Efficient IoT Data Management for Geological Disasters Based on Big Data-Turbocharged Data Lake Architecture

Xiaohui Huang 1, Junqing Fan 1,2, Ze Deng 1,2, Jining Yan 1,2, Jiabao Li 1 and Lizhe Wang 1,2,*

1 School of Computer Science, China University of Geosciences, Wuhan 430074, China; xhuang@cug.edu.cn (X.H.); fanjq@cug.edu.cn (J.F.); cugdz1122@gmail.com (Z.D.); yanjn@cug.edu.cn (J.Y.); wutonglbj@cug.edu.cn (J.L.)
2 Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China
* Correspondence: lizhe.wang@gmail.com

Abstract: Multi-source Internet of Things (IoT) data, archived in institutions’ repositories, are becoming more and more widely open-sourced to make them publicly accessed by scientists, developers, and decision makers via web services to promote researches on geohazards prevention. In this paper, we design and implement a big data-turbocharged system for effective IoT data management following the data lake architecture. We first propose a multi-threading parallel data ingestion method to ingest IoT data from institutions’ data repositories in parallel. Next, we design storage strategies for both ingested IoT data and processed IoT data to store them in a scalable, reliable storage environment. We also build a distributed cache layer to enable fast access to IoT data. Then, we provide users with a unified, SQL-based interactive environment to enable IoT data exploration by leveraging the processing ability of Apache Spark. In addition, we design a standard-based metadata model to describe ingested IoT data and thus support IoT dataset discovery. Finally, we implement a prototype system and conduct experiments on real IoT data repositories to evaluate the efficiency of the proposed system.

Keywords: geohazards; IoT data; data management; data lake; distributed computing

1. Introduction

As a type of disaster, geological hazards (a.k.a., geohazards) refer to irrecoverable and devastating events that cause considerable loss of life, destruction of infrastructure, and negative social-economic impacts [1]. Recently, with the advancement of data acquisition technologies, geohazards data are generated at a staggering rate to help developers and decision makers understand and simulate geohazards, and thus perceive their harmful implications to human beings [2–4]. Specifically, Internet of Things-based monitoring has become a convenient, vital method for geohazards prevention [5,6]. Therefore, organizations and governments have deployed a plethora of devices to realize long-term, fast monitoring of features (e.g., temperature, rainfall) in geohazards bodies. Along with it, a veritable deluge of IoT data, being obtained, managed by the above institutions, are stored in independent data silos that are distributed across many locations.

With the advancement of sensor technologies, we have hundreds of thousands and peta bytes of IoT datasets at the disposal of scientists and developers. These IoT datasets are commonly accumulated in various applications deploying a number of sensor devices. For example, the research project KlimaDigital (https://www.sintef.no/en/latest-news/2018/digitalization-of-geohazards-with-internet-of-things/, accessed on 25 July 2021) mitigates societal risks due to geohazards in a changing climate with the help of data obtained from sensors in an IoT network. These IoT data, inherently a type of big data [7], become data islands due to different management strategies adopted by institutions or organizations. For example, security and privacy are two challenges that are commonly
considered for applications adopting IoT datasets generated by embedded devices [8]. Data islands limit the promotion of IoT data in various applications such as geohazards monitoring [9,10]. To overcome this limitation, many institutions have open-sourced partial IoT data and make them publicly accessible via web services (e.g., HTTP, FTP) to promote the usage of IoT data. Hundreds of thousands of IoT data, represented in different formats (e.g., CSV, JSON, Parquet), are at the disposal of scientists, developers, and decision makers for effective researches on various applications such as geohazards prevention, soil health monitoring [11], etc. Figure 1 depicts an example that obtains only one record of regional meteorological observation dataset open-sourced by Shenzhen Open Data service via sending an HTTP GET request. Note that this repository limits at most 10,000 records returned per HTTP request; a total of 274 HTTP requests, each of which obtains IoT data records and stores them in a JSON file, are required to ingest all IoT data.

```
{
  "total":2738519,
  "data":[
    {
      "P":10154,
      "DDATETIME":"2015-01-09 22:00:00",
      "OBTID":"G1155",
      "T":155,
      "WD10DP":10,
      "HOURR":0,
      "U":52,
      "WD10DD":327,
      "CRTTIME":"2015-01-09 22:29:18"
    }
  ],
  "page":1,
  "rows":1
}
```

