University of Camerino

Master Degree in Computer Science

Technologies for Big Data Management

DSTORAGE#JUICEFS

Students

Vlad Dogariu
vlad.dogariu@studenti.unicam.it

Daniele Porumboiu
daniele.porumboiu@studenti.unicam.it

Mounir Taouafe
mounir.taouafe@studenti.unicam.it

Supervisor

Massimo Callisto De Donato

A.A. 2022/2023

1. Abstract

This report provides an overview of the various technologies used for processing and storing IoT data, including Apache Kafka, MQTT broker, Redis, Minio, Java producer and consumer, JuiceFS and Spark Analysis. This report presents the evaluation of the distributed storage technology, JuiceFS, for use in data analytics jobs.

The objective of the project was to analyze the features of JuiceFS and compare it with HDFS and Amazon S3. The students were required to provide a demonstration of using JuiceFS to store data, load the data from Spark, and perform some data analysis on the loaded data. The project was implemented using Kafka as a message buffer. The data was stored in JuiceFS, and Spark was connected to JuiceFS to perform standard analysis. The students tested the support for append in JuiceFS, and checked the compatibility with compressed file formats, such as Parquet files or Avro. The results of the evaluation showed the potential of JuiceFS as a distributed storage technology for data analytics jobs.

Contents

Listings

List of Figures

List of Tables

2. Introduction

The Internet of Things (IoT) is a rapidly growing technology that connects devices, appliances, and sensors to the Internet, allowing for real-time data collection and analysis. The large amounts of data generated by IoT devices pose significant challenges for data processing, storage, and analysis. In order to effectively handle IoT data, it is essential to use the right tools and technologies.

Data storage has become an important aspect of modern data analytics. With the rapid growth of data, it is crucial to have a scalable and reliable data storage solution that can accommodate the needs of data analytics jobs. The objective of this project is to evaluate a new and promising distributed storage technology called JuiceFS. The goal of this project is to provide a comprehensive analysis of JuiceFS and its features, and compare it with other well-known solutions such as HDFS and Amazon S3 API. Additionally, the project will demonstrate the connection of JuiceFS with Apache Spark and the implementation of data analysis over the stored data.

The report concludes with a discussion of the factors to consider when choosing a data storage solution for IoT applications, including performance, scalability, and cost. With the right tools and technologies, it is possible to effectively handle the large amounts of data generated by IoT devices and turn it into valuable insights.

2.1 Application Domain

This report focuses on the application of IoT data processing and storage technologies in the context of big data and real-time data processing. The technologies discussed in this report are commonly used in the following application domains:

- **Industrial Internet of Things (IIoT):**
  The use of IoT technologies in the industrial sector, including the monitoring and control of manufacturing processes, supply chain management, and predictive maintenance.
- **Smart Home:**
  The use of IoT technologies in the home, including the automation of lighting, heating, and cooling systems, security systems, and entertainment systems.
- **Healthcare:**
  The use of IoT technologies in the healthcare sector, including remote patient monitoring, telemedicine, and health informatics.
- **Transportation:**

The use of IoT technologies in the transportation sector, including the monitoring and control of vehicles, traffic management, and logistics management.

- **Agriculture:**
  The use of IoT technologies in the agriculture sector, including precision agriculture, livestock monitoring, and weather monitoring.

- **Energy:**
  The use of IoT technologies in the energy sector, including smart grids, renewable energy management, and energy efficiency.

The report provides an understanding of the role of IoT data processing and storage technologies in these application domains, highlighting the importance of real-time data processing and storage in these areas. The report also highlights the challenges associated with the processing and storage of IoT data.

2.2 Project Goals

The objective of the project is to evaluate a distributed storage technology based on JuiceFS, with a focus on its features and suitability for data analytics jobs. The main goals of the project are as follows:

1. The project will evaluate the various features of JuiceFS, including its architecture, scalability, security, and reliability, among others. This will be done through a comprehensive analysis of the JuiceFS documentation and hands-on experimentation.

2. The project will compare the features of JuiceFS with HDFS and the Amazon S3 API. This will include a comparison of the performance, scalability, security, and reliability of each technology.

3. The project will explore the connection between JuiceFS and Apache Spark, with a focus on the ease of use and the performance of the integration.

4. The project will demonstrate the use of JuiceFS for storing data and performing data analysis. This will include the use of Spark to load data from JuiceFS and perform some standard data analysis over the loaded data.

In addition to these main goals, the project will also explore the use of Kafka as a message buffer in the implementation demonstration. This will involve evaluating the support of JuiceFS for append operations, as well as its support for compressed file formats such as parquet and avro.

2.3 Developement Technique

The development of the project will involve the following steps:

- Analysis of JuiceFS features:
  The first step will be to understand the features and capabilities of JuiceFS, including its architecture, data storage mechanisms, and data access methods.

- Comparison with HDFS and Amazon S3 API:
  After gaining a clear understanding of JuiceFS, we will compare it with the popular data storage solutions HDFS and Amazon S3 API to identify the strengths and weaknesses of each system.

- Implementation of the Prototype:
  The implementation of the prototype will involve the use of Apache Kafka as a message buffer, with the persistor subscribing to new messages in Kafka, storing them in JuiceFS, and publishing writing confirmation messages back to Kafka. Spark will connect to JuiceFS to perform data analysis.

- Data analysis:
  Once the data has been stored and retrieved, data analysis will be performed using Spark.

- Compressed file format support:
  We will check if JuiceFS supports compressed file formats, such as parquet and Avro, and if it is possible to store and retrieve data in these formats.

- Demonstration:
  A demonstration of the prototype will be performed to show the integration of JuiceFS with Apache Spark and Kafka, and to demonstrate the ability to store and retrieve data, as well as perform data analysis.

This project will be developed using a combination of big data frameworks and tools, including Apache Spark, Apache Kafka, and JuiceFS.

2.4 Report Structure

//TODO

3. Project Structure

The project focuses on the evaluation and implementation of JuiceFS as a distributed storage technology for IoT data. The project is structured into four main stages:

