DOI: 10.14529/jsfi250206

Dynamic Content-Oriented Indexing and Replication for High-Performance Storage and Analysis of Big Data in the IPFS Network

Maxim V. Shevarev[1], *Stanislav V. Suvorov*[2]

This paper presents an architecture for dynamic, content-oriented indexing and adaptive replication that enables high-performance storage and analysis of big data on IPFS. We first outline key gaps of vanilla IPFS for analytics – no global content search, non-guaranteed persistence without coordinated pinning, static replication, and highly variable retrieval latency – and address them with two components: (1) a two-tier distributed index (per-attribute/keyword inverted lists as IPFS objects plus a lightweight catalog that maps search keys to index CIDs via DHT/IPNS or CRDT-based dissemination); and (2) an adaptive replication service that aggregates access telemetry and adjusts replica counts and placement using hysteresis thresholds and topology-aware selection. The contribution is a theoretical proposal and architectural blueprint; no prototype or experimental results are reported here. We discuss integration with analytical engines through two paths: a pragmatic FUSE mount that exposes IPFS content as a local filesystem to Spark/Flink, and prospective native connectors that parallelize block reads over the IPFS API. For tabular datasets, dataset metadata (schema, partitioning, file CIDs) is maintained in IPFS to support versioning and reproducibility. A plan for comparative evaluation versus HDFS, Ceph, and S3 (e.g., TPC-DS and subsets of Common Crawl) is outlined. Expected benefits are faster content discovery, higher throughput under skew and multi-tenant load, and improved resilience, with modest index/coordination overheads. The approach combines the openness of a decentralized P2P substrate with the manageability required by enterprise-scale analytics.

Keywords: dynamic indexing, IPFS, big data, replication, content-oriented indexing, high-performance storage.

Introduction

Modern Big Data storage solutions such as HDFS, cloud storage, and others, have a number of limitations in scalability, flexibility, and decentralization. The IPFS network offers content-addressable, decentralized storage capable of eliminating single points of failure and increasing data availability. However, IPFS in its basic implementation does not contain the means for efficient content indexing and dynamic data replication necessary for typical Big Data tasks. As a result, the problem of efficient storage and high-performance parallel access to big data in the IPFS environment has not been solved. This study aims to solve this problem by developing a new architecture that combines the capabilities of IPFS with additional indexing and replication mechanisms, which will allow IPFS to be used for Big Data tasks, providing both scientific novelty and practical value. The novelty includes the use of a content-based approach to metadata organization and adaptive replication management based on access profiles – such combinations have not previously been implemented in big data storage systems. The practical significance of the work is due to the potential reduction in the load on centralized nodes, increased speed of access to popular data and system resilience to failures due to the decentralization of storage.

The article is organized as follows. Section 1 surveys the current state of the field and limitations of existing storage systems. Section 2 reviews the architecture of IPFS and its applicability to Big Data. Section 3 presents the proposed approach, including content-oriented indexing,

[1]Moscow Polytechnic University, Moscow, Russia. E-mail: shevarev.max@mail.ru
[2]Candidate of Economics, Head of the Department, Professor, Moscow Polytechnic University, Moscow, Russia

adaptive replication, and integration with analytics frameworks. Section 4 introduces the reference systems for comparison. Section 5 describes the evaluation methodology, metrics, and scenarios. Section 6 outlines the test datasets. The Conclusion summarizes the study and points directions for further work.

1. Analysis of the Current State of the Field

Storage and processing of big data today. Big data is characterized by volume (petabytes), velocity of accumulation, and variety of structures. The traditional platform for Big Data is the Hadoop ecosystem, where HDFS (Hadoop Distributed File System) serves as a distributed storage system, and MapReduce or Apache Spark are used for data processing. HDFS splits files into blocks and replicates each block, typically three times on different cluster nodes [4, 8], to ensure fault tolerance. This strategy provides high throughput for sequential reading and "divide and conquer" writing, close to the performance of local disks on each node. In addition to HDFS, large infrastructures use distributed object storages such as Amazon S3, Ceph, as well as HPC-level file systems (e.g., Lustre, GlusterFS). Cloud storages (S3, Google Cloud Storage) have become popular due to their elasticity and integration with analytical engines (Presto, Hive, etc.), while Ceph offers unified object, block, and file storage for private clouds.

1.1. Technologies Used and Their Limitations

Each of the existing solutions has its limitations. HDFS relies on a central metadata node (NameNode) to store information about the file structure. This means potential non-scalability of metadata and the need for High Availability mechanisms for the NameNode; otherwise, a NameNode failure paralyzes the cluster. In addition, HDFS is designed for a "write once, read many" model: after writing, a file cannot be arbitrarily modified, only append and truncate operations are supported. This simplifies integrity maintenance but makes HDFS unsuitable for applications with frequent updates or random writes; such workloads are better supported by Ceph or traditional DBMSs. Replication in HDFS is fixed: the replication factor is the same for all blocks (by default, 3), without considering the actual popularity of the data. During peak demand for certain "hot" data, three copies may be insufficient (causing overload of nodes storing these blocks), while less popular data is stored in redundant three copies unnecessarily. The lack of a dynamic replication mechanism is a known drawback that research groups are trying to address. The literature proposes approaches where the number of HDFS block replicas changes depending on file popularity [4]. For example, X. Cao et al. describe an algorithm for predicting file "hotness" using a Markov model and adapting the number of copies proportionally to the forecast, which significantly increases cluster efficiency during surges in requests for the same data [4]. However, such solutions have not yet been implemented in mainstream HDFS versions and require external management.