Figure 1. Requesting one record of a regional meteorological observation dataset via the Shenzhen Municipal Government Data Open Platform (https://opendata.sz.gov.cn/data/api/toApiDetails/29200_00903515, accessed on 25 July 2021). We request the first record on page 1 via the HTTP GET service with two parameters named page and rows. Ten attributes nested in an attribute named “data” refers to data creation time (termed “DDATETIME”), observation station number (termed “OBTID”), temperature (termed “T”), rainfall (termed “HOURR”), relative humidity (termed “U”), wind speed (termed “WD10DP”), wind direction (termed “WD10DD”), atmospheric pressure (termed “P”), visibility (termed “V”) and record creation time (termed “CRTTIME”).

Managing these IoT big data ingested from open-source repositories suffers the following challenges. The first one is how to quickly ingest such a large volume of IoT data that are distributed over multiple data repositories. Then, how to manage and store enormous IoT data files, with small data sizes (KB, MB) or large data sizes (GB), in a scalable and reliable way, and provide the ability to enable efficient discovery and access to these IoT data. Finally, how to provide researchers with the IoT data exploration ability to satisfy the changing requirements for the data analysis in various geohazards prevention applications with the help of existing big data technologies. Recently, with the prosperous development of distributed computing techniques, leveraging big data platforms that can assist in ingesting IoT datasets from diverse data sources as well as in accelerating the data processing process becomes vital [12]. Both Hadoop and Spark, two milestones in large-scale distributed computing, have been applied in fast data processing to obtain insights from such a huge volume of IoT datasets [13]. In recent years, a number of representative works that couple IoT data management with big data technologies were conducted. DeepHive (https://devicehive.com/, accessed on 22 October 2021), ThingsBoard (https://thingsboard.io/, accessed on 22 October 2021), and SiteWhere (https://sitewhere.io/, accessed on 22 October 2021) are presentative open-source IoT platforms that are primarily designed for the scenario of ingesting real-time IoT data from
devices deployed in IoT environments. The above-mentioned platforms are inconvenient for researchers because they are built with the help of cloud computing technologies such as OpenStack and Kubernetes and thus are not light. Additionally, DeepHive still adopts PostgreSQL, a relational database management system, to manage ingested IoT data. Therefore, DeepHive is limited to its scalability. Apache IoTDB [14] is another representative open-source platform that adopts big data technologies in IoT data management. IoTDB ingests real-time IoT data from devices and stores them on the Hadoop Distributed File System (HDFS) with the TsFile format (similar to Parquet). In addition, IoTDB supports distributed computing with both Hadoop or Spark. However, there is a single point of failure in the Hadoop cluster due to it following the “master–slaves” architecture, and thus it suffers inefficient when managing the deluge of small IoT data files. Paula et al. [15] proposed hut, an architecture based on cloud computing for the ingestion and analytics of historical IoT datasets as well as real-time IoT data to serve smart city use cases. Hut stores IoT data on OpenStack Swift in Parquet format to enable their efficient processing with Apache Spark. All ingested IoT data, historical or real-time, need to be converted to the Parquet file format before they are analyzed by Spark. Therefore, there is extra overhead in converting IoT data from their original data format to the Parquet format. Additionally, Hut is built on top of OpenStack and thus is, as we mentioned above, not light enough for researchers who are not experts in cloud computing. The paper Quoc et al. [16] is another work that monitored and analyzed real-time SpO2 signals in the healthcare industry with IoT techniques. The authors designed a three-tier architecture that provides users with abilities of real-time IoT data ingestion and real-time IoT data processing. This architecture is implemented in a cloud, adopts ThingsBoard for IoT data ingestion as well as data storage and applies Apache Spark to enable fast IoT data processing. However, current IoT data management solutions that combine big data technologies still suffer the following limitations. First, platforms in most of the works are primarily designed for real-time IoT data that are continuously generated by devices in IoT networks. Works rarely focus on the efficient management of historical IoT datasets that are archived in distributed data islands. Second, current solutions require extra data format conversion between the original format of IoT data and the designated format in their platforms when storing IoT data. Historical IoT datasets are huge, and hence this step limits the performance when ingesting historical IoT data from data sources. Additionally, users’ requirements are various, which means there will be data format conversions once the designated format is not the user’s desired format. Third, there are still some parts that can be further improved when leveraging big data technologies in these platforms. For example, distributed caching can further improve the processing ability of Apache Spark when analyzing big IoT data.

