Decentralizing Big Data Processing on Spark using IPFS

Jaspreet Arora Nandan Parikh Srishti Majumdar Swati Sharma

Abstract— **The information age has made vast amounts of data available to governments and corporations. Most of this data has been privately collected by these entities for personal analysis and processing. Therefore, the architecture of most big data applications rely on local clusters for the storage and management of data. The most widely used solutions for data management on these clusters are platforms like HDFS(Hadoop File System), HBase, Hive, S3, which rely on privately sourced resources for data management and storage. Big Data processing systems like Spark are tightly coupled[5] and highly reliant on these storage platforms. With the advent of IoT systems, as the internet moves to a more decentralized architecture[6, 15, 10, 13, 9], the need for data management will move from private centralized servers to more decentralized architectures. In addition, as more entities seek to open source their data or share data with other entities, platforms like HDFS come up inadequate for such use cases. Sharing data through centralized storage environments inevitably involves replication of data by the user and thus, leads to waste of resources for storage and management of this replicated data. Decentralized cloud storage is an efficient and economical solution for large-scale storage and data sharing. A number of decentralized data distribution platforms have been developed for this use case such as Storj[12] and IPFS(Inter Planetary File System)[8], but they remain untested for use cases involving storage and processing of Big Data[11]. With this work, we aim to enable such use cases by provisioning the use of the most highly used big data processing tool Spark with the distributed file storage platforms IPFS.**

I. INTRODUCTION

The need to perform analytics and research on the increasing volume of data by corporations and government has led to the development of Big Data processing systems such as Google MapReduce[1], Apache Hadoop, Apache Spark[5], Dryad[] among others. Distributed storage and file systems form the backbone of such systems to provision the access to this mass store of data. The most popular architecture of Big Data applications consists of the following parts.

1) A processing engine such as Hadoop or Spark which processes the data in parallel across various nodes in a cluster.
2) A distributed filesystem such as HDFS[4] which stores the data across the different nodes in the cluster.
3) A resource management or scheduling system such as YARN[7] that schedules jobs on the cluster nodes.

Most of the data being available for processing is centralized and stored in private data centers. The design and architecture of HDFS and other big data processing frameworks support this paradigm and are designed for fully connected and controlled clusters designed to run on a controlled and privately managed cluster of servers. The tight coupling of Big Data processing systems and HDFS present a number of challenges as discussed below.

1) Centralization – HDFS clusters are centralized clusters owned and controlled by a single entity. If a data set is shared, there are concerns regarding the longevity of the data set. This places a lot of power and responsibility in the hands of large entities who can afford to store these large datasets. If the HDFS cluster goes down, the data might be lost forever.
2) Moving computation to data – In most Big Data applications, the processing system and the storage are deployed on the same cluster nodes. This ensures efficiency for the entity that owns the data, but any third party user of the data set would need to maintain a local copy of the data on their cluster. The centralized nature of HDFS presents a challenge for sharing data among entities for processing and analytics.
3) Performance concerns – Private data centers are susceptible to privacy breaches, unavailability concerns, high storage costs and slow upgradation problems.

During recent times, peer to peer systems have emerged as a viable alternative for the implementation of distributed file systems. In a peer to peer system, data is stored on shared resources in the network and can be directly accessed by the members of the network instead of through a centralized server. Peer to peer servers are potentially more scalable as compared to centralized and client-server solutions. Apache Cassandra[3] and Amazon DynamoDB[2] are examples of popular peer to peer storage solutions available for Big Data applications.

Although these systems offer a self sustained, scalable and fault tolerant network of machines, they are not truly decentralized as they are designed for privately owned and managed cluster of machines as key-value stores. They cannot support a full fledged distributed file system. As the internet evolves into a more decentralized[6, 15, 10, 13, 9] architecture, the community has developed various decentralized storage solutions to provide performant and secure systems that can share and use the resources already available in the internet. One such system is the Interplanetary File System(IPFS). Theoretically, these systems can scale well for storing large amounts of data[12], but their viability or performance have not been tested with any Big Data systems[11].