Other systems also have limitations. Ceph is a fully distributed storage system with no single point of failure (metadata is distributed using the CRUSH algorithm), supports replication and erasure coding, and provides more flexible space utilization [9]. However, this flexibility comes at a cost: deploying and maintaining a Ceph cluster requires high expertise, and configuration is more complex compared to HDFS.

Amazon S3 and similar cloud storages provide high reliability through replication at the data center level, but when working with big data, other bottlenecks appear: access latency over the Internet, as well as charges for each request and data retrieval. In addition, S3 objects have eventual consistency for write operations, which complicates scenarios with frequent data updates. GlusterFS and other distributed file systems (Lustre, etc.) are often used in HPC and can provide high throughput, but their efficiency in Big Data scenarios also depend on the workload, and scaling to thousands of nodes is associated with challenges.

Thus, modern solutions are either centralized or require complex support, and do not dynamically adapt to changing workload profiles. This opens up opportunities for new architectures that combine decentralization and intelligent data management algorithms.

2. Architecture of IPFS and Its Applicability to Big Data

The InterPlanetary File System (IPFS) is a protocol and a peer-to-peer network for distributed data storage and delivery. IPFS uses a content-addressable model: each file (and its fragments) is assigned a Content Identifier (CID) – a cryptographic hash of its content. The network layer of IPFS is a distributed hash table (DHT) based on the Kademlia algorithm, where nodes store records about which CIDs they can provide [6]. To retrieve a file, a node queries the DHT to find which peers have the required CID and establishes direct connections to download fragments (blocks). A simplified P2P network with DHT is shown in Fig. 1.

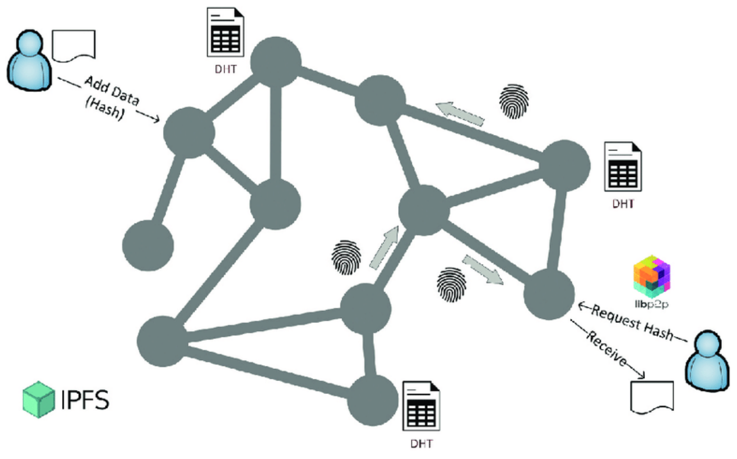

Figure 1. Simplified P2P network with DHT

Block transfer is performed using the IPFS-specific Bitswap protocol, similar to BitTorrent. Bitswap allows nodes to request missing blocks from multiple peers in parallel, theoretically scaling throughput: popular content can spread like a “swarm” in a P2P network, offloading individual nodes. IPFS also uses a Merkle DAG to organize files: each file is split into fixed-size blocks, and the hashes of the blocks form a tree structure (Merkle graph), enabling integrity verification and assembly of large files from parts.

The advantages of IPFS for big data include the absence of a central node: content is automatically duplicated on as many nodes as have downloaded it, and unified addressing by hash eliminates storage duplication – identical files are automatically the same, with no double storage. Content addressing also provides built-in data integrity verification, which is critical for large volumes where recalculating checksums is expensive. In addition, IPFS supports content versioning: a new file version receives a new CID, while the old one remains accessible by its previous hash, solving the lack of versioning typical for classic replicated storages [1]. This is important for reproducibility in big data experiments – IPFS can store multiple versions of a dataset in parallel, guaranteeing their immutability.

However, in practice, IPFS has serious limitations that hinder its direct application to Big Data. First, there is no guaranteed storage persistence: if data is not pinned on nodes, nodes may free up space and delete content, making it unavailable over time. IPFS lacks a built-in automatic replication mechanism: the network relies on either the user to ensure pinning on multiple nodes or other participants to download the data and thus automatically become replicas. If a file is not in demand, there is a risk that the only copy will disappear (the owner node disconnects or clears the cache). This is unacceptable for most Big Data tasks, where data must be stored reliably regardless of short-term interest. The solution in the IPFS ecosystem is IPFS Cluster: an add-on that coordinates a group of IPFS nodes and automatically replicates pinnable content among them.

IPFS Cluster maintains a global pinset and distributes objects across cluster nodes according to a user-defined replication factor, using Raft consensus for consistency between nodes [7]. If one of the cluster nodes fails, the Cluster can automatically re-replicate "orphaned" content to other nodes, maintaining the required number of copies [7]. Thus, IPFS Cluster provides fault tolerance, but replication in the Cluster is static – it is set by the number of copies, which does not change dynamically with the load. In addition, the Cluster assumes trust between nodes and is usually deployed as a private network (e.g., within a single organization), not leveraging the full global potential of the IPFS network. The architecture of an IPFS Cluster is shown in Fig. 2.

