifies how the dataset should be changed to reach this new point in the lineage. For ease of use, methods created using syntactic sugar applied to the base transformation method provide a more familiar dataset interface to the developer, such as the ability to map and filter the dataset.

The developer can also choose to *cache* a Dataset, which hints to the cluster that the dataset, at its current lineage, should be materialized and stored (on the Workers) now. While Flint can automatically determine good cache points, such as intermediate Datasets on which different transformations are subsequently performed, we allow developer the granular control over adding more such cache points.

```
log = client.create_dataset("app.log")
errors = log.filter(lambda x: x.contains("
    ERROR"))
errors.cache()
severity = errors.map(lambda x: 2 if x.
    contains("CRITICAL") else 1)
auth_errors = errors.filter(lambda x: x.
    contains("403"))
print(auth_errors.collect())
```

Figure 1: An example of the Flint programming model, demonstrating the use of transformations and caching to filter a log dataset.

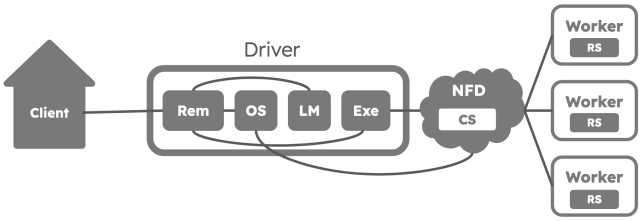

Figure 2: Diagram showing the bi-directional communication between different components on each node of the cluster, including the Client, Driver remote interface (Rem), Driver object store (OS), Driver lineage manager (LM), Driver executor (Exe), NFD content store (CS), and Worker result store (RS).

Finally, the developer can call a method to perform a *operation* on the Dataset, such as the collect operation. Operations are computations that have some effect visible to the user. For example, the collect operation returns the dataset as-is to the user, allowing them to view and manipulate the final result locally.

An example demonstrating transformations, caching, and collection over a Dataset object is listed in Figure 1.

3.2 System Components

Flint's cluster consists of one Driver node and at least one Worker node, connected through network infrastructure running an NDN forwarder supporting in-network caching through the content store. The client communicates to the cluster via RPC call to the Driver, who then coordinates and plans the lazy execution and materialization of a distributed dataset. The Driver talks through the network and reaches a Worker through interests for certain names (see Subsection 3.3). The Worker itself determines how to best execute the computation before doing so and saving the result. These system components are described in more detail below, and the communication between these components and sub-components are depicted in Figure 2.

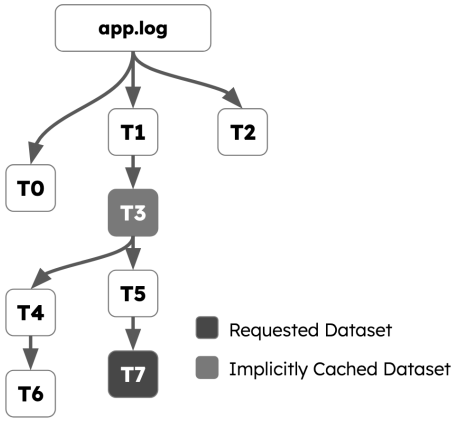

Figure 3: Example of a lineage manager execution plan. The lineage ending with transformation $T7$ is requested to be materialized. Thus, the lineage ending with transformation $T3$ should be materialized as well.

3.2.1 Driver

The Driver runs its modules described below on a single machine, using multiprocessing as necessary to accommodate for simultaneous communication (e.g., to simultaneously provide objects as well as return a result to the client).

Remote Interface: The remote interface serves as the entry point into the driver from the client. Several methods are exposed through RPC, roughly corresponding to the client programming interface described in Subsection 3.1. The primary pattern of the Remote Interface is to dispatch other modules on the driver via their interface to perform tasks.

Object Store: The object store is used to provide *objects* (blobs supplied by the client to the workers), organized by collection and object ID. An example of an object is the bytecode corresponding to a callable (closure) for a transformation; upon registration by the client, the bytecode is stored in the "Transformation" collection with a random object ID. The Driver announces the prefix for the object store to the forwarder, allowing for Workers to reach object store objects through Interests. As object store names are semantically meaningful (the prefix followed by the collection and object ID), Workers can directly retrieve from the object store without any additional lookups.