In this work, we implement an interface between Apache Spark and IPFS to enable Spark to utilize IPFS directly as a data store for retrieving and storing data. We also test the implementation with a simple use case of word counting to prove that such an integration is practical and well realized.

Section 2 describes similar work to our own in extending big data systems to other forms of storage. Section 3 provides the background to IPFS and other decentralized peer to peer file systems. Section 4 describes the high level architecture of our system. Section 5 delves into the technical details of our implementation. The results of our experiments are presented in Section 6.

II. RELATED WORK

There have been multiple attempts at proposing an alternative filesystem to HDFS. Most of them have been driven by an approach different from the design decisions discussed above. One of the most frequent consideration has been to remove the NameNode bottleneck. Some solutions have tried to alleviate the need for huge storage arising from maintain multiple complete copies of the content across the cluster. Finally, a few of them address the POSIX non-compliance. The motivation for this project comes from a different a place, it aims to eliminate the need to move the data from its data storage system to HDFS for performing analytics in a cluster.

A. Spark on Lustre

Spark allows working with other file systems other than HDFS. It supports in-memory iterative computations and there is less urgency to have the data local for long computations. Spark can use a POSIX interface to the filesystem that stores the content. There has been an effort to build an interface between Spark and the Lustre File System. Lustre, like HDFS, is a distributed scalable file system deployed to a cluster of commodity servers. The benefit that it provides over HDFS is that it is more scalable for high-performance computing applications. This helps in eliminating the need for two different file systems and avoiding the overhead of data transfer between them.

B. Spark with Cassandra

Cassandra is a decentralized key value store. They provide a connector library for reading and writing data to Cassandra clusters[1]. They use the DataSource API in Spark which makes Cassandra tables as first class citizens in Spark. Cassandra tables are directly parsed and manipulated inside Spark as RDD objects. They use the spark context parallelize method to parallelize the fetching of data from a Cassandra store.

III. BACKGROUND

The increasing amount of data being generated in the internet age motivates the consideration of peer-to-peer based cross-organizational analytics. Traditional platforms like Spark and Hadoop are designed to function in a cluster of servers in an organization on a dataset contained within the analytics cluster. Peer-to-peer systems, on the other hand, provide the flexibility to store data on personal devices and perform the computation without the need to transfer the complete data to the analytics cluster. This project aims to implement an interface between Spark and a decentralized data storage system for peer-to-peer analytics. This will allow the execution of MapReduce tasks without the data leaving the source peer-to-peer data storage system.

[1]https://github.com/datastax/spark-cassandra-connector

This section discusses the various options available to replace HDFS with a decentralized peer-to-peer file systems as the data source. All the peer-to-peer file systems overcome the bottleneck of data ingestion from a single file system, the need to move the computation closer to the data and failures due to the centralization of the data. The differentiating parameters were the data-sharing mechanism among the peers, the data replication scheme and an API to connect the computation systems to the file system, large file support, data versioning capabilities and approach to access control which were the basis of choosing IPFS over other existing peer-to-peer file systems.

A. Spark on HDFS

Analytics on a centralized data storage architecture using Spark employs Hadoop Distributed File System. HDFS is a distributed and scalable file system designed to perform analytics in large clusters of cheap commodity servers. It connects the local storage of the compute nodes in a cluster into a single distributed file system. Some of the key features and design decisions for HDFS which act as a drawback for the current use-case are discussed.

First, the file metadata and actual content are stored separately. NameNode is used to store the metadata while the DataNodes are used for content storage. Though this model simplifies the system greatly, this can act as a bottleneck in case of NameNode failure since the system contacts the NameNode to determine the location of the actual content.

Second, HDFS achieves replication by copying the data-blocks several times, usually 3, across a number of DataNodes. This plan ensures fault-tolerance and facilitates data distribution which increases the probability of a copy of the data being closer to the computation server. But the drawback is that storing multiple copies of the complete dataset is expensive in terms of storage requirements.

Finally, HDFS is not POSIX compliant giving it performance leverage but limits its compatibility with POSIX reliant applications.