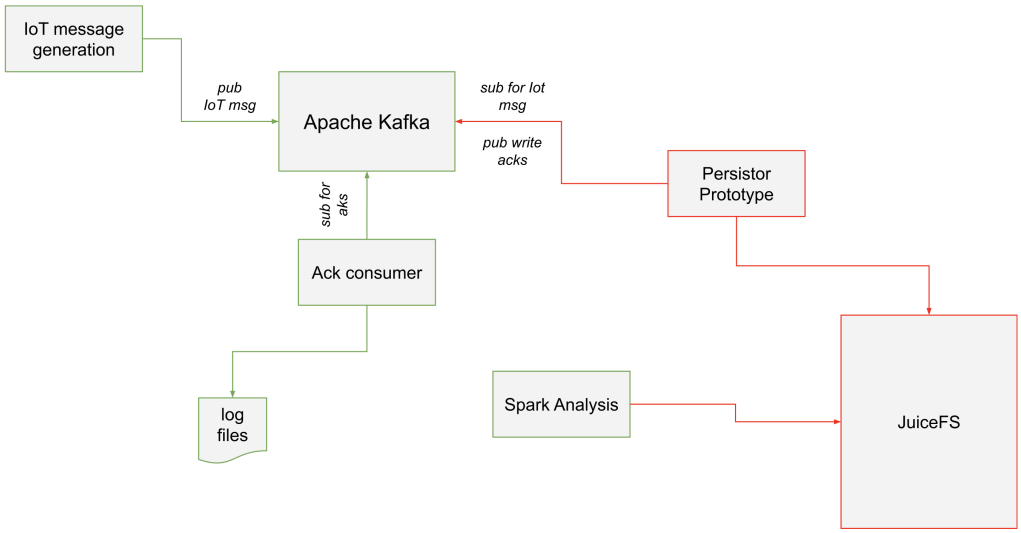

Figure 3.1: Project Structure

1. The first stage involves the generation of IoT messages which will be sent to Apache Kafka for further processing.
2. Apache Kafka acts as a message buffer and enables the transfer of messages from the IoT devices to the Java Consumer.
3. The Java Consumer subscribes and consumes the messages from Apache Kafka and sends back from the Producer the acknowledgement (ack) messages which will be stored in log files. The Consumer also stores the IoT messages in JuiceFS.
4. JuiceFS is the main component of the project and serves as the distributed storage technology for the IoT data. The data stored in JuiceFS will be analyzed and processed to support data analytics operations.

In addition, the implementation will also involve a comparison of JuiceFS with other storage technologies such as HDFS and Amazon S3 API to evaluate its performance and capabilities. The implementation will also demonstrate the connection of JuiceFS with Apache Spark to perform data analysis operations on the stored data.

4. Apache Kafka

Apache Kafka is an open-source, distributed streaming platform that provides a number of benefits for organizations looking to handle real-time data streams in big data and real-time data processing applications. Whether you are looking to process data from IoT devices, to handle data pipelines in big data processing applications, or to provide real-time data feeds for data visualization and analysis, Apache Kafka is a highly effective solution that is well-suited to meet your needs.

Apache Kafka is widely used in a variety of application domains, including the industrial Internet of Things (IIoT), smart homes, healthcare, transportation, agriculture, and energy. Apache Kafka is often used to process real-time data streams from IoT devices, to handle data pipelines in big data processing applications, and to provide real-time data feeds for data visualization and analysis.

4.1 Architecture

Apache Kafka is designed as a distributed system, with each node in the cluster responsible for a portion of the data stream. The nodes in a Kafka cluster are organized into topics, which are collections of records that are written and readed from the cluster. Each record in a topic consists of a key, a value, and a timestamp.

4.2 Features

Apache Kafka provides a number of key features that make it well-suited for handling real-time data streams, including high throughput, low latency, and high reliability. Additionally, Apache Kafka provides a number of tools for data processing, including a publish-subscribe model, a distributed log architecture, and support for parallel processing.

4.2.1 Consumers

Consumers in Apache Kafka are responsible for consuming messages from one or more topics and processing them as per the requirement. In Kafka, consumers are organized into consumer groups, where each consumer in a group is assigned a unique partition of the topics to be consumed. This way, the processing of messages can be done in parallel, thereby increasing the overall processing speed and capacity. By dividing the processing of messages into multiple consumer instances, the system can handle larger loads and higher message rates, thereby improving the overall performance. Additionally, parallel

processing can help in the effective utilization of resources, as the load can be balanced between multiple consumers

4.3 MQTT broker

MQTT (Message Queuing Telemetry Transport) is a popular publish-subscribe communication protocol that is widely used in IoT applications. Apache Kafka provides native support for MQTT, allowing MQTT data to be processed and stored using the Kafka platform. This provides a number of benefits for organizations looking to handle MQTT data streams, including scalability, reliability, and the ability to process large amounts of data in real-time. The MQTT protocol in Apache Kafka works by subscribing to topics and consuming the messages that are published to those topics. The messages are then processed by the Kafka platform, which can perform a variety of operations, including filtering, transforming, and storing the data. This provides a powerful solution for handling MQTT data streams in real-time, allowing organizations to perform complex data processing operations and store large amounts of data in a scalable and reliable manner.

In conclusion, Apache Kafka provides native support for MQTT, providing a powerful and flexible solution for handling MQTT data streams in real-time. Furthermore, Kafka is widely adopted and has a large and active community of users and contributors, making it well-supported and easy to integrate with other technologies.

4.4 IoT Simulator

IoT devices are often used to generate large amounts of real-time data streams that are difficult to handle using traditional data processing methods. Apache Kafka provides a powerful solution for receiving and processing data from IoT devices, including data generated by an IoT data simulator. An IoT data simulator is a tool that is used to generate realistic data streams from IoT devices for testing and development purposes. The Kafka broker publishes the data to a specific topic, which can be consumed by a Java consumer or other data processing tools.

5. Persistor Prototype

In this chapter, we will describe a Persistor Prototype that has been implemented in Java as a producer and consumer connected to Apache Kafka and JuiceFS. This prototype is designed to demonstrate the feasibility of using Hadoop, Apache Kafka, and JuiceFS to receive, process, and store real-time data from IoT devices.

The Persistor Prototype is a data pipeline architecture used to store IoT JSON messages. It is composed of two main components: the Producer and the Consumer. The Consumer is responsible for consuming the IoT JSON messages from the Kafka topic and storing them in JuiceFS or consuming the messages from the ack topic and using another method to create a log file after the ack with the UUID of each file. The Producer is responsible for sending the ack to Kafka.

JuiceFS provides Hadoop-compatible File System by Hadoop Java SDK. Various applications in the Hadoop ecosystem can smoothly use JuiceFS to store data without changing the code.

5.1 Configuration

Kafka and JuiceFS configurations are required to use the Persistor Prototype, as shown in the following code snippets:

```
1 //Configuration for Kafka Consumer
2       String bootstrapServers = "127.0.0.1:9092";
3       String groupId = "kafka";
4       List<String> topics = new ArrayList<String>();
5       String topicMqtt = "mqtt.echo";
6       String topicAck = "ack";
7       topics.add(topicMqtt);
8       topics.add(topicAck);
```

Listing 5.1: Configuration of Kafka

```
1 //Configuration for connecting to JuiceFS using Hadoop Compatible SDK
2       Configuration conf = new Configuration();
3       conf.set("fs.jfs.impl", "io.juicefs.JuiceFileSystem");
4       // JuiceFS metadata engine URL
5       conf.set("juicefs.meta", "redis://127.0.0.1:6379/1");
6       Path p = new Path("jfs://myjfs/myjfs");
7       FileSystem jfs = p.getFileSystem(conf);
```