Lineage Manager: The lineage manager plans the execution of a series of transformations when a distributed dataset is requested to be materialized. As the tree of lineages rooted at a base dataset may have nodes with multiple children, it is more efficient to cache an intermediate result where the path from the root to the tip of the lineage contains such nodes. A depiction of the task of the lineage manager is shown in Figure 3. The lineage manager keeps track of every lazily-requested transformation, and advises the executor on which

lineage prefixes to materialize when a Dataset object is cached or collected.

Executor: The executor is responsible for the materialization of a Dataset, as well as any requested operations. For a requested lineage, it sends requests for the transformation at the shard level to workers using the Dataset protocol. Sharding is transparent to the user; the executor is responsible for sending requests for all available shards forming the complete dataset. If the execution results from an operation (such as collect) it is also responsible for combining and returning the final data from each shard to the remote interface and hence the client.

3.2.2 Worker

Each worker operates independently and can perform requests asynchronously, allowing it to take on simultaneous requests. Each module described below fires its tasks when needed in a central event loop.

Handler: The handler is the entry point for each Worker and contains request handlers for Interests related to the worker-side of the Dataset protocol. Handler invocations are short-lived and computational tasks are not done on the handler's "green thread"; the protocol is designed to not block subsequent Interests. It is also responsible for reporting which shards of the base datasets it owns, implicitly through the announcement of prefixes.

Compute: The compute module performs the necessary computations to materialize and store transformations done to a specific shard of the distributed dataset. The compute module is designed to be as efficient as possible, i.e., perform as few transformations in the dataset lineage while maintaining correctness. It does so by exploiting the caching properties of NDN as well as previous cache points in any worker's result store. Specifically, it tries every path prefix in the transformation lineage in order from longest to shortest (a path prefix represents an intermediate state of computation with a few operations missing at the end of the lineage from the desired distributed dataset). For every such prefix, it first looks for the resulting data in its own result store. Then, it checks the network for the corresponding result name of the path prefix, according to the dataset protocol. Logically, the latter step is equivalent to first checking in the network cache (content store), and then checking the result store of other Workers. From any intermediate dataset the Worker finds, it performs the necessary subsequent operations to it before saving the final result in its result store.

Result Store: The result store saves computed views (results) of a dataset shard corresponding to transformation lineages. The data is stored in memory for fast retrieval, with an LRU policy to evict old views, as any evicted view can be re-requested and computed. Additionally, the result store further sub-divides the data into segments which fit into the network MTU (maximum packet size supported by links) in which the cluster resides, as individual shards are still orders of magnitude larger than network MTU. When the Worker's handler requests a specific result and segment number, the result store fetches the appropriate segment such that the handler can sign and reply to the Interest with the result data. Workers do not announce a general prefix for the result store, instead, results are announced on the level of an individual result. This ensures the network will only route requests for results to the Worker which has them.

3.3 Dataset Protocol

The design of any application running on a data-centric network architecture leans heavily on naming conventions, both for hierarchical semantics and security. In Flint, we exploit the fundamental concept of lineage and its correspondence to hierarchy to design the protocol by which distributed datasets can be requested. To illustrate the protocol, Table 1 overviews an exemplar request and result name.

In NDN, names are a series TLV (type-length-value) blocks, each representing a name component. Most name components are generic-typed, meaning they are an arbitrary sequence of bytes. We use two other types for delimiting purposes as well as for their semantic meaning, namely Keyword components (`32=`) and Segment components (`50=`). The Keyword components are used to denote different spans in the hierarchy (e.g., file path hierarchy versus lineage hierarchy). The Segment component is used to denote the segment number of the result.

Generally, both the request and the result name consist of the application prefix followed by the type of interest. A *request* interest tells the Worker to begin the computation of a specific distributed dataset by performing the transformations, while a *result* interest requests a piece of the actual data computed at the behest of a request. Following the type is the path of the file relative to the root of the distributed file system, representing the base distributed dataset. Subsequently, a number indicates the shard on which to perform the computation. Two Keyword components follow, indicating the the start of the lineage hierarchy and the list of transformations in the lineage respectively. The transformations, in hierarchy order, are included in the name, represented by the object ID (UUID) which the Driver's object store assigned to the transformation bytecode. An ending Keyword component serves as an end-cap, such that NDN's in-network name discovery mechanism will not return a cached dataset with more transformations applied to it due to the hierarchical scheme. (Except for the end-cap, distributed dataset requests with a prefix of another dataset's transformations applied to it are also NDN name prefixes to that dataset).