B. Interplanetary File System

The Interplanetary File System (IPFS) addresses the deficiencies of centralized data storage by using a distributed peer-to-peer file-sharing system. It is an open-source project by Protocol Labs that aims to connect all peers with the same set of files. One of the defining design decisions is content-based file addressing makes it ideal for implementing data versioning.

1) Distributed Hash Tables: Distributed hash tables (DHT) spreads the data across a network of computers that coordinate among themselves to enable efficient access and lookup between nodes. The advantages of DHT include decentralization, fault tolerance, and scalability. In the absence of a centralized NameNode, the system ensures reliability even on node failure or if nodes are withdrawn from the network. Another benefit is that DHTs are scalable and can accommodate millions of nodes.

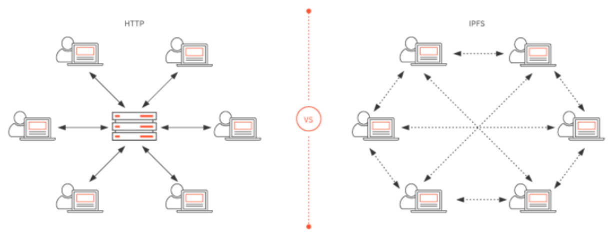

Fig. 1. HTTP vs IPFS architecture

2) Block Exchanges: IPFS's design is influenced by the file-sharing system BitTorrent that is capable of coordinating the data transfer between millions of nodes by relying on an innovative data exchange protocol limited to the torrent ecosystem. IPFS implements a generalized version of this protocol called BitSwap which operates as a marketplace for any type of data and incentivizes data replication.

3) Merkle DAG: IPFS models all its data on a generalized Merkle DAG making the data secure. A Merkle DAG is a blend of a Merkle Tree and a Directed Acyclic Graph (DAG). Merkle trees ensure that data blocks exchanged on peer-to-peer networks are correct and unaltered. This verification is performed by organizing data blocks using cryptographic hash functions that calculate a unique alphanumeric string value corresponding to the input. It is difficult to decipher the original input from the hash value. A Merkle tree summarizes all of the transactions in a block by producing a digital fingerprint of the entire set of transactions. This allows a user to verify if a transaction is included in a block or not. The individual blocks of data are called leaf nodes which are then hashed to form non-leaf nodes. These non-leaf nodes can then be combined and hashed repeatedly until all the data blocks can be represented by a single root hash.

A DAG represents sequences of information that have no cycles. A Merkle DAG is a data structure where hashes are used to reference data blocks and objects in the DAG. The advantage of this approach is that all content on IPFS can be uniquely identified since each data block has a unique hash. It also makes the data tamper-resistant as changing the content will alter the hash value as well.

4) Version Control Systems: Merkle DAG structure allows building a distributed version control system. A version control system allows users to independently duplicate and edit multiple versions of a file, store these versions and later merge edits with the original file. IPFS uses a similar model for data objects. If the objects corresponding to the original data and new versions are accessible, it is possible to retrieve the complete file-history. Also, IPFS does not depend on internet protocols for data distribution facilitating censorship-free web.

5) Inter-Planetary Name Space: Self-certifying File System (SFS) is a distributed file system that does not require special permissions for data exchange because data served to a client is authenticated by the file name signed by the server. IPFS uses this concept to create the Inter-Planetary Name Space (IPNS). It is an SFS using public-key cryptography to self-certify objects published by users of the network. All objects on IPFS can be uniquely identified. Similarly, each node on the network has a set of public keys, private keys and a node ID which is the hash of its public key and hence can be uniquely identified. Hence, the nodes can use their private keys to sign any data objects they publish, and the authenticity of this data can be verified using the sender's public key.

IV. ARCHITECTURE

A typical Spark job has the following structure:

1) A Spark job is created by specifying the source dataset in the job and deployed to the Spark master as a Java or Scala project
2) The Spark master runs as an independent process and coordinates the job using a SparkSession object to drive the execution
3) Tasks are assigned to workers by the resource or cluster manager
4) A task works on a subset of the dataset in its parition and outputs a new or modified dataset
5) Results are sent back to the driver application for saving

The design goals that guide our system architecture are as follows.