Listing 5.2: Configuration of JuiceFS

5.2 Consumer and Producer

The Java consumer is a component of the Persistor Prototype that is responsible for receiving data from the Apache Kafka broker, and processing it. The consumer is listening on two topics, "mqtt.echo" and "ack". In the prototype, the consumer receives real-time data from the IoT data simulator, which is then processed and stored in JuiceFS, a distributed file system. If the message coming in is from the mqtt.echo topic, the Consumer creates a path based on the year, month, and day and saves the file in that directory. In this way, messages are saved according to their date. Prior to saving the file, a random UUID assigned to the filename is generated, while the contents of the file is the contained message. The Java Kafka Consumer can be created using the KafkaConsumer class, which requires a set of configuration properties to connect to a Kafka broker. These properties include the Kafka broker URL, the deserializer class for the message key and value, and the offset reset policy. In addition, the KafkaConsumer class provides the poll() method to read messages from a Kafka Topic. The following code snippet will illustrate the behavior of the consumer:

```
1  // Create consumer configs
2  Properties properties = new Properties();
3  properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
4     bootstrapServers);
5  properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
6     StringDeserializer.class.getName());
7  properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
8     StringDeserializer.class.getName());
9  properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, groupId);
10 properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
11
12 // Create consumer
13 KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
14 // get a reference to the current thread
15 final Thread mainThread = Thread.currentThread();
```

Listing 5.3: Configuration of Consumer

```
1  // subscribe consumer to our topic(s)
2  consumer.subscribe(topics);
3
4  // poll for new data
5  while (true) {
6     ConsumerRecords<String, String> records =
7        consumer.poll(Duration.ofMillis(100));
8
9     for (ConsumerRecord<String, String> record : records) {
10       String recordTopic = record.topic();
11
12       if (recordTopic.equals(topicMqtt)) {
13          LocalDate currentDate = LocalDate.now();
14          String datePath =
15             currentDate.format(DateTimeFormatter.ofPattern("yyyy/MM/dd/"));
16          UUID filename = UUID.randomUUID();
17          Path filePath = new Path(datePath + filename.toString());
18
19          FSDataOutputStream outputStream = jfs.create(filePath);
20          outputStream.writeChars(record.value());
21          outputStream.close();
22
23          produce(filename.toString());
24       }
25
26       if (recordTopic.equals(topicAck)) {
```

```
27              logToFile(record.value());
28          }
29      }
30  }
```

Listing 5.4: Consumer Behavior

Once saved, the Producer method is called and sends Kafka on topic ack the name of the saved file. The Java Kafka Producer can be created using the KafkaProducer class, which requires a set of configuration properties to connect to a Kafka broker. These properties include the Kafka broker URL, the serializer class for the message key and value, and the partitioner class. In addition, the KafkaProducer class provides the send() method to send messages to a Kafka Topic. The following code snippet will illustrate the behavior of the producer:

```
1   // Create the Producer
2   KafkaProducer<String, String> producer = new KafkaProducer<>(propertiesProd);
3
4   // create a producer record
5   ProducerRecord<String, String> producerRecord =
6      new ProducerRecord<>("ack", message);
7
8   // send data - asynchronous
9   producer.send(producerRecord);
10
11  // flush data - synchronous
12  producer.flush();
13
14  // flush and close producer
15  producer.close();
```

Listing 5.5: Producer Behavior

Meanwhile, if a message is sent on the topic ack, the Consumer calls a method that generates a log file. This log file contains all the names of the saved files. The following code snippet will illustrate the behavior of the method:

```
1   public static void logToFile(String message) {
2
3      String projectPath = Paths.get("").toAbsolutePath().toString();
4      String path = projectPath + "/output.txt";
5
6      try {
7         FileWriter writer = new FileWriter(path, true);
8         BufferedWriter logWriter = new BufferedWriter(writer);
9
10        logWriter.append(message + "\n");
11        logWriter.close();
12
13     } catch (IOException e) {
14        e.printStackTrace();
15     }
16  }
```

Listing 5.6: Logfile Method Behavior

6. JuiceFS

JuiceFS is a distributed file system that is designed to be fast, efficient, and scalable. It provides a flexible and powerful solution for storing and retrieving large amounts of data, making it well-suited for use in a variety of applications, including IoT data processing. One of the key benefits of JuiceFS is its ability to scale horizontally, allowing it to accommodate growing amounts of data without sacrificing performance. This makes it an ideal choice for use in IoT environments, where the amount of data generated by devices can be substantial. In addition to its scalability, JuiceFS also provides a number of features that make it well-suited for use in IoT environments. In the Persistor Prototype, JuiceFS is used to store data that is received from Kafka and processed by the Java consumer. This provides a flexible and scalable solution for storing large amounts of data, and it allows the Persistor Prototype to handle increasing amounts of data over time.

6.1 Architecture

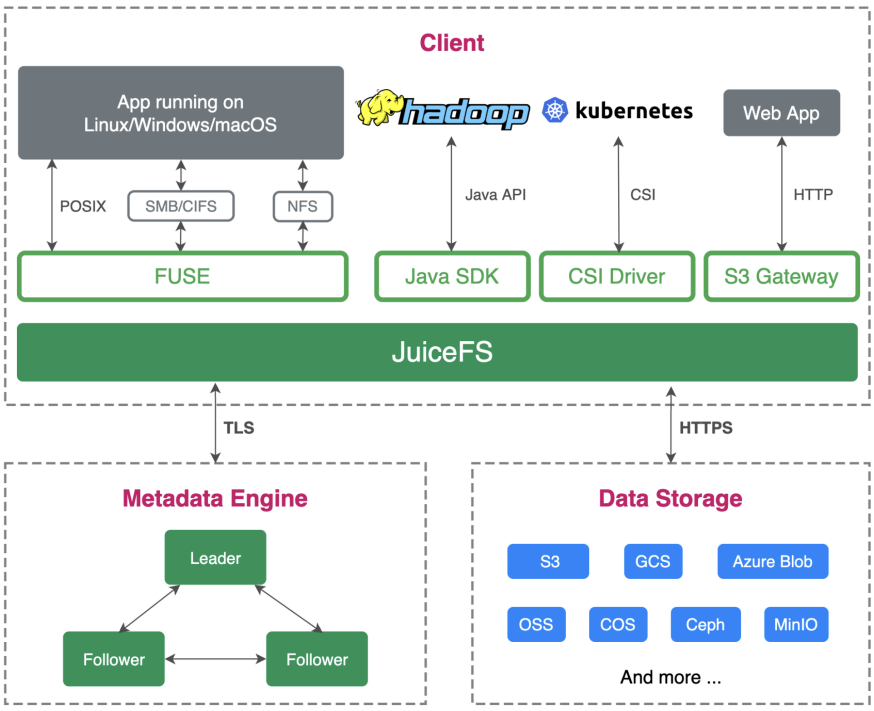

Figure 6.1: JuiceFS Architecture

As you can see in the figure 6.1, the JuiceFS architecture consists of three parts:

- **JuiceFS Client:**
  All file I/O happens in JuiceFS Client, this even includes background jobs like data compaction and trash file expiration. So obviously, JuiceFS Client talk to both object storage and metadata service. A variety of implementations are supported.