In the case of the result name, a Segment component denotes the specific 0-indexed segment number to retrieve, as data from a distributed Dataset cannot fit within most network MTUs.

Type	Name
Request	/ndn-compute/request/app.log/5/32=LINEAGE/32=TRANSFORMATIONS/uuid-1/uuid-2/32=END
Response	/ndn-compute/response/app.log/5/32=LINEAGE/32=TRANSFORMATIONS/uuid-1/uuid-2/32=END/50=7

Table 1: An example of a request and result name for two consecutive transformations (with object IDs uuid-1 and uuid-2 respectively) on the 5th shard of a log file. The result data name is requesting the 7th segment.

3.3.1 Driver Behavior

When a client requests to cache or collect a dataset, the Driver will make a requests to the Worker by expressing n Interests for request names to the connected NDN network, where n is the number of shards. The request names will include the file name, shard number, and object ID's corresponding to the transformation.

If the Worker requested to collect the dataset, the Driver will subsequently express Interests for the corresponding result names for the n shards, retransmitting the Interest as necessary into the network until actual data (and not a negative acknowledgment) is returned. This change in state indicates that the Worker is finished computing the result. Then, the Driver will iterate over all $s(n)$ segments for each shard, sending an Interest for each and concatenating received data in-order until all data is collected.

3.3.2 Worker Behavior

Upon receipt of a request Interest, a Worker in the cluster will parse the name and begin a compute task in the background. The Worker will immediately return an empty Data to acknowledge the start of the computation process.

Upon receipt of the result Interest, a Worker in the cluster will look for the result in its result store. By the design of the worker's result store, all Interests received should be for a result which the Worker has. Before the Worker announces the specific prefix for a result, any Interests for that result name are returned as a negative acknowledgment by the network; such Interests do not make their way to any Worker. The Worker will parse the specific segment requested, and return an MTU-compatible chunk of the result as the data for the Interest.

Workers will express Interests for result names during computation of a distributed dataset. This is to abide by the principle of efficiency; the Worker will perform as few transformations in the lineage as possible should a path prefix of the transformation lineage exist in the network or in another Worker's result store. However, Workers will not express Interests for request names; workers assigned to compute transformations on a certain shard should do so by themselves.

4 Implementation

Flint is implemented as a Python 3 package spanning approximately 2,500 lines of code. The package is organized into modules; certain modules run on nodes assigned to Driver or Worker roles while other modules are used as utilities or for cluster management.

While Flint is designed to be run on any containerization technology or even bare-metal hardware, it is tested on a Docker cluster managed through either Docker Compose or a management module utilizing the Python Docker SDK (the latter allows for more flexibility in configuration and management). Currently, the cluster design allows for a Driver to work with up to 200 Workers. The python-ndn library [7] is used to provide network functionality to the Driver and Worker as a native NDN SDK.

Nodes in the cluster are designed to connect to any NDN network (given the authorization as well as the proper certificates); however, we bundle a local NFD (Named Data Networking Forwarding Daemon) [4] instance, configured with a 5-gigabyte content store (cache) to allow for sufficient in-network caching capability, emulating real networking hardware.

The Driver and Workers automatically register themselves to the cluster through the NDN architecture's hierarchical named-based routing. For example, upon startup, a Worker will announce name prefixes to the network for each shard that it determines itself to have. Thus, there is no need for an explicit runtime cluster manager. Node health and reachability are synonymous with the equivalent functions in the routing plane.

To increase performance and compatibility, the current dataset processing design for Flint uses a Pandas [9] backend. Transformation callables (closures) are expressed with Pandas library calls. Base distributed datasets are stored as flat files in `jsonl` format. To distribute the dataset, a bundled utility shards arbitrarily large files into shards of a certain size and replication factor to distribute to each Worker.