- Provisioning the use of IPFS as a data source for a Spark job – Spark jobs can use data from the IPFS network as source for processing
- Provisioning the use of IPFS as a data sink for a Spark job – Spark jobs can write the results from a job to the IPFS network
- Provisioning the system as a plug-and-play module for Spark

The overall architecture of the system has been described in Figure 1. The IPFS system is a peer-to-peer storage system with a communication channel between the IPFS nodes. The IPFS storage system exposes an HTTP endpoint which facilitates its connection with external systems and hence a way to access the data stored in the file system. Spark provides a File System API interface to fetch and use the data stored in a file system such as IPFS. This interface is implemented to contact the IPFS HTTP endpoint and stream the data for processing by Spark computation jobs.

A. Installation Options

The Spark File System interface implementation is compiled as a jar file and placed on each node in the Spark cluster. An IPFS daemon process runs on each cluster node to allow communication with the IPFS filesystem. There are multiple ways in which the system can be setup:

1) Single system: Both IPFS and Spark can be installed together on a single system. A single system acts as the data source and holds the data stored in the IPFS file system and at the same time houses Spark for performing computation on this data via Spark

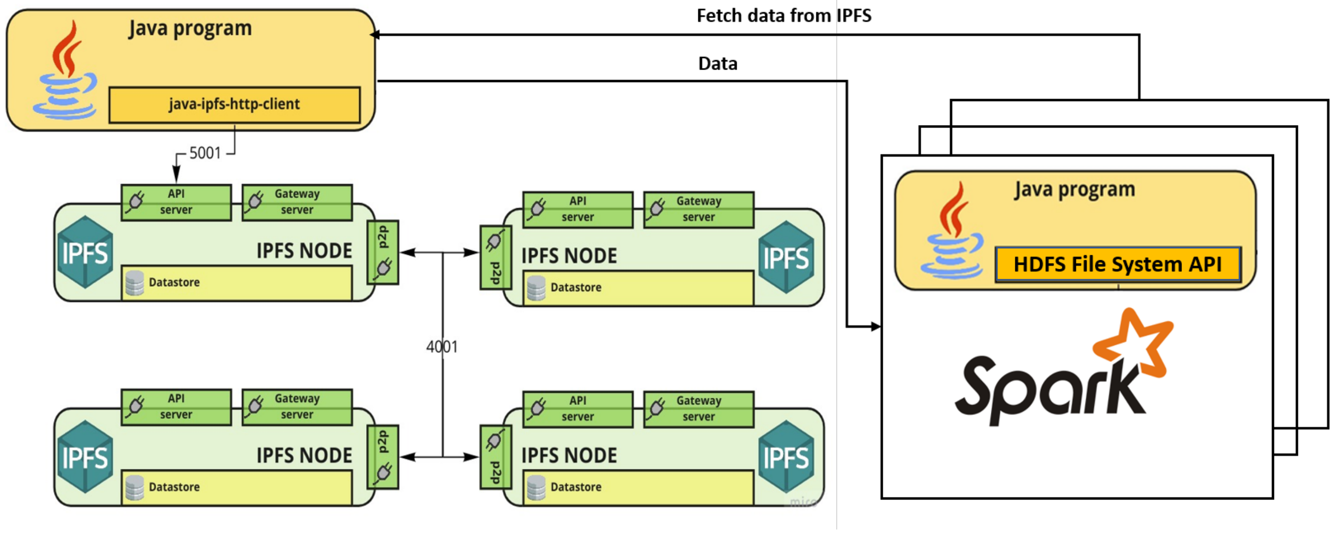

Fig. 2. Architecture Diagram

jobs. It will also hold the Spark file system API implementation for communication between IPFS and Spark.