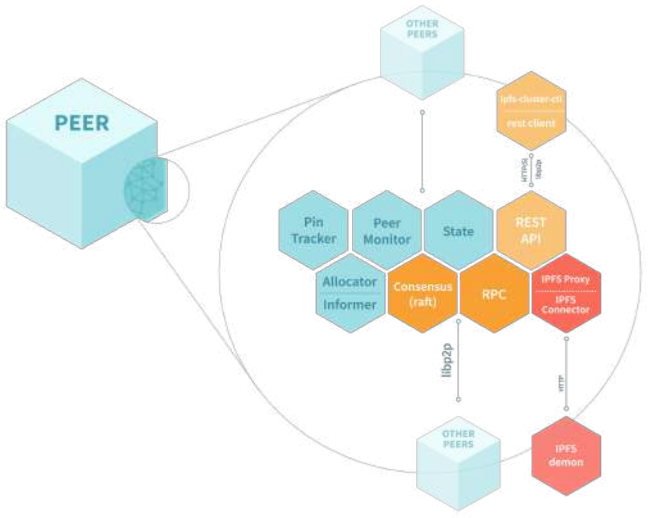

Figure 2. Architecture of an IPFS Cluster

The second problem is data search and indexing. In IPFS, a file can be efficiently retrieved only if its CID is known. Out-of-the-box search by name, keywords, or content is not supported. The global DHT provides hash-based routing but does not solve the problem of obtaining the hashes for the required data. In traditional big data systems, metadata (file names, table schemas, etc.) is managed centrally (e.g., HDFS NameNode stores the file directory, Hive Metastore stores table schemas and part locations on HDFS). In IPFS, there is no unified catalog of all content – this is a deliberate architectural decision (a network without a centralized index), but it complicates the use of IPFS in analytical tasks where, for example, it is necessary to find all files of a dataset by name pattern or perform a selection by attribute value.

Currently, there are experiments on building decentralized search systems on top of IPFS. One approach is to use two-level indexing with distributed index updates. For example, Ling Cao and Yue Li (2022) proposed a system where each node, when adding a file to IPFS, automatically extracts keywords and updates a local inverted index – a list of words and documents [3]. For each keyword, a separate index file is created – the first level – storing a list of CIDs of files containing this word. These index files themselves are placed in IPFS, receiving their own CIDs. Then, at the second level, a distributed index is built: an association "keyword $\rightarrow$ CID of the index file" [10]. This second-level index can be stored either in a distributed hash table or updates can be broadcast via the IPFS PubSub mechanism, using CRDTs to ensure consistency between nodes [10]. As a result, searching by keyword reduces to finding the corresponding index file via DHT and then loading the list of documents from IPFS. Experiments show that this approach can provide content search, but as the number of documents grows, the volume of index data also increases, requiring optimization and caching. Nevertheless, the work of Ling Cao demonstrates the fundamental feasibility of content-oriented indexing in IPFS and serves as a starting point for developing a custom indexing system in this study.

The third problem is data transfer performance. Despite the potential of P2P distribution, in reality, delays in IPFS can be higher than with direct access to high-speed cluster storage. Data requests in IPFS include latency for peer discovery via DHT (hundreds of milliseconds), then establishing multiple connections and exchanging blocks. If source nodes are geographically distributed or have limited bandwidth, downloading a large file may be slower than in the traditional "high-speed cluster" model. IPFS performance is highly variable: download speed can be inconsistent, especially for large files if slow or distant nodes are involved. On the other hand, with proper node organization, distributed caching can outperform centralized schemes in multi-node environments, but it requires protocol-level optimizations and careful tuning. For Big Data, it is likely necessary to combine the IPFS approach with data pre-placement on compute nodes (data locality) to reduce inter-node traffic. The next section discusses how IPFS can be integrated with frameworks like Spark to run computations close to the data or move data closer to computations.

In summary, current big data solutions suffer from centralization and static architectures, while IPFS offers the prospect of decentralized storage but requires mechanisms for indexing, reliable replication, and performance assurance. The following section formulates the proposed approach, taking these observations into account.

3. Proposed Approach: Content-Oriented Indexing and Adaptive Replication in IPFS

General architecture of the solution. The new system assumes the use of IPFS as the basic data storage layer (transport and distributed block storage), on top of which two key components operate: the first one is a content-oriented data indexing subsystem, and the second one is an adaptive replication module based on access profiles. In addition, an integration layer with analytics tools (Spark, Flink, etc.) is provided, ensuring seamless access for analytical jobs to data in IPFS via standard interfaces. These components are described below.

3.1. Content-Oriented Indexing over IPFS

The goal of this subsystem is to enable fast search and addressing of required data by content, metadata, or logical names, rather than by a hash unknown to users. The approach is based on content-centric indexing ideas. Importantly, indexing *does not imply centralization*: index artifacts are data structures (e.g., inverted lists, manifests) that are themselves stored and versioned in a distributed manner (e.g., in IPFS), and can be published/replicated without introducing a single authoritative catalog. A coordinating component may orchestrate publication for practicality, but the storage and retrieval model remains decentralized.

Each file in a large dataset, when added to the system, is accompanied by the generation of metadata for the index. Metadata may include: file name or logical identifier, size, data type/schema (e.g., CSV, Parquet – then the table schema), file hash (CID), and possibly keywords or content statistics. These metadata are registered in a distributed index. A two-level scheme is proposed, inspired by the work of Ling Cao (described above): at the first level, specialized index structures are formed, for example, for each attribute or word – a list of files containing the given attribute/word. At the second level, a global index catalog is maintained, linking the search key to the location of the corresponding index block.

A possible implementation is as follows. Let us suppose our dataset consists of many files, each described by a set of attributes (e.g., date, topic, source). For each attribute (or a combination like "field name=value"), a separate index document is created – a file containing a list of CIDs of all files matching this attribute/value. This index document itself is placed in IPFS and receives a CID_i. Then, at the second level, a mapping table "attribute $\to$ CID_i" is formed. This table can be stored in several ways:

- as a separate index file (a large JSON or a database like IPFS-Lite) with records for all attributes;
- distributed among nodes responsible for different attributes, using DHT/IPNS;
- duplicated on all nodes via CRDT (less scalable, but simple).