- **Data Storage:**
  File data will be split into chunks and stored in object storage, you can use object storage provided by public cloud services, or self-hosted, JuiceFS supports virtually all types of object storage, including typical self-hosted ones like OpenStack Swift, Ceph, and MinIO.

- **Metadata Engine:**
  The high performance metadata storage of JuiceFS uses a multi-engine design. In order to create an ultra-high-performance cloud-native file system, JuiceFS first supports Redis, an in-memory Key-Value database, that we choosed for this project.

6.2 Storage

The strong consistency and high performance of JuiceFS is ascribed to its special file management model. Traditional file systems use local disks to store both file data and metadata, while JuiceFS formats data first and then stores them in object storage, with the corresponding metadata being stored in a dedicated metadata engine.

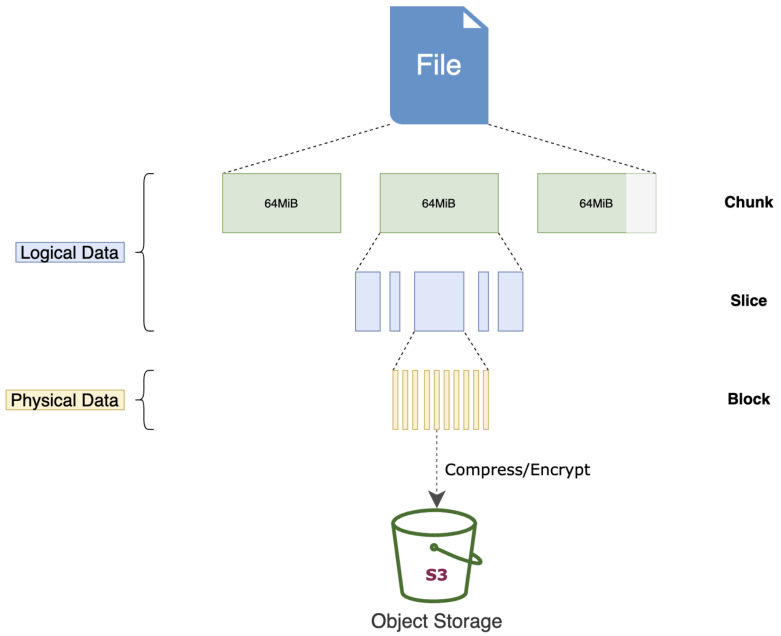

Figure 6.2: JuiceFS storage architecture

As shown in the figure 6.2, JuiceFS utilizes a unique file storage structure in which

each file is divided into one or multiple "Chunk(s)", each with a size limit of 64 MiB. These Chunks are further divided into "Slice(s)" which serve to optimize different types of write operations. The length of each Slice varies based on the method of file writing. These Slices are then divided into "Block(s)" of size 4 MiB by default. The Blocks are ultimately stored in object storage as the basic unit of storage. As a result, you won't be able to locate the original file directly in the object storage, instead, you will find a "chunks" directory and various numbered directories and files within the bucket. But don't be alarmed, this is the standard format in which JuiceFS stores its data. Meanwhile, the relationships between the file, Chunks, Slices, and Blocks are recorded in metadata engines. This detached design contributes to JuiceFS' high performance as a file system.

6.3 Consistency

JuiceFS offers a consistency guarantee known as "close-to-open." When multiple clients are accessing the same file at the same time, any changes made by one client, referred to as A, may not be immediately visible to the others. However, once client A closes the file, all other clients will see the latest changes upon reopening the file, regardless of whether they are on the same host as client A or not.

It's worth noting that "close-to-open" is the minimum consistency guarantee offered by JuiceFS, and in some cases, it may not even be necessary to reopen the file in order to access the latest written data. JuiceFS divides files into data blocks, which have a default size of 4MiB, and assigns unique IDs to each, as illustrated in the figure 6.2. The blocks are then saved on the object storage. Any modification made to the file will generate a new data block, while the original block remains unchanged, including any cached data on the local disk. However, once the file is modified, JuiceFS will read the updated data block from the object storage, rather than from the cache. The overwritten data block will eventually be deleted. This design helps ensure the consistency of the data.

6.4 JuiceFS vs Amazon S3

JuiceFS and Amazon S3 are both popular cloud-based storage solutions, but they differ in several key areas, as follows:

6.4.1 Scalability

JuiceFS is designed to be highly scalable, allowing for easy and efficient processing of large amounts of data. JuiceFS uses a distributed file system that allows for horizontal scalability.
Amazon S3 is also scalable, but it may not offer the same level of performance and scalability as JuiceFS. S3 is designed to provide scalable object storage, and it can store and retrieve any amount of data.

6.4.2 Performance

JuiceFS provides high performance and low latency, making it well-suited for use in real-time data processing and analysis tasks. JuiceFS leverages a distributed file system and optimized data transfer protocols to deliver high performance and low latency.
Amazon S3 also provides good performance, particularly when dealing with large amounts of data. S3 uses a distributed object storage architecture that can handle high levels of concurrency and large amounts of data.

6.4.3 Cost

JuiceFS is designed to be cost-effective, providing a cost-effective alternative to traditional storage solutions. JuiceFS offers flexible pricing options and can be a cost-effective solution for large amounts of data.
Amazon S3 is more expensive than JuiceFS, and it may not be a cost-effective solution for large amounts of data. S3 uses a pay-as-you-go pricing model that charges for storage and data transfer, and it can become expensive for large amounts of data.

6.4.4 Integration with Other Technologies

JuiceFS integrates seamlessly with popular data processing and analysis technologies, such as Apache Spark and Apache Kafka. JuiceFS also provides a REST API for easy integration with other systems and applications.
Amazon S3 also integrates with a wide range of technologies and platforms, including popular data processing and analysis tools. S3 also provides a REST API for easy integration with other systems and applications.

6.4.5 Storage format

In JuiceFS, objects are stored in a distributed file system which is spread across multiple nodes. The system is designed to provide high performance, high scalability and high availability. The system uses a combination of metadata and data management techniques to ensure data consistency, reliability and scalability. This helps to ensure that data is always available and can be accessed quickly even in large-scale deployments. In Amazon S3, objects are stored in a single, flat address space. This means that the system has a large number of objects in a single namespace, rather than having

a hierarchical structure. This design decision was made to ensure that Amazon S3 can scale to accommodate billions of objects. Amazon S3 uses a distributed object store to manage the data, which provides for high scalability, high performance, and high availability.