5 Evaluation

To demonstrate the feasibility of a data-centric paradigm for distributed computing, we evaluate Flint with microbenchmarks to test the performance of features enabled or enhanced through the NDN architecture. Unless otherwise stated, the evaluations are performed on a single machine running macOS Sequoia 15.3.2 with an ARM 10-core processor, 32 gigabytes of RAM, and solid-state disk, with all nodes running within their own Docker containers.

In-network Caching: The microbenchmark tests transfer-

ring an already-computed 64 MiB distributed dataset from a result store to the client using two Workers. During the first pass, the content store of network routers are empty. Then, for the second pass, all segments of the dataset are pulled into the content store before the test begins. In both passes, the end-to-end time from client request to client return is measured over five trials.

Pass	Mean (s)	Stdev (s)
Pass 1: content store empty	35.17	2.19
Pass 2: content store has data	15.62	1.11

Thus, when data is stored in the network content store, the end-to-end latency of retrieving segments decreases by roughly 50%.

Name Hierarchy and Lineage: This microbenchmark tests the use of semantically meaningful names to retrieve intermediate Datasets. When a Worker receives a request for a Dataset through the naming convention, it can directly determine the necessary lineage of transformations, as well as the names of intermediate datasets (which may be cached). Thus, by using the network cache as well as the result store of other Workers, the Worker may be able to decrease request latency.

To evaluate this latency decrease, we request a CPU-intensive series of transformations on a 500,000-row distributed dataset (approximately 200 MiB), measured over five trials. The dataset is tested on a cluster of two Workers. When taken together, the lineages form a computation tree with two branches and thus a split point. We test the end-to-end latency for the result to be collected and returned to the client under two different policies: with automatic caching, which computes the dataset at the split point before the two ends of the lineage branches, or without such caching.

Configuration	Mean (s)	Stdev (s)
With auto caching	81.67	0.55
W/o auto caching	93.07	0.07

From the data, we observe a latency decrease with automatic caching, indicating the advantage of exploiting hierarchy both in the semantics of distributed dataset processing as well as content naming.

Data Locality and Availability: Requests for computation are expressed as Interests at the shard-level and are routed to a Worker who has the shard. Additional factors are used to rank different interfaces that can produce data under the prefix; namely, routing cost and congestion. This prevents any Worker from being inundated with requests, thus bringing load-balancing into the network layer.

We utilize this microbenchmark to evaluate how increasing the replication factor and thus the proportion of data owned by each Worker affects the performance of Flint. Using a cluster with 16 Worker nodes, we run two distinct jobs over a 200 MiB dataset. Job 1 is a data-filtering job, while Job 2 is a CPU-intensive job. We then measure the time for the request to be completed and for the result to be returned to the user. During each trial, we vary the average proportion of data each Worker node has by changing the replication factor.

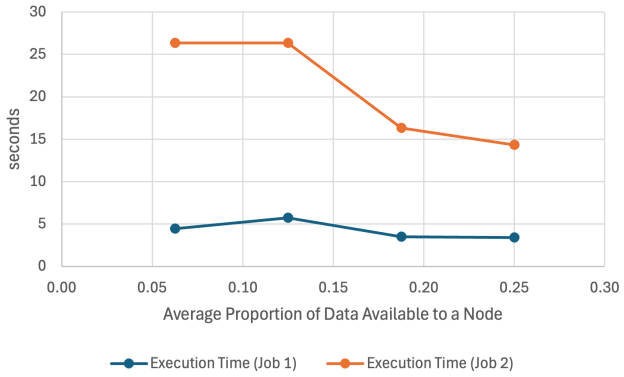

Figure 4: Data availability per node versus execution time.

The results are shown in Figure 4. Especially for the higher-intensity Job 2, we see a general negative correlation between replication factor and latency. We notice that further enabling data locality by placing the data on more nodes may have a positive effect on performance.

Interest Volume and Network Performance: Chunk size is often an important tuning knob for distributed datasets and file systems. Similarly, shard size in Flint can affect the performance of the system. Smaller shard sizes increase the number of request Interests that are sent, as the names are at a per-shard granularity. Note that the number of result Interests sent is independent of the chunk size, as result Interests are split into segments orders of magnitude smaller than chunk size. However, in Flint, chunk size affects the size of the unit of work, and thus the difficulty of placement. Utilizing NDN routing, we compare how the volume of request Interests (inversely proportional to chunk size) affects performance using a cluster of 16 Workers performing a computation over a 500,000-row distributed dataset. During each trial, we vary the chunk size by 10-megabyte increments and measure the end-to-end latency.