The most suitable is the DHT approach: we use the existing IPFS DHT, but publish attribute hashes as keys instead of content hashes. The IPFS DHT allows storing a small data block (usually up to 1 KB) in a record. Therefore, for each attribute (e.g., a keyword), it is possible to publish in the DHT a record: key = hash(attribute name), value = CID_i of the index file. Thus, any node, knowing the attribute, can use the DHT to find the CID of the index file and then download the index itself from IPFS.

To update indexes when new data is added, provider publication is used: when a new file is added, the initiating node either updates the corresponding index file (adds the new CID) and republishes it in IPFS, or creates a new index fragment. To avoid race conditions and ensure consistency during simultaneous additions, a CRDT model is applied: for example, each node maintains a local copy of the index file for each relevant key and exchanges delta updates via PubSub. Centralized coordination via a dedicated "catalog node" is also possible, but this introduces a single point of failure, which is undesirable. In this work, a basic variant is assumed: one node (the index coordinator) collects updates and periodically publishes new versions of index files. This compromise simplifies the initial implementation, and the coordinator's fault tolerance can be ensured by running its copies on different nodes, with leader election, for example, via Raft. This indexing component is content-oriented, as the key data identifier is not its location, but either a semantic feature (keyword, attribute) or a content hash (CID). A user or program can request data by semantic criteria, the system will find the required CIDs, and then IPFS will deliver the content.

It is expected that this approach will eliminate one of the obstacles to using IPFS for big data – the problem of searching and organizing large numbers of files. It certainly introduces overhead (storage of index structures, search time), but these costs are justified by the scale of the data: without an index, working with thousands of files in IPFS is impossible within a reasonable time.

3.1.1. Data model and supported data types

The system targets data that is either immutable or versioned and can be addressed content-wise. The primary supported types are:

- **Structured tabular datasets** (e.g., Parquet, ORC, CSV), accompanied by a schema and optional partitioning metadata. A dataset is represented by a manifest (JSON/protobuf) that lists file CIDs and logical partitions.
- **Large binary objects** (e.g., model checkpoints, archives, images, WARC segments) that are naturally chunked by IPFS into fixed-size blocks and referenced by a root CID.
- **Directory-like collections** of homogeneous files referenced from a root directory CID.

The system does not target arbitrary in-place mutable files. Updates are performed by publishing a new version (new CID/root) while previous versions remain accessible. For many small files, ingestion tools can optionally pack them into container objects (e.g., CAR-like bundles) to reduce per-file indexing overhead and improve read parallelism.

3.2. Adaptive Replication Algorithm Based on Access Profile

The second pillar of the proposed solution is dynamic management of the number and placement of data copies in the IPFS network based on access analysis. The idea is as follows: in Big Data workloads, access patterns are often highly skewed – a small portion of the data (hot data) is requested very frequently (e.g., recent logs, active database partitions), while "cold" archival data is accessed rarely. In traditional systems (HDFS, Ceph), the replication factor is usually fixed (three copies), and in object storages – two or three copies plus occasional caching via CDN. It is proposed to adaptively increase the number of replicas for hot objects to improve throughput and fault tolerance, and decrease the number of replicas for cold data to save space. This mechanism is similar to automatic caching, but differs in that the copies are full-fledged and are considered in task planning (unlike random cache, which the system may not be aware of).

Specifically, the following algorithm is proposed. The system monitors data requests: each node, when serving a request for a specific CID (block or file), signals the replication module about the access event. The module aggregates statistics for each large object (e.g., file or logical data partition). Periodically, at intervals of Δt (e.g., one hour), an estimate of popularity $P(obj)$ is calculated for each object over the recent time window. Metrics such as request frequency, number of unique nodes requesting the object, and volume of data transferred are used. Based on $P(obj)$, a decision is made to change replication:

a) If $P(obj)$ exceeds the upper threshold T_{hot} (the object is clearly hot) and the current number of replicas is less than some maximum R_{max}, the system initiates the creation of additional copies. How to create a copy in IPFS? In the global IPFS network, it is sufficient for another node to download this CID and pin it locally. Our module can send a command to specific nodes to perform pinning. Node selection for replication can take topology into account – for example, aiming to place copies in different data centers for resilience, or closer to expected consumers (if a particular analytics group frequently requests the data, replicate to their site). The selection algorithm is a separate task; in the simplest case, N nodes can be randomly chosen from the participants, excluding those already storing a copy.

b) If $P(obj)$ drops below the lower threshold T_{cold} and there are more than the minimum required R_{min} copies (e.g., one or two), some copies can be removed (unpin). Removal should be done cautiously: it may be advisable to reduce gradually and always keep at least one main copy.

c) For objects of medium popularity (between thresholds), leave as is, without changes.

Such a hysteresis with thresholds prevents oscillation in the number of replicas due to load fluctuations. An important aspect is that the access profile may change over time (e.g., data ages and becomes less used), so the algorithm should account for popularity decay. The literature describes approaches where popularity is predicted using machine learning or time series methods [4], but, in this study, a heuristic method is sufficient: a sliding average of requests over a window, with exponential decay of past counters.