One of the main differences between JuiceFS and Amazon S3 is that JuiceFS is designed to provide a high-performance file system, while Amazon S3 is designed to provide a scalable and highly available object store. This difference is reflected in the way in which the systems store data and organize the metadata associated with objects. In terms of performance, JuiceFS is designed to be much faster than Amazon S3 when it comes to processing large datasets. This is because JuiceFS provides a high-performance file system that is optimized for handling large amounts of data. JuiceFS also uses a metadata management technique that is optimized for fast processing times, which makes it well suited for big data and analytics applications.

6.4.6 Architecture

JuiceFS is based on a distributed file system architecture, which means that data is stored across multiple nodes in a network. The system uses a combination of data management and metadata management techniques to ensure that data is consistent, reliable, and scalable. The system is designed to provide high performance, high scalability, and high availability. JuiceFS uses a combination of hardware and software optimizations to ensure that data can be stored and processed quickly and efficiently. Amazon S3, on the other hand, is based on a distributed object store architecture. This means that data is stored as objects in a single, flat address space, but ensure the same benefits as JuiceFS. Amazon S3 uses a simple key-value data model, which makes it well suited for use cases that require simple data storage and retrieval. Amazon S3 serves as a bridge between local storage and object storage, allowing users to access cloud storage as if it were stored locally. However, this simple architecture comes with drawbacks. Direct interaction with the object store for file retrieval, reading, and writing may result in slower performance and user experience due to network latency. In contrast, JuiceFS employs a technical architecture that separates data and metadata. Files are first split into smaller data blocks before being uploaded to object storage, with corresponding metadata stored in a separate database. This design enables faster file retrieval and metadata modification as it allows direct interaction with the database, bypassing the network latency associated with object storage. Moreover, when dealing with large files, S3 may struggle with the time and bandwidth consumption associated with rewriting and uploading entire objects. In contrast, JuiceFS splits large files into smaller chunks locally, with data blocks generated for rewriting and appending operations rather than modifying existing blocks, reducing the waste of time and bandwidth resources.
In terms of scalability, both JuiceFS and Amazon S3 are designed to be highly scalable. However, JuiceFS has the advantage of being based on a distributed file system architecture, which provides more flexibility and scalability than the object store architecture used by Amazon S3. This means that JuiceFS can be scaled to accommodate larger datasets and handle more concurrent users than Amazon S3.
In terms of performance, both JuiceFS and Amazon S3 are designed to provide high performance. However, JuiceFS is designed to provide a higher level of performance, especially for big data and analytics use cases. This is because JuiceFS is designed

to provide a high-performance file system, while Amazon S3 is designed to provide a scalable and highly available object store.

6.4.7 Data Processing Workflow

JuiceFS provides a data processing workflow that is tightly integrated with its file system. This means that data can be processed directly on the file system, without the need to transfer it to another system for processing. This allows organizations to take advantage of the high performance and scalability of JuiceFS to process their data efficiently. JuiceFS also provides a number of data processing tools, such as Apache Spark, which can be used to perform complex data processing tasks. Amazon S3, on the other hand, provides a data processing workflow that is separate from its file system. This means that data must be transferred from Amazon S3 to another system for processing. This can be more time-consuming and less efficient than processing data directly on the file system. However, Amazon S3 provides a number of tools and services that make it easier for organizations to process their data.

6.4.8 Caching

JuiceFS offers in-memory caching, which can provide incredibly fast access to data for read operations. This is because data is stored in RAM, which is much faster than disk-based storage. This makes JuiceFS a good choice for applications that require low latency and high throughput. Additionally, JuiceFS provides a global namespace that spans multiple servers, allowing organizations to scale their infrastructure horizontally to accommodate growing amounts of data and increase performance. Amazon S3, on the other hand, does not provide in-memory caching. Instead, Amazon S3 relies on its underlying infrastructure to provide fast access to data. This means that organizations need to have a good understanding of their data access patterns in order to configure the infrastructure to provide fast access to the data they need. Amazon S3 also provides a number of performance optimization options, such as the use of caching and content delivery networks (CDN).
In terms of cost, JuiceFS's in-memory caching can be more expensive than Amazon S3's disk-based storage. However, organizations that need fast access to their data may find that the cost of JuiceFS is outweighed by the performance benefits it provides.

6.4.9 Features

The features comparison is available at the Table 6.1 at page 31.

In conclusion, both JuiceFS and Amazon S3 offer scalable, high-performance storage solutions, but they differ in terms of cost, performance, and integration with other technologies. JuiceFS offers a cost-effective alternative to traditional storage solutions and provides excellent integration with data processing and analysis tools, while Amazon S3 provides a more expensive but widely-used and well-integrated storage solution. It is important to consider the specific needs and requirements of each individual application when deciding between JuiceFS and Amazon S3.

	JuiceFS	**Amazon S3**
Data storage	✓	✓
Type of Data Storage	S3 other object storage WebDAV local disk	S3
Scalability	Scales horizontally	Scales horizontally
Object storage	✓	✓
Storage engine	Object Storage WebDAV	Object Storage
Metadata storage	✓	✓
Customizable Metadata	✓	✓
Support for User-Defined Metadata	✓	✓
Support for Advanced Metadata Queries	✓ (With Spark)	X
Operating system	Linux macOS Windows	Linux macOS Windows
Access interface	REST API, S3 API POSIX, HDFS API	POSIX S3 API
POSIX compatibility	✓	Partially compatible
Shared Mounts	✓	X
Ensuring strong consistency	✓	✓
Local caching	✓	✓
Symbol links	✓	✓
Unix standard permissions	✓	✓
Strong consistency	✓	X
Extended attributes	✓	X
Hard links	✓	X
File grouping	✓	X
Atomic operations	✓	X
Data compression	✓	X
Client-side encryption	✓	X
Language	Go	C++
Open Source License	Apache 2.0 Open Source	GPL V2.0

Table 6.1: JuiceFS vs AmazonS3

6.5 JuiceFS vs HDFS

JuiceFS is fully compatible with Hadoop, Spark, Hive, HBase, Presto or Impala. You can use it as a supplement to HDFS to store cold data (usually takes up most space), or completely replace HDFS with JuiceFS, this separates compute and storage, which fully takes advantage of flexibility & scalability provided by public cloud computing.

HDFS is a widely used big data storage system, which has been deposited and accumulated for more than ten years, and is the most suitable reference frame. JuiceFS is a new player in the big data community, built specifically for big data in the cloud. It is a big data storage solution that conforms to the native features of the cloud. JuiceFS uses cloud-based object storage to store customer data content, and uses the JuiceFS metadata service and Java SDK to achieve full compatibility with HDFS, giving the same experience as HDFS without any changes to the data analysis component. In short, JuiceFS can do things faster, cheaper, and easier on a petabyte scale.

6.5.1 Scalability

As we saw in the section 6.4, JuiceFS is designed to be highly scalable, allowing for easy and efficient processing of large amounts of data. HDFS is also scalable, but it may not offer the same level of performance and scalability as JuiceFS. HDFS is designed to store large files and support parallel processing, but it may not provide the same level of scalability as JuiceFS.