The results are shown in Figure 5. Generally, tasks for smaller chunk sizes are easier to place. The architecture of Flint, while requiring more Interests for smaller chunk sizes, takes advantage of the correlated ease of placement, providing higher performance.

6 Discussion

6.1 Performance

Flint's implementation aims to be robust and relatively performant; however, we do not include an explicit quantitative com-

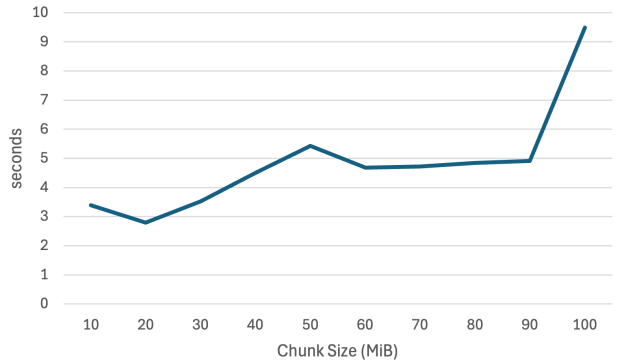

Figure 5: Chunk size versus execution time.

parison with Spark and other distributed processing frameworks. These frameworks are often written in lower-level languages with bespoke multiprocessing features, resulting in increased performance and complexity. In the interest of providing a beginning a discussion about the benefits of a data-centric architectural paradigm for big data, we provide a comparison of performance with Flint itself as a baseline in Section 5. This baseline allows for a more accurate comparison of performance benefits.

Most saliently, we observe that utilizing in-network caching significantly improves the end-to-end data transfer performance of Flint. In a larger-scale deployment of a data-centric distributed processing framework, where data and compute may be spread in datacenters worldwide or on the edge, such caching may provide a critical performance benefit without the need for additional edge infrastructure or a content delivery network (CDN).

6.2 Comparison to Spark

Flint draws heavily in its design from Spark [11], especially in its programming model. However, we summarize a few notable improvements. First, in the common usage of Spark with the Hadoop Distributed File System (HDFS) [8], scheduling a distributed computation task and achieving data locality is achieved through a call to `getPreferredLocations()`, which incurs the additional round-trip-time of requesting the best DataNode for a certain partition through the NameNode. However, in Flint, this additional round-trip-time is eliminated through the usage of a data-centric network to provide data locality. The Driver can directly determine a semantic name for the requested shard (partition) of the dataset and request it from the network.

Additionally, Spark makes heavy use of caching. A limitation is that cache use is limited to each local node only; cache is not shared between nodes or in the network. Flint, however, can utilize both in-network caching as well as the storage (result store) of other Workers. During computation of a distributed dataset, Flint checks every prefix of the lineage to see if a prior result is cached not only on the local node, but also in the entire cluster.

Similarly to another data-centric application, Kua [5], Flint's underlying data store is resilient to failure.[1] While a popular solution for Spark, HDFS, relies on a central NameNode to be available for the functionality of the filesystem. However, in the case of Kua and Flint, no central node controls the underlying storage. Even if some forwarders in the network go down, given a sufficient replication factor and connected Workers, Flint can continue operating, as data is always located directly through the network.

6.3 Security

Security [12] is a fundamental tenet in NDN. In addition to data security through signatures, we also explore routing security through the design of Flint.

We explore an example where malicious bytecode may be spread to Workers through improper retrieval or verification of transformations. While the authorized Drivers should be the only ones able to produce valid object store data, buggy or lazy Workers may not verify this. To prevent any Worker from receiving a malicious transformation in the first place, we introduce routing security through signed Prefix Announcements, a new protocol that increases granularity in the security of forwarding commands sent to an NDN forwarder. Specifically, only Drivers who are able to sign a Prefix Announcement data [6] with a key for Drivers can announce the prefix for object store data. Routing security is enforced for all entities and prefixes in the Flint cluster.