Adaptive replication requires coordination: if each node decides independently which objects to replicate, conflicts may arise. Therefore, a centralized replication coordinator is introduced, which may be combined with the indexing coordinator or be separate. This coordinator collects the global popularity picture, for example, via periodic node reports or by subscribing to PubSub events indicating "interest in CID." It then calculates new replication levels and sends commands to nodes: "create a copy of CID X" or "it is safe to remove a copy of CID Y." Upon receiving such a command, a node either executes `ipfs pin add <CID>` (downloading the content) or `ipfs pin rm <CID>` (allowing the garbage collector to remove the content if needed). For reliability, the IPFS Cluster's API can be used to manage the pin-set on nodes, but this brings us closer to the IPFS Cluster architecture. The difference is that IPFS Cluster requires a fixed replication factor, while we change it dynamically. In principle, it is possible to build an additional layer on top of IPFS Cluster: let the Cluster assume that each pin has `replication_factor=`∞ (i.e., any node that wants to can pin), and we decide which nodes actually receive the pin command.

It is expected that adaptive replication will significantly improve throughput during mass simultaneous access to the same data, as the load will be distributed across more nodes (CDN effect). In addition, resilience increases: even if several nodes fail, a hot object likely had many copies and remains available. On the other hand, the downside is increased disk space usage for popular data. However, for Big Data, CPU/Memory or network are often the bottlenecks, while disks are relatively cheap; moreover, as data cools, excess copies will be removed, freeing up space.

3.2.1. Coordination model: centralized coordinator and path to decentralization

In the prototype, coordination (index publication and replication control) is centralized for pragmatic reasons: (i) predictable convergence and simpler reasoning about consistency; (ii) lower operational complexity and clearer auditability in multi-tenant settings; (iii) ease of enforcing policy constraints (quotas, placement rules) and reproducing experiments. The coordinator runs in a highly available configuration (leader election, write-ahead log, periodic snapshots) so that the *control plane* has no single point of failure in practice, while the *data plane* remains decentralized.

Limitations of this approach include tighter bounds on control-plane throughput and potential coupling to a trust domain. Our roadmap to decentralization foresees: (a) CRDT-backed index state with gossip-based dissemination over PubSub; (b) leaderless aggregation of access statistics using sketches and windowed counters; (c) conflict resolution via monotonic policies and attribute-level last-writer-wins; (d) eventual-consistency guarantees with bounded staleness for popularity signals. This evolution preserves the benefits of decentralized operation while maintaining verifiability and policy compliance required in enterprise environments.

3.2.2. Local block placement and device selection

To avoid I/O bottlenecks on a single physical drive, each node exposes a set of storage devices (mount points) with capacity and performance descriptors. For every block/object identified by a CID, the node selects a target device using capacity-aware rendezvous hashing (a.k.a. highest-random-weight), with per-device weights reflecting free space and service rate. This yields balanced distribution, minimal data movement when devices are added/removed, and awareness of heterogeneous media (e.g., SSD vs. HDD).

Replica placement applies an anti-affinity rule: multiple replicas of the same object are never co-located on the same device. For large sequential reads, adjacent blocks can be co-scheduled on the same device to exploit readahead, while concurrent readers are spread across devices to maximize queue depth utilization. These policies are enforced at the storage-adapter layer and do not require changes to IPFS content addressing.

3.3. Integration with Analytical Frameworks

For practical use of the proposed system, it is necessary to provide convenient access for analytical tools to data stored in IPFS. Most big data frameworks (Spark, Presto, Flink) can read data from HDFS, S3, or the local file system, but are not aware of IPFS. Two integration approaches are proposed: via FUSE mounting and via specialized connectors.

FUSE bridge: IPFS provides the ability to mount its content to the local file system (ipfs mount). For example, ipfs mount mounts /ipfs (content by CID) and /ipns (named content) into the file system tree. Knowing the CID of a dataset's root directory, it can be mounted and presented to Spark as a local directory. Then Spark (or Hadoop) will read files without knowing they come from IPFS. However, the standard FUSE implementation of IPFS is not designed for high-performance reading of huge files: it works sequentially and may incur context-switching overhead. Nevertheless, at the prototyping stage, FUSE is the simplest path. The plan is to mount the IPFS dataset on each Spark executor node. Thanks to adaptive replication, by the time a job starts, the required data is likely already replicated on those nodes (or nearby ones) that will read it – this is similar to data locality in Hadoop (when computation moves to the node with the data). Even if some data is not present locally, IPFS will fetch the missing blocks over the network.

In the future, a connector for Spark can be developed, implementing the Hadoop InputFormat interface, which will use the CID to access the IPFS API (HTTP gateway or go-ipfs library) for reading blocks. Such a connector could more efficiently parallelize reading by requesting different file fragments from different IPFS nodes. This is more complex but potentially faster, bypassing FUSE. Similarly, connectors can be developed for Flink or Presto.

Integration also includes a schema registry for tabular data: if the data is a table, e.g., Parquet files, it is necessary to store the table schema and partitioning. The indexing system can be extended: in addition to search indexes, dataset metadata can be maintained, similar to Hive Metastore, storing column descriptions, types, partitioning, and a list of CIDs for all file parts. This metadata file (JSON or protobuf) itself resides in IPFS and is versioned. The analytical engine (SparkSQL, Presto) can retrieve it via an IPNS link (a human-readable name pointing to the current metadata CID), thus always working with the up-to-date schema. In this work, we limit ourselves to describing the concept; implementation of the metadata repository is beyond the scope.

The uniqueness of the proposed method lies in combining content-dependent indexing with dynamic replication in a distributed system. These ideas have been seen separately (indexing for IPFS [10], dynamic replication for HDFS [4]), but their joint application in the context of IPFS for Big Data has not been described in the literature. The novelty is also in adapting IPFS, originally not intended for analytical clusters, by developing an additional data management layer. The proposed architecture essentially forms a hybrid of a decentralized P2P network with elements of centralized control (index and replication coordinators). This hybrid approach preserves the main advantage of IPFS – the absence of a single point of failure for the data itself – while introducing manageability characteristic of enterprise storage.