6.5.2 Performance

The Hadoop ecosystem is popular in big data, and HDFS requires long term maintenance especially when data continues to grow, which keeps posing challenges to the community. JuiceFS solves those problems by designing for the cloud, providing users with fully managed service, which can handle tens of billions of files on a single file system, making it the ideal data storage option for big data on public cloud providers.

6.5.3 Cost

As we saw in the section 6.4, JuiceFS is designed to be cost-effective, providing a cost-effective alternative to traditional storage solutions. JuiceFS offers flexible pricing options and can be a cost-effective solution for large amounts of data. HDFS is open-source software, so it is free to use, but it may require a significant investment in hardware and infrastructure to set up and maintain. This may not be a cost-effective solution for some organizations.

6.5.4 Integration with Other Technologies

JuiceFS integrates seamlessly with popular data processing and analysis technologies, such as Apache Spark and Apache Kafka. JuiceFS also provides a REST API for easy integration with other systems and applications, as explained in the section 6.4. HDFS integrates with a wide range of technologies and platforms, including popular data processing and analysis tools. HDFS also provides APIs for easy integration with other systems and applications.

6.5.5 Storage format

In JuiceFS, objects are stored in a distributed file system which is spread across multiple nodes. The system is designed to provide high performance, high scalability, and high availability. The system uses a combination of metadata and data management techniques to ensure data consistency, reliability, and scalability. In HDFS, files are stored in a distributed file system across multiple nodes. HDFS is designed to support large files and provide high availability and scalability, but it may not provide the same level of performance as JuiceFS.

6.5.6 Architecture

JuiceFS is based on a distributed file system architecture, which means that data is stored across multiple nodes in a network. The system uses a combination of data management and metadata management techniques to ensure that data is consistent, reliable, and scalable. The system is designed to provide high performance, high scalability, and high availability. JuiceFS uses a combination of hardware and software optimizations to ensure that data can be stored and processed quickly and efficiently. HDFS, on the other hand, is also based on a distributed file system architecture. This means that data is stored across multiple nodes in a network. HDFS uses hardware and software optimizations to ensure that data can be stored and processed efficiently, but the optimizations may not be as advanced as those used by JuiceFS.

6.5.7 Data Processing Workflow and Caching

In the section 6.4 we saw that JuiceFS provides a data processing workflow that is integrated with its file system, allowing for efficient and fast processing of data. It offers in-memory caching for fast read operations, a global namespace for horizontal scaling, and data processing tools such as Apache Spark. On the other hand, HDFS does not have a tight integration with its data processing workflow. The data must be transferred to another system for processing and does not provide in-memory caching. HDFS provides a reliable and scalable storage system, but organizations may need to invest in additional infrastructure and tools to achieve efficient and fast data processing. Additionally, HDFS may not be the best choice for applications that require low latency and high throughput as it primarily relies on disk-based storage.

6.5.8 Features

The features comparison is available at the Table 6.2 at page 34.

	JuiceFS	**Hadoop HDFS**
Data storage	✓	✓
Type of Data Storage	S3 other object storage WebDAV local disk	Distributed File System
Scalability	Scales horizontally	Scales horizontally
Object storage	✓	X
Storage engine	Object Storage WebDAV	Distributed File System
Metadata storage	✓	✓
Customizable Metadata	✓	X
Support for User-Defined Metadata	✓	X
Support for Advanced Metadata Queries	✓ (With Spark)	X
Operating system	Linux macOS Windows	Linux macOS Windows
Access interface	REST API, S3 API POSIX, HDFS API	HDFS API
POSIX compatibility	✓	X
Shared Mounts	✓	X
Ensuring strong consistency	✓	X
Local caching	✓	X
Symbol links	✓	X
Unix standard permissions	✓	X
Strong consistency	✓	X
Extended attributes	✓	X
Hard links	✓	X
File grouping	✓	X
Atomic operations	✓	X
Data compression	✓	✓
Client-side encryption	✓	X
Language	Go	Java
Open Source License	Apache 2.0 Open Source	Apache 2.0 Open Source

Table 6.2: JuiceFS vs HDFS

6.6 Metadata Performance Comparison
HDFS vs S3 vs JuiceFS

The metadata of a storage system plays a crucial role in determining the efficiency and scalability of the entire data platform, especially when dealing with large files. During the creation, execution, and completion phases of a platform task, there are numerous operations performed on the metadata, such as creation, opening, renaming, and deletion. Thus, the performance of metadata is a crucial factor to consider when selecting a file system.

To evaluate the performance of metadata, three prevalent storage solutions - HDFS, Amazon S3, and JuiceFS - were selected for testing. HDFS has been widely adopted for over a decade, while Amazon S3 is becoming increasingly popular as a big data storage solution on the cloud. JuiceFS, being a new player in the big data world, is designed for cloud environments and built on object storage for big data scenarios.

For more details, this chapter was inspired from a JuiceFS blog article.

The test environment was the following:

```
emr-6.4.0, hadoop3.2.1, HA deployment
master (3): m5.xlarge, 4 vCore, 16 GiB
core (3): m5.xlarge, 4 vCore, 16 GiB
JuiceFS community version: v1.0.0
JuiceFS metadata engine: ElastiCache, 6.2.6, cache.r5.large
```