To address the above-mentioned challenges, we present an IoT data management system adopting the data lake architecture, to offer researchers the ability to ingest, manage, query, and explore multi-source IoT big data archived in distributed repositories and accessed via web services. A data lake is seen as an evolution of existing data architecture (e.g., data warehouse) [17]. It gathers data from various private or public data islands, holds all ingested data, structured, semi-structured, or unstructured, in their raw data formats, and provides a unified interface for query processing and data exploration, thus enabling on-demand processing to meet the requirements of various applications [18–21]. We first propose a multi-threading parallel data ingestion method by adopting the threading pool technique to realize efficient IoT data ingestion from distributed repositories accessed via web services. Next, we organize the primary ingested JSON files in a multi-level directory structure and store them in a distributed object file system Ceph [22] in a scalable and reliable way. This is done considering that applications access IoT data following an observation attribute-based access pattern, and that in records of single observation attributes of an IoT dataset there exist repeated observation values. We also adopt Apache Parquet [23], a column-oriented, big data-friendly data model, to organize processed IoT data in a data lake. In addition, we use Apache Alluxio [24] to provide a memory-based distributed cache to speed up data access to IoT data stored in Ceph. The reason why we adopt Ceph
instead of HDFS is that there is a single point of failure in the Hadoop cluster due to it following the "master–slaves" architecture, thus suffering inefficiency when managing the deluge of small files. Unlike HDFS, Ceph is a decentralized cluster consisting of multiple server nodes and thus avoids the single point of failure problem. Additionally, data stored in the Ceph cluster is split into a number of small objects (e.g., 4 KB), each of which is distributed across the cluster. Finally, we design a standard-based metadata model for ingested IoT data to enable data discovery. In order to process such a deluge of IoT datasets efficiently, we chose Apache Spark [25], a de-facto standard for distributed computing, as the underlying processing engine as in other IoT data applications [15,16,26]. The reason is twofold. First, Apache spark provides extensive availability for high-performance computing, coupled with the support for a vast variety of data formats (e.g., CSV, JSON, Parquet, etc). In addition, Apache Spark has an abundant ecosystem that provides a number of other big data technologies, e.g., storage, machine learning, etc, and can be utilized in IoT data analysis. Additionally, we chose Apache Spark SQL, an engine that is built on top of Apache Spark [27], to provide researchers with self-defined IoT data processing with SQL-like interfaces [28].

In brief, we highlight our contributions as follows:

- We propose a multithreading parallel data ingestion method, which adopts a fixed-size thread pool, to enable fast data ingestion from IoT data repositories via web services leveraging the CPU level parallelism.
- We propose scalable, reliable, high-performance storage mechanisms for primarily ingested and processed IoT data in distributed environments, and provide fast IoT data access by building a distributed cache layer with Apache Alluxio.
- We design an ISO standard-based metadata model for both ingested and processed IoT datasets to realize IoT dataset discovery. Additionally, we provide a unified SQL-based interface for efficient IoT data exploration by taking advantage of the excellent processing capability of Apache Spark.
- We implement a prototype system, turbocharged by existing big data technologies, for IoT data management following the data lake architecture and adopt real IoT data repositories provided by the Shenzhen Municipal Government Data Open Platform to evaluate the performance of the proposed system.