The practical significance of the solution is manifested in the potential to build inter-organizational data lakes based on IPFS. For example, several organizations can place large public datasets in IPFS; our system will ensure that these data are indexed for search and replicated where there is demand. This will facilitate data sharing without burdening a single storage center, eliminate duplication (thanks to content addressing), and reduce transfer costs – data will be loaded from the nearest nodes if they already have copies. For internal use, a company can avoid deploying expensive HDFS/Ceph clusters and instead use an IPFS cluster across different sites, automatically optimizing data placement for current tasks. In addition, replication flexibility will allow more efficient use of resources: unlike the rigid three-copy rule in HDFS, the system does not store unnecessary copies of cold petabytes, reallocating space for hot fragments.

The next section describes the plan for experimental evaluation of the proposed approach compared to traditional solutions.

4. Reference Systems for Comparison

Based on prevalence and architectural diversity, the following systems have been selected:

- **Apache Hadoop HDFS + Spark:** the classic big data cluster. It serves as a benchmark for sequential distributed processing. Configuration: three data replicas on HDFS (default), Spark deployed in Standalone or YARN mode. This allows us to assess how competitive our P2P system is compared to a time-tested technology.
- **Cloud storage S3 + Presto (Trino):** as a representative of object storage with an analytical engine. Presto (now Trino) is used for SQL queries on data in S3. It is interesting to compare query speeds in our system versus S3 (considering that S3 lacks data locality and each request pulls data from the cloud). Optionally, Amazon Athena (the same engine) can be used for purity.

- **Ceph (RADOS) + Spark:** CephFS or RADOS as the storage layer, Spark as the processing engine. Ceph will be configured for three replicas or equivalent fault tolerance via erasure coding. This will show how our system compares to a more modern distributed storage.
- **Lustre (parallel file system):** a widely used HPC parallel file system with dedicated metadata servers (MDS) and object storage servers (OSS). Lustre is included to represent a high-throughput baseline for large sequential scans and parallel I/O. Depending on resource availability, it may be evaluated on a small testbed; otherwise, it is listed for completeness with qualitative discussion, since deploying and tuning Lustre requires specialized infrastructure and administrative privileges.
- **(Optional) HDFS with dynamic replication:** if it is possible to implement or emulate a dynamic replication algorithm in HDFS (e.g., via HDFS Balancer or a third-party tool), the idea of adaptive replication in a centralized environment can be compared to the proposed approach. However, such a ready-made product is unlikely to be available for testing.
- **(Optional) Basic IPFS Cluster:** to see what improvements our extensions provide, a comparison can be made with "vanilla" IPFS Cluster, where the same data is stored with a fixed number of replicas. The difference will be due to indexing (Cluster lacks content search, only direct CID access) and replication dynamics.

For fairness, all systems will be deployed on similar hardware or in equivalent cloud conditions. The cluster size (number of nodes) and total storage volume will be the same to ensure a fair comparison. For example, we will use 10 nodes, each with 16 CPUs, 64 GB RAM, and a 1 Gbit/s network (for IPFS, 10G is possible, but then HDFS should also have 10G, otherwise the conditions differ).

It is expected that the system will demonstrate better fault tolerance (due to the absence of a NameNode and overall decentralization) and potentially higher throughput under a very large number of parallel requests (IPFS can scale content distribution via P2P, whereas HDFS, when a DataNode is overloaded, hits the network limit). We also expect acceleration under skewed workloads: when a small volume of hot data is requested frequently, adaptive replication should provide an advantage over static three copies [4]. On the other hand, for sequential reading of the entire dataset, traditional HDFS will likely perform as well or better – since it has been optimized for years, while IPFS has overhead. Success can be considered if our scan time is no more than 10–20% longer than HDFS. As for S3, we expect a significant advantage for our system in speed, especially for repeated queries, due to local copies and the absence of Internet latency.

Overall, the comparative analysis plan will reveal in which scenarios the proposed architecture truly outperforms existing solutions, and where it may lag and require further improvements.

5. Plan for Comparative Efficiency Analysis

To rigorously assess the advantages and limitations of the new system, a comparative experimental analysis must be conducted. The plan includes defining metrics, test scenarios, and selecting systems for comparison.

Efficiency metrics. The following key indicators are highlighted:

- **System throughput for reading and writing data.** For Big Data, the speed of reading large volumes is critical (how many MB/s or GB/hour the cluster can process). The aggregate speed of typical jobs (e.g., scanning the entire dataset) will be measured.
- **Average and maximum data access latency.** In a distributed environment, the time from data request to receiving the first byte and to receiving the last byte is important. In IPFS, there may be additional delays for peer discovery; these will be measured and compared to HDFS/S3.
- **Scalability:** how performance changes with the number of nodes and data volume. Linear scalability is important – e.g., when doubling the number of nodes, how much faster is processing.
- **Fault tolerance:** system behavior when part of the nodes fail (e.g., 10% or 30% of nodes go down during a job). The ability to continue operation without data loss and the time to restore replication will be evaluated.
- **Overhead:** the volume of service data (indexes, metadata) and traffic (e.g., how much extra traffic is generated by our replication algorithm compared to basic data transfer). CPU/memory usage on coordinating nodes will also be measured.
- **Execution time of typical analytical queries:** e.g., a SQL query with filtering, an aggregate query, machine learning on a dataset. This is a complex metric, depending on both storage and computation, but it is important to see if the ultimate goal – faster data analysis – is achieved.