7 Conclusion

Our work explores the viability of creating a Spark-like distributed computation engine over Named Data Networking. We present a novel cluster architecture that utilizes NDN network features to perform optimizations not possible under a TCP/IP network architecture. NDN's hierarchical namespace allows workers to announce prefixes corresponding to the data that they have local access to. Drivers can automatically issue execution requests to workers without needing to know where they are in another distinct namespace (e.g., IP). The strength of Flint lies in the NDN architecture, where data names and Worker names are one.

In-network caching allows many other optimizations. Transformations at previous (and current) lineage cache points are stored in-network. Other Workers may see these save points by expressing an Interest. This affords Workers greater efficiency by relying on the efforts of other Workers. Transformation bytecode is generated by the client; the driver regis-

[1] Flint's data store philosophy shares similarities with Kua's. Kua is a distributed object store; we further add a distributed computation system to the base store.

ters its object store prefix to make it available to the workers. Since transformation bytecode is generally small, in-network caching greatly reduces the burden of the driver when serving bytecode. Flint builds on top of the network caching mechanism through the object and result stores of the Driver and Workers respectively.

The built-in forwarding strategy of NDN forwarders automatically implement congestion control–forwarding to different paths based on data result metrics. Since NDN forwarding is closed-loop and stateful, it can automatically detect congestion (i.e., worker overload) and decide to forward computation requests to a different worker.

7.1 Future Work

Currently, some limitations exist with our work. Like Spark, the underlying datastore (in this case, Datasets) are read-only. Allowing clients to modify datasets would require a significant change to our design. This would enable clients to have finer-grained control over computation, but at the potential cost of complexity.

Another potential area of improvement is Flint's performance. Since this is a proof-of-concept, we did not perform any benchmarks against existing distributed computation engines. We theorize that we can improve performance by switching to a more performant language (such as Go) and taking better advantage of multiprocessing. Additionally, we hope to adjust our design to be less reliant on retransmitting interests. Currently, we send out interests with some backoff to fetch results, waiting on the Worker's computation to complete.

We enumerate additional potential improvements to Flint below:

- Allowing workers to provide negative acknowledgment to execution requests for finer-grained balancing control in addition to built-in congestion control.
- An implementation of shared variables (broadcast and accumulators).
- An agnostic computation model rather than one only supporting Dataset manipulation through Pandas DataFrames.
- Support for other data-centric storage such as Kua [5] in addition to the current proof-of-concept filesystem.
- Implementation of more operation semantics, such as Spark's `foreach` and `reduce`.

8 Author Contributions

As visible in the commit history of our GitHub repository, Jacob set up the Docker cluster, NDN wiring, lineage manager, and programming interface for the client. Paul handled the cache trigger and distribution of work to the cluster workers through NDN interests and handlers. Omar did the logic for executing transformations on the workers and collecting the results as a DataFrame back at the client.

Omar created a majority of the presentation slides and contributed to the Introduction and Conclusion sections of the paper. Paul wrote the Discussion and Design sections, while Jacob wrote the Background, Implementation, and Evaluation sections.

References

[1] Alex Afanasyev, Jeff Burke, Tamer Refaei, Lan Wang, Beichuan Zhang, and Lixia Zhang. A brief introduction to named data networking. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE, 2018.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[3] Mu Li, David G Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. *Advances in neural information processing systems*, 27, 2014.

[4] Named Data Networking Project Team. NFD Developer's Guide, July 2021.

[5] Varun Patil, Hemil Desai, and Lixia Zhang. Kua: a distributed object store over named data networking. In *Proceedings of the 9th ACM Conference on Information-Centric Networking*, pages 56–66, 2022.

[6] Davide Pesavento and Junxiao Shi. Prefix announcement protocol. Technical report, NDN Issue Tracking (Redmine), 2025.

[7] python-ndn authors. python-ndn documentation, 2023.

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[9] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[10] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. A case for stateful forwarding plane. *Computer Communications*, 36(7):779–791, 2013.

[11] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*, 2010.

[12] Zhiyi Zhang, Yingdi Yu, Haitao Zhang, Eric Newberry, Spyridon Mastorakis, Yanbiao Li, Alexander Afanasyev, and Lixia Zhang. An overview of security support in named data networking. *IEEE Communications Magazine*, 56(11):62–68, 2018.