Listing 6.1: Test Environment

6.6.1 Performance Tests

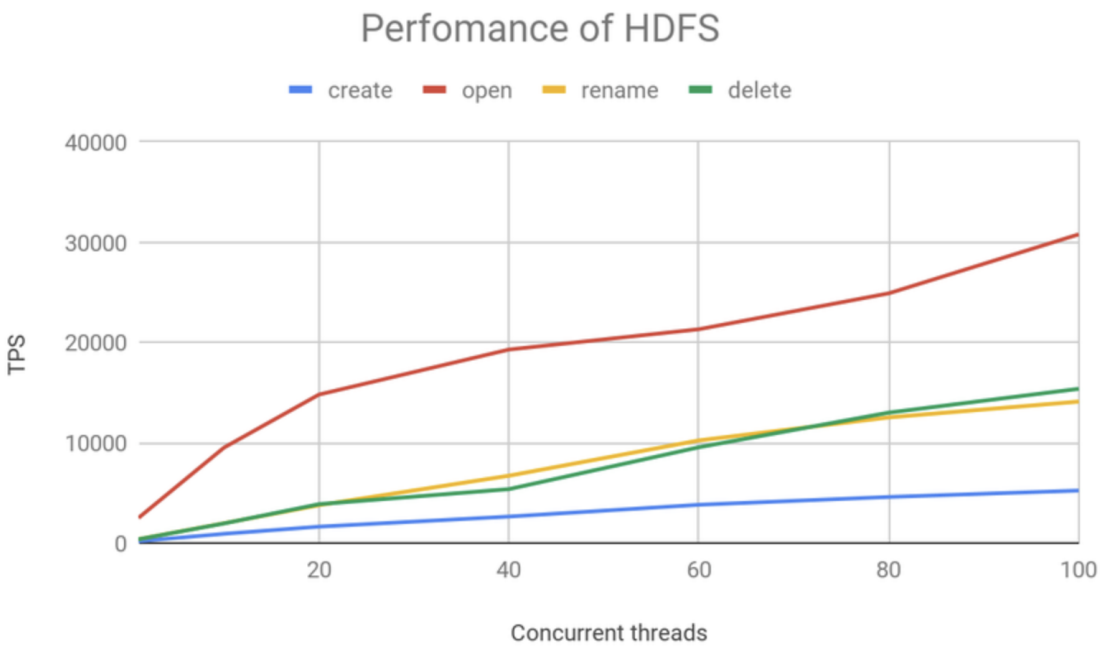

Figure 6.3: Performance of HDFS

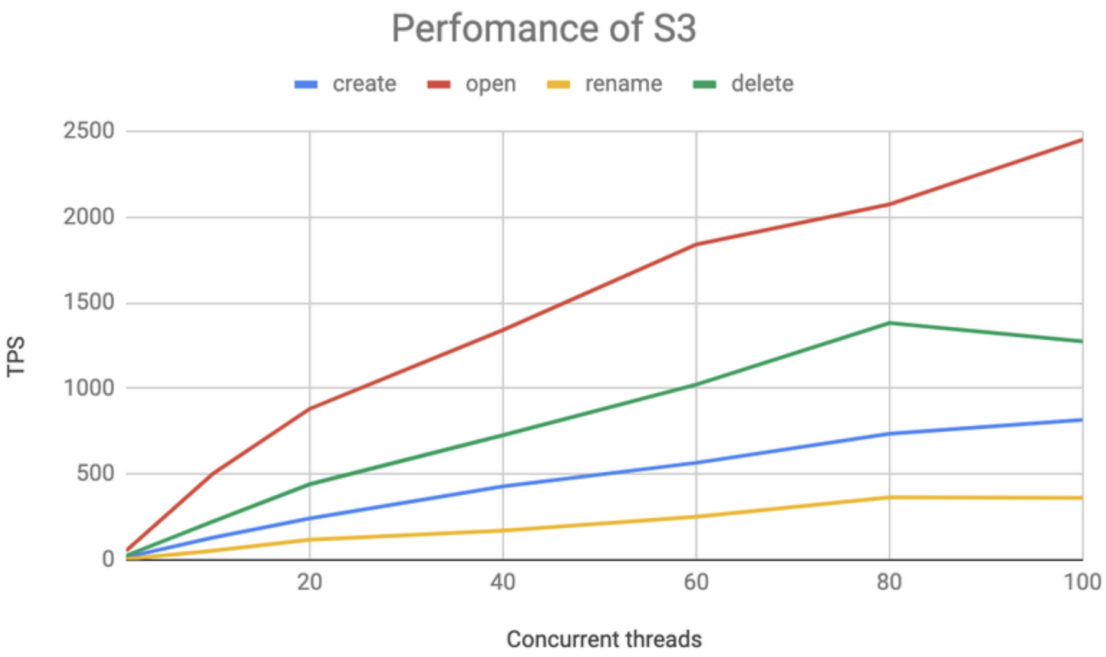

Figure 6.4: Performance of S3

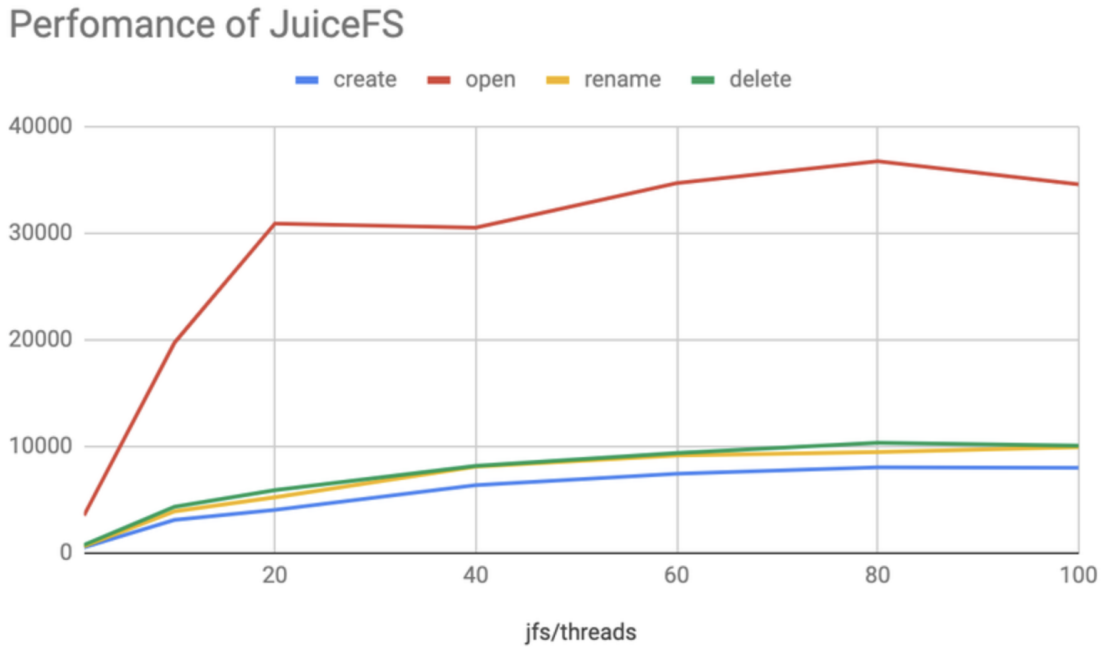

Figure 6.5: Performance of JuiceFS

Observing those graphics we can evaluate the following observations:

- JuiceFS is significantly ahead of S3 in all metadata operations.
- JuiceFS is ahead of HDFS in Create and Open operations.
- In this test, JuiceFS used ElastiCache as the meta engine. The operations reach a performance bottleneck around 80 concurrency which is worse than HDFS.

6.6.2 Conclusion

The evaluation of the performance of HDFS, S3 and JuiceFS was focused on the operation latency (time consumed by a single operation) and throughput (processing power under full load), as follows.