2) Independent systems: Another installation option which provides much more flexibility is independent installation on different systems. This allows the data to be present anywhere in the world in an IPFS cluster. The cluster with Spark nodes used for MapReduce computations can be in a totally different location. This architecture eliminates the need to place both IPFS and Spark on the same server and the data can be fetched from the IPFS file system as and when required by the Spark jobs.
In this architecture, the IPFS file system can be placed on any private or publicly accessible network. The worker nodes which will run the Spark jobs and perform computations on this data will be placed in their own cluster. The file system API implementation will be placed on these cluster nodes. Since this implementation contains HTTP calls to the IPFS HTTP endpoint, the data will be streamed in by this API as and when requested by the Spark jobs.

B. Setup

1) Setting up IPFS: The IPFS protocol is written in Go which requires the *go-ipfs* package. A program called *ipfs-upadate* is used to install and update IPFS. Once *ipfs-upadate* is installed, *ipfs-update* install latest will install the latest *go-ipfs* version. The next step is to initialize an IPFS repository using *ipfs init*, the directory that holds the IPFS configuration and datastore. This command generates an IPFS repository with a standard configuration file. The datastore is the location the node hosts its share of network data. Initializing a repository generates a key pair and identity hash specific to the node. The *config* file is saved as *config* in the repository root and can be changed when needed. It contains information on the IPFS repository like peer list and data storage options.

2) Connecting IPFS nodes as peers: Firewalld will be configured to open the necessary ports for the network to interact with other peers. The nodes are connected as peers such that they have direct links between them. Since the nodes will be aware of each other, IPFS protocols will be able to optimize data transfer between them. To see a list of peers communicating with a node we use *ipfs swarm peers*. To connect the nodes together as peers *ipfs swarm connect <address>* is used where *address* is in the IPFS multiaddr format which requires the peer identity hash value.

3) Storing data on IPFS: To verify the connection between the IPFS file system and the Spark cluster, an experimental dataset is stored on IPFS. The dataset contains around 3000 text files with a total size of 1.2GB. The text files contain a corpus of multiple English language literature pieces obtained from Project Gutenburg.

4) Running Spark jobs: An experimental Spark MapReduce job is written for execution on the Spark cluster by streaming data from IPFS using the file system API implementation. The Spark job counts the word frequency on the IPFS dataset containing multiple files with English literature pieces.

C. Challenges

1) The latest available JDK version is 13. But, the systems like Spark, Hadoop and IPFS lag in adopting the latest versions. Hadoop has support only till Java 8 whereas some maven dependencies for Spark projects do not support Java 8+.
2) There are additional dependencies while dealing with IPFS clusters and using their open-source configurations to manage data via Java. As the first part of the project was connecting the systems and checking the feasibility of our generic wrapper, configuring the systems was a huge task. Going forward we feel that if these systems do not keep up to the pace of Java

releases, it can become a huge burden on the longevity of all these individual systems. On the part of our generic wrapper, the support for all such minor-major version combinations is not a feasible solution.

3) Although these systems and frameworks are used extensively throughout the industry, not much effort is spent by the developers to open-source system-level changes. Hence, customizing these systems becomes a huge challenge.
4) IPFS is in its initial stages and currently is more focused on developing a core user group. The focus on integration with different projects and existing systems is not well-maneuvered and hence it is very difficult to find generic examples for the same.
5) With the installation of IPFS, the users are not directly provided with clusters for replication as well as there is no direct support for IPNS. The content-based hash which is generated for each file has to be used to retrieve files which is non-intuitive.

V. IMPLEMENTATION

As the main aim of our system is the provisioning of IPFS as data source and sink, we explored the different options available for loading data into Spark through the IPFS cluster. The initial steps were the exploration of various methods of integration available in the Spark API for data provisioning, as well as the access methods available for the IPFS API, as discussed below.