According to these metrics, the new system will be considered efficient if:

- The throughput of reading/analysis is not lower, and preferably higher, than that of the compared systems (e.g., more GB/s per node).
- Access latency is acceptable (not much higher than HDFS). Our system may be slightly worse in latency due to DHT lookup, but this should be offset by parallel delivery. If the first-byte latency is no more than twice that of HDFS/S3, this is acceptable; the full scan latency should be comparable or better.
- In case of failures, the system does not lose data (this is a mandatory criterion) and continues to deliver it with minimal delays. Ideally, a 30% node failure should slow down operation by no more than $X\%$ compared to normal.
- The volume of index data and additional storage does not exceed reasonable limits, say, less than 10% of the main data volume (i.e., our index and metadata do not create significant redundancy).
- Infrastructure cost under equal conditions is not higher: if our system requires significantly more nodes or resources, this is a drawback. We assume similar nodes are used. The IPFS system has the advantage that nodes from different organizations can participate without a single storage center.

Test scenarios and workload. To comprehensively test the system, several types of tests are planned:

1. **Sequential scan of the entire dataset** – simulates a large batch analytical query (e.g., "compute statistics for the entire dataset"). The total execution time on our system and on the comparison systems will be measured. This will show the pure throughput of the disk subsystem and network.

2. **Multiple parallel queries to different data parts** – simulates the situation of several analyst teams or services working simultaneously. This will reveal the system's ability to share resources and parallelize without conflicts.
3. **Hot data:** a scenario where 20% of the data is requested very frequently, the remaining 80% rarely. It is checked whether replication adapts – i.e., whether access to hot data accelerates over time compared to cold data. In our system, this should be evident: the first requests to a hot object are slower (few copies), later – faster (replicas added). In HDFS/standard systems, speed is fixed regardless of access frequency. We will compare the average access time to frequently requested files at the beginning and end of the test.
4. **Fault tolerance:** a long-running job is started, and during its execution, random nodes are "shut down". We will measure whether the job completes successfully, how many tasks (Spark tasks) were restarted, and compare with HDFS/Spark when a DataNode fails – there should also be resilience, but it is interesting to compare recovery time.
5. **Indexing overhead:** the time to add a new file to the system with index update is measured. In HDFS, this is a file creation operation via NameNode – usually milliseconds; in our system – hash calculation, index update (may be hundreds of ms). It is acceptable if our delay is slightly higher, as indexing is not on the critical path of query execution, but we must ensure that even adding thousands of files does not "overload" the index coordinator.

6. Test Datasets

To evaluate the system, a representative large public dataset is required, sufficient in volume and complexity. Selection criteria: volume of at least several terabytes (to reveal scalability issues), presence of structure or content diversity, open availability and recognition, so that experimental results are reproducible by the community. Several candidates are considered:

- **Common Crawl** [5] – an open archive of web pages (HTML, WARC files) totaling tens of petabytes accumulated over many years. For our purposes, it is too large, but a subset can be used. For example, a monthly Common Crawl dump is about 100 TB of uncompressed data; a 1–5 TB sample is sufficient. The structure is many compressed WARC files (1 GB each), with diverse content (text, HTML). This dataset is good for testing with a large number of files and content-based indexing (e.g., by words). However, it is more difficult to analyze semantically (raw web data).
- **OpenStreetMap (OSM)** – global geodata describing the world map. The full OSM dump (planet file) is about 100 GB (XML or PBF format), which is relatively small; but derived data can be generated, such as tiles, indexes, geo-queries. OSM is well-structured (has schemas), allowing attribute-based search (roads, buildings, etc.). However, the volume may be insufficient for stress tests (hundreds of GB, not terabytes).
- **Enron Email Dataset** – a corpus of 0.5 million emails (tens of GB). Interesting for content analysis (text, communication graphs), but too small in volume, processed too quickly on a modern cluster, and will not reveal scalability issues.
- **CERN Open Data** – datasets from the Large Hadron Collider (LHC), particularly from CMS and ATLAS experiments. These data weigh many terabytes and have complex structure (ROOT files, particle event sets). The plus is realistic scientific Big Data, used to test distributed analysis (CERN uses systems like XRootD, similar to HDFS). The downside is the specific format, difficult to interpret without physics knowledge; however, for infras-

tructure testing, this is not a problem, as speed can be measured without understanding the content.

- **Generated data (TPC-DS benchmark)** – an artificial dataset can be generated according to the TPC-DS standard, e.g., at 10 TB scale. It is structured (simulates retail data: sales, products, customers, etc.) and widely used for comparative testing of SQL engines. The advantage is that identical SQL queries can be run on different systems and execution times compared. This option is good for evaluating analytical performance, though it is not a "real" external dataset.

Given the goals, the best choice appears to be a limited-volume Common Crawl in combination with a generated TPC-DS dataset. Common Crawl will allow testing on unstructured data and will check content-based indexing capabilities (e.g., searching for web pages containing specific terms). TPC-DS (10 TB) will provide a structured workload for SQL queries and comparison with benchmark results (it is known how long Spark or Presto take on these benchmarks).

Both can be used: first, configure the system on Common Crawl (e.g., 1 TB of data, 1000 files of 1 GB each) and test search/replication on it; then, on the same cluster, place TPC-DS (in Parquet format, partitioned by folders) with a volume of 10 TB and run the standard set of 99 TPC-DS queries in our system vs. competitors.

Specifically, the plan for Common Crawl: take, for example, the July 2022 Crawl, selectively download 1000 WARC files (1 GB each). Upload them to IPFS, record the CIDs. Set up indexing: for example, index by domains or frequently occurring words. Check that a search for "pages containing the word COVID" finds the corresponding CIDs via our index, and then these pages can be quickly retrieved. Also, simulate multi-user access: parallel requests for the 100 most popular WARC files – see if adaptive replication manages to distribute their copies to different nodes by the time the second/third user requests the same file.