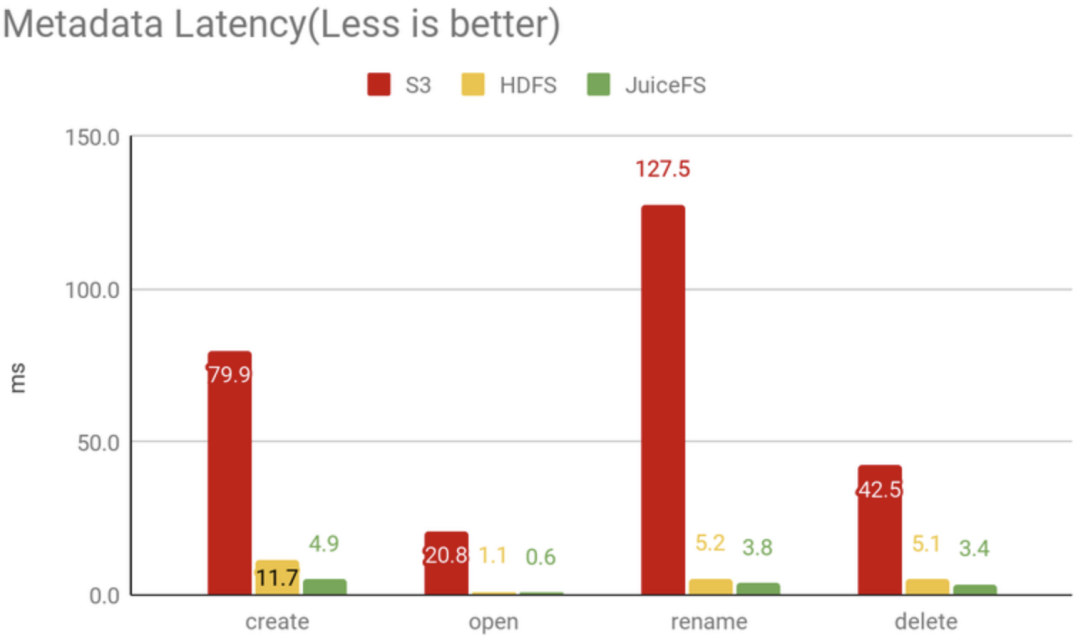

Figure 6.6: Metadata Latency

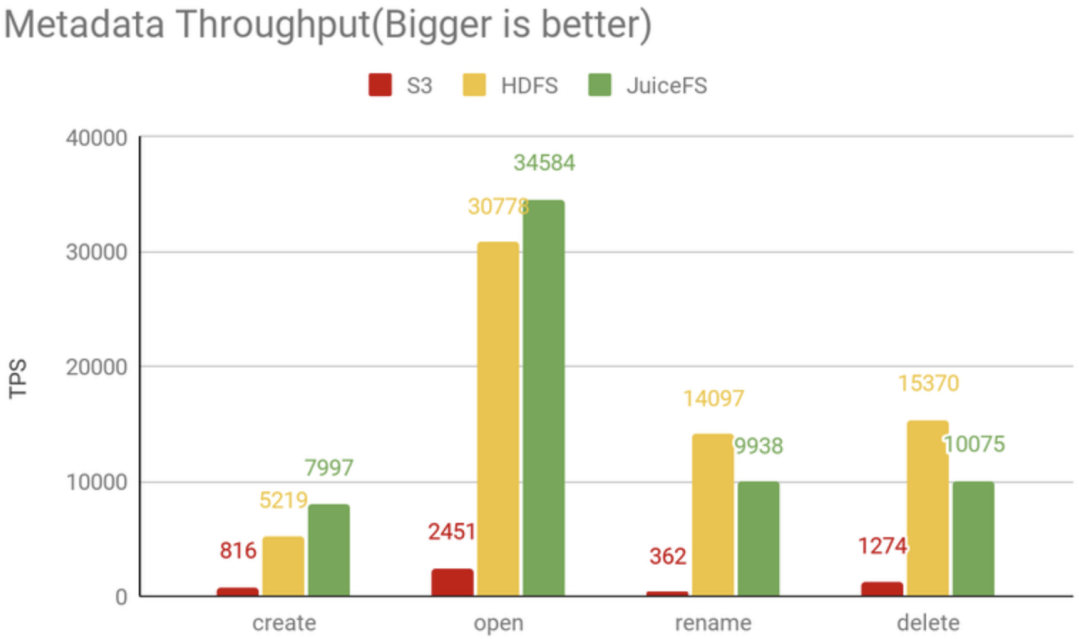

Figure 6.7: Metadata Throughput

Observing the 6.6 Metadata Latency Graphic we can evaluate the following observations:

- The performance of the Rename operation in S3 is notably poor, due to its implementation through the Copy + Delete process. This was demonstrated in our testing, where the Rename was performed on a single, empty file. It's worth noting that in actual big data scenarios, it's much more common to rename an entire directory, which would exacerbate the performance issue even further.
- JuiceFS is faster than HDFS.

Observing the 6.7 Metadata Throughput Graphic we can evaluate the following observations:

- The performance of S3 in terms of throughput is significantly lower compared to the other two products, requiring a higher level of computational resources and increased concurrent processing to achieve similar levels of processing power.
- JuiceFS and HDFS have comparable processing power, with JuiceFS exhibiting better performance for certain specific operations.
- When it comes to increasing concurrency, HDFS can still show improvement in its performance. However, the performance of JuiceFS is constrained by its metadata engine and can hit a bottleneck. To achieve high throughput, it is recommended to use TiKV as the metadata engine.

7. Spark Analysis

Spark Analysis is a data processing framework that provides fast and flexible ways to analyze large amounts of data in real-time. Spark is designed to work in concert with other data storage and retrieval technologies, including Apache Kafka and JuiceFS, and it can be used to perform complex data processing and analysis tasks.

There are several advantages to using Spark for data analysis, including:

- Speed:
  Spark is designed to be fast, and it can quickly process large amounts of data in real-time.
- Scalability:
  Spark is highly scalable, making it well-suited for use in large-scale data processing environments.
- Flexibility:
  Spark is flexible and can be used for a wide range of data analysis tasks, from simple data transformation to complex machine learning algorithms.
- Integration:
  Spark integrates seamlessly with other data storage and retrieval technologies, such as Kafka and JuiceFS, allowing for real-time data processing and analysis.

7.1 JuiceFS and Spark Analysis

JuiceFS can be used as a back-end storage system for Spark, allowing for fast and flexible data analysis and processing. JuiceFS provides high performance and low latency, making it well-suited for use in data analysis and processing tasks. It is also is highly scalable, allowing for easy and efficient processing of large amounts of data. JuiceFS is cost-effective, providing a cost-effective alternative to traditional storage solutions. Spark and JuiceFS can be used together to provide fast and flexible data analysis and processing for large amounts of data.

7.2 Analysis of IoT data

Analyzing massive volumes of IoT data can be a challenging task. Traditional databases are not designed to handle such large amounts of data. Instead, distributed computing frameworks like Apache Spark have become popular for processing large datasets.

Spark provides various APIs to load data from different sources. Since our data is stored in JSON format, we will use the JSON data source API to load the data. Once we have configured the Spark session, we can use the spark.read method to load the data from JuiceFS. Now that we have loaded the data into Spark, we can start querying it using Scala. Spark provides a DataFrame API to perform SQL-like queries on the data. For example, let's say we want to count the number of IoT devices that generated data. We can use the groupBy and count methods of the DataFrame API to do this. Similarly, we can perform various other queries on the data. For example, we can filter the data based on a specific condition, aggregate the data, and join multiple data sources.