A. Spark API

Spark provides a variety of options for declaring external data sources. Spark's primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets[16]. It natively supports storage systems such as the local file system, HDFS, Cassandra, HBase, Amazon S3, etc. for accessing and parsing files into DataFrames and RDDs[16]. Spark can directly parse text files from a URI using the read function available on a SparkSession. The method takes the URI for the file (either a local path on the machine, or a hdfs://, s3n://, etc URI) and reads it as a collection of lines[16]. Another method as used by the Spark-DynamoDB connector uses is the extension of the Spark Data Source API, which makes the DynamoDB a first-class citizen in the Spark ecosystem, such as CSV files. A custom data source can be implemented by defining two things –

1) A DefaultSource class which functions as the access point to the custom data source
2) A subclass of BaseRelation which describes the behaviour supported by the custom data source

As IPFS provide files for data processing, we found the first method more promising for our implementation. This can be achieved by implementing a central layer which implements the FileSystem API of the HDFS file system. The layer can then be deployed to Spark as a maven dependency or as jar file. Once this is deployed, we can declare the class as a valid data source[14]. This enables the support of IPFS URLs directly in Spark jobs, such that they can be referred as -

```
sparkSession.read().text("ipfs://$filehash$")
dataFrame.write().text("ipfs://$outputhash$")
```

The main functions to be implemented to extend the FilsSystem API are as follows:

- public FSDataOutputStream append(Path f, int bufferSize, Progressable progress);
- public void concat(Path trg, Path[] psrcs) throws IOException;
- public FSDataOutputStream create(Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress) throws IOException;
- public boolean delete(Path f, boolean recursive) throws IOException;
- public boolean exists(Path f);
- public int getDefaultPort();
- public void initialize(URI name, Configuration cfg) throws IOException;
- public FileStatus getFileStatus(Path f) throws IOException;
- public String getScheme();
- public URI getUri();
- public Path getWorkingDirectory();
- public boolean isDirectory(Path f) throws IOException;
- public boolean isFile(Path f) throws IOException;
- public FileStatus[] listStatus(Path f) throws IOException;
- public boolean mkdirs(Path f, FsPermission permission);
- public FSDataInputStream open(Path f, int bufferSize) throws IOException;
- public boolean rename(Path src, Path dst) throws IOException;
- public void setWorkingDirectory(Path new_dir);

The main methods implemented include the create function for creating files, getUri, and open, etc. The complete implementation of other methods is still in progress.

B. IPFS API

IPFS provides an HTTP based API for accessing files. The first step is the development of a Java layer on top of the IPFS API to encapsulate and provide all functionalities required for extending the HDFS FileSystem API. A Java class is created for this purpose. The Java class implemented the above mentioned methods of the HDFS FileSystem API and provide the main interface for communication with the ipfs network. Internally, the methods are implemented using the IPFS Java interface[2], which works on HTTPS. The IPFS Java API provides classes and methods for direction interaction with the IPFS HTTP methods. The main methods provided by this interface include all file system management methods

[2]https://github.com/ipfs/java-ipfs-http-client

including add, get, getStreamed, delete, etc. which are used to implement the required functionality for the FileSystem API. The files can be directly accessed by creating HTTP links such as https://ipfs.io/$filehash". This class is packaged as a jar file which is deployed at the Spark servers as a dependency and added to the classpath. In its current form, this file is added to the Spark job project as a dependency.

VI. Evaluation

The system is developed with an incremental approach with constant evaluation to help us understand the system APIs and the working of the systems as we went through each step. The job used for evaluation is the word frequency problem. The final goal is making the integration of Spark and IPFS seamless so that an IPFS file can be specified as a source inside a Spark job similar to an HDFS file through the Spark read API.

A. Local Access

In the first iteration, we created a Spark job to fetch a file from the IPFS network to the local Spark server for processing. The IPFS FileSystem API was deployed as part of the Spark job project. The files are first fetched by the job to the local file system of the Spark server using the IPFS FileSystem API classes. These files are then loaded into Spark using the local file system and are sent for compuation to the cluster. The final result is stored first in a local file system file, and then written to the IPFS cluster. This was a proof of concept to ensure for our IPFS FileSystem API. It proved that we can fetch data from the IPFS file network for processing by Spark and then write the output back to the IPFS file network.