Ultimately, the main focus of testing will be on TPC-DS 10 TB as a standardized benchmark. TPC-DS consists of fact tables (sales, 60 billion records) and dimension tables (products, stores, etc.). In Parquet format, the total volume is 10 TB. It is well suited because:

- it is widely known,
- contains both several very large files (fact tables) and many small ones (e.g., date table – 365 records),
- allows for complex SQL queries, forcing the system to read data with filtering, join, and aggregate – close to real retail analytics scenarios.

This dataset is structured, and for it, the indexing mechanism will be used for "dataset metadata": we will register the schema and possibly build an inverted index by some attribute (e.g., by year for the sales table, to quickly find all partition files for a year).

The chosen volume (10 TB) is large enough that a scan job on a 10-node cluster will last several minutes, allowing us to measure the difference. But not so large as to be unmanageable (the data can be stored and copied). If time and resources permit, 30 TB (TPC-DS scale 30TB) can be attempted.

Justification for dataset suitability: TPC-DS is the de facto standard for big data testing, so results will be understandable to the community and easily comparable with published results for Spark, Presto, etc. Common Crawl complements it by testing functions specific to IPFS (e.g., full-text search on unstructured content, which TPC-DS does not cover). Moreover, Common Crawl has immediate practical relevance: if our system can efficiently store a web archive, it demonstrates suitability for real scenarios where data is not neat tables but heterogeneous files.

In sum, testing on these datasets will demonstrate both the versatility (working with tables and texts) and scalability of the proposed architecture. Successful completion of the tests will confirm that content-oriented indexing and adaptive replication enable the IPFS network to evolve into a Big Data platform, combining the strengths of decentralization with the requirements of high-performance analytics.

Conclusion

This work proposes a new architecture for big data storage and analysis, combining the decentralized IPFS network with mechanisms for dynamic content-oriented indexing and adaptive replication. The analysis shows that existing approaches (HDFS, Ceph, S3), while having become industry standards, have significant limitations in flexibility and require substantial resources for scaling, whereas IPFS offers a promising foundation due to content addressing and P2P interaction, but needs extended functionality for Big Data applications. The proposed solution fills this gap: the content-oriented index provides efficient search and organization of data by content, and the adaptive replication algorithm distributes data copies optimally according to current load, improving system performance and reliability.

The scientific novelty of this work lies in the integration of indexing and search methods with the distributed hash tables of IPFS and in introducing self-* properties into storage – the system autonomously responds to access patterns, reallocating resources. The practical value is confirmed by scenarios where decentralized storage is essential: inter-organizational data exchange, open data publication, fault-tolerant storage without a single owner. The proposed architecture enables such systems to be built without strict dependence on cloud providers and with savings on redundant data copies.

Future work includes completing the prototype and conducting the planned comparative analysis. The empirical evaluation is outside the scope of this paper and will be reported separately. The knowledge gained will serve as a basis for further optimization – in particular, improving replica placement algorithms, possibly considering network topology (to minimize latency), and implementing a fully decentralized indexing variant using CRDT, eliminating even the index coordinator as a point of failure. Another direction may be integration with blockchain technologies for access auditing or storage incentives (e.g., Filecoin) [2].

In summary, dynamic content-oriented indexing and replication based on IPFS represent a step toward the next generation of Big Data systems, combining the scalability and openness of peer-to-peer networks with the requirements of enterprise storage. The proposed architecture is both scientifically novel and practically significant, and its successful implementation could greatly expand the applicability of IPFS and similar decentralized technologies in the era of big data.

Acknowledgements

The author expresses gratitude to the scientific advisor Stanislav V. Suvorov for consultations and support during this work.

References

1. Benet, J.: IPFS - content addressed, versioned, P2P file system. `https://doi.org/10.48550/arXiv.1407.3561`

2. Benet, J., Greco, N., *et al.*: Filecoin: A decentralized storage network. Protocol Labs report, 2017.

3. Cao, L., Li, Y.: IPFS keyword search based on double-layer index. In: Proceedings of the International Conference on Electronic Information Engineering and Computer Communication (EIECC 2021), vol. 12172, pp. 1217209. SPIE (2022). `https://doi.org/10.1117/12.2639406`

4. Cao, X., Wang, C., Wang, B., He, Z.: A method to calculate the number of dynamic HDFS copies based on file access popularity. Mathematical Biosciences and Engineering 19(12), 12212–12231 (2022). `https://doi.org/10.3934/mbe.2022583`

5. Common Crawl Foundation. Common Crawl - Open Web Data (HTML, WARC files). `https://commoncrawl.org/` (2025), accessed: 2025-05-25

6. Maymounkov, P., Maziéres, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol. 2429, pp. 53–65. Springer, Berlin, Heidelberg (2002). `https://doi.org/10.1007/3-540-45748-8_5`

7. Estrada-Galiñanes, V., ElRouby, A., Theytaz, L.: Towards efficient data management for IPFS-based applications. `https://doi.org/10.48550/arXiv.2404.16210`

8. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010, pp. 1–10. IEEE (2010). `https://doi.org/10.1109/MSST.2010.5496972`

9. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: Grid resource management - CRUSH: controlled, scalable, decentralized placement of replicated data. In: Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA, pp. 122. ACM (2006). `https://doi.org/10.1145/1188455.1188582`

10. Zhu, Z., Cen, F.: Research on key technologies of information search based on IPFS. In: International Conference on Electronic Information Engineering and Computer Communication (EIECC 2021), pp. 1217201. SPIE (2022). `https://doi.org/10.1117/12.2640819`