B. Direct Access

In the next iteration, we created a Spark job to fetch a large file for the word frequency map reduce job. The files are streamed directly for processing to the Spark cluster using the IPFS FileSystem API methods, which internally use the Java 1.8+ streams. This enables Spark to chunk the file as it is being streamed to be processed in parts at the cluster nodes. The result is directly written back to a file in the IPFS network using the FileSystem API. This proves that we can directly connect the IPFS network files to a Spark cluster for processing. In the current form of the solution, the IPFS FileSystem API is deployed as a library dependency in the job project to the Spark cluster. The code inside the job explicitly uses 1the methods in the FileSystem API to perform the various read/write operations. The task of making Spark recognize the ipfs:// links as valid file sources and implicitly using the IPFS FileSystem API to perform the read/write operations to the IPFS cluster is still under development due to various dependency issues and a lack of proper documentation regarding the extension of the HDFS FileSystem API.

VII. Future Work

1) Implementing a layer for the conversion of IPFS content-based addressing to name-based addressing for Spark using Inter-Planetary Name System (IPNS) or DNS_text so that the data on IPFS can be accessible via ipfs://<address>. Inter-Planetary Name System (IPNS) is a system for creating and updating mutable links to IPFS content. Since objects in IPFS are content-addressed, their address changes every time their content does. A stable IPNS address will then point to the latest version of the data. A name in IPNS is the hash of a public key. It is associated with a record containing information about the hash it links to that is signed by the corresponding private key. New records can be signed and published at any time. DNSLink uses DNS TXT records to map a domain name (like ipfs.io) to an IPFS address. Because we can edit the DNS records, we can use them to always point to the latest version of an object in IPFS. Since DNSLink uses DNS records, we can assign names or paths or sub-domains that are easy to type, read, and remember. A DNSLink address looks like an IPNS address, but it uses a domain name in place of a hashed public key.
2) An important consideration for distributed systems is to place the computation closer to the input dataset. Currently, the system is not considering the geo-location of the IPFS node from which will be pulled when requested by the Spark job. Since many nodes in the IPFS file system may contain the same dataset, it is more efficient to contact the node closet to the DataNode performing computations on the dataset.

VIII. Conclusion

This project was motivated by eliminating the need to move dataset to HDFS for processing by big data systems by Spark and use decentralized storage instead. The approach followed was to implement an interface between Hadoop and the Inter-Planetary File System. This was aimed at eliminating the drawbacks of a centralized data store which includes inefficient data ingestion, single point of failure and complicated methods for data versioning. An ever-increasing volume of data fuels the need for decentralized analytics techniques as it will enable knowledge gain from datasets hosted on decentralized file systems not owned by a single organization that may not be suitable for the traditional analytics platforms. We were successfully able to prove that big data processing systems like Spark can be integrated with decentralized data sources such as IPFS.

References

[1] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[2] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[3] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[4] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[5] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[6] *The Mission to Decentralize the Internet*. 2013. URL: https://www.newyorker.com/tech/annals-of-technology/the-mission-to-decentralize-the-internet.

[7] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[8] Juan Benet. “IPFS - Content Addressed, Versioned, P2P File System”. In: *CoRR* abs/1407.3561 (2014). arXiv: 1407.3561. URL: http://arxiv.org/abs/1407.3561.

[9] *The future is a decentralized internet*. 2017. URL: https://techcrunch.com/2017/01/08/the-future-is-a-decentralized-internet/.

[10] *Advancing Toward a Decentralized Internet in the Real World*. 2018. URL: https://www.nasdaq.com/articles/advancing-toward-decentralized-internet-real-world-2018-06-01.

[11] *Application: Data Processing Workflows on IPFS*. 2018. URL: https://github.com/ipfs/ipfs/issues/248.

[12] *Storj: A Decentralized Cloud Storage Network Framework*. 2018. URL: https://github.com/storj/whitepaper.

[13] *The Decentralized Internet Is Here, With Some Glitches*. 2018. URL: https://www.wired.com/story/the-decentralized-internet-is-here-with-some-glitches/.

[14] *Apache Ignite Documentation*. URL: https://apacheignite-fs.readme.io/docs/file-system.

[15] *Decentralisation: the next big step for the world wide web*. URL: https://www.theguardian.com/technology/2018/sep/08/decentralisation-next-big-step-for-the-world-wide-web-dweb-data-internet-censorship-brewster-kahle.

[16] *Spark Reference*. URL: https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html.