The rest of this paper is organized as follows. Section 2 details the design and implementation of the proposed IoT data management system. Section 3 provides extensive experimental evaluations of the proposed system. Section 4 presents discussions on experimental results and points out our future works. Finally, Section 5 concludes the paper.

2. System Design

In this Section, we first present an overview of the proposed IoT data management system in Section 2.1. Next, we present the design and implementation of four main components of the system in Sections 2.2–2.5, respectively.

2.1. Overview

The proposed system focuses on historical IoT datasets that are archived in distributed data islands and are accessed via web services. The design goal of the system is to provide researchers with abilities that can ingest, store, process, and analyze big IoT datasets efficiently in a light architecture adopting big data technologies as in other works. The proposed system can be regarded as an extension, which adds an efficient management ability to historical IoT datasets and to existing IoT data management platforms such as Apache IoTDB. The proposed system preserves techniques widely used in current works. For example, the proposed system adopts Apache Spark processing a deluge of IoT datasets and stores the processed IoT datasets in the Parquet format. In addition, the proposed system has the following pros by comparing to platforms mainly designed for real-time IoT data streams in current works. First, the proposed system follows the data lake architecture. All historical IoT datasets are ingested and stored in their primary data formats (e.g.,
JSON, XML) and thus avoid extra overhead caused by data format conversion. Second, the proposed system adopts Ceph as the underlying storage backend instead of HDFS to avoid the single point of failure problem. Third, the proposed system builds a distributed caching layer to further improve the performance when adopting Apache Spark to process IoT data. Figure 2 depicts the architecture and four main components, i.e., data ingestion, data storage, metadata management, and data processing, of the proposed system.

![IoT Data Management System](image)

**Figure 2.** The architecture of the proposed IoT data management system.

The data ingestion component provides the ability to parallelly ingest IoT data from user pre-defined repositories via web services. All ingested IoT data are organized in a multi-level directory tree and stored in Ceph as a wealth of unstructured files. The data storage component also provides users with the ability to store processed IoT data in the Parquet format. A distributed cache layer is built upon the storage layer to enable fast access to IoT data. The data processing layer provides users with an SQL-based interface for IoT data exploration with the help of Apache Spark SQL. The metadata management component creates metadata for both ingested IoT datasets and processed IoT datasets following the designed metadata model. All metadata are organized in JSON format and submitted into the ElasticSearch cluster to provide IoT dataset discovery ability.

2.2. Data Ingestion

The data ingestion component is responsible for ingesting IoT data from distributed repositories via web services, mainly HTTP or FTP, into the proposed system in their native format. However, substantial remote repositories limit the data volume returned by a web service request due to the large volume of archived IoT data. Therefore, users need to submit bulk of web service requests to ingest all IoT data in the repository. How to quickly ingest IoT data from remote repositories is a challenge.

To solve this challenge, our solution is to speed up the data ingestion process by sending web service requests in parallel. The solution, termed a multithreading parallel data ingestion approach, depicted in Figure 3, is twofold: (1) we first provide a high-level abstraction, a class named `DataIngestionTask` that implements the `call()` function defined in the `Callable` interface, for various web service requests. `DataIngestionTask` receives two parameters, a url representing the web service request and IoTDataFileStorePath representing the storage file of the IoT data returned by the request. Additionally, `DataIngestionTask` overrides the `call()` function to execute the ingestion process including sending a web
service request to remote data repositories, obtaining the responded data, storing data in Ceph, generating metadata, and then submitting it to ElasticSearch. Users can extend the `DataIngestionTask` to meet various web service-based data ingestion scenarios. Next, (2) we create a queue to receive objects of `DataIngestionTask` and its derived classes in a fixed-size thread pool, a CPU-level parallelism technique, to parallel execute data ingestion tasks maintained in the queue with multiple threads. In this way, we meet the goal that enables fast IoT data ingestion from remote repositories via web services.

![Image](image_url)

**Figure 3.** Multithreading parallel data ingestion method.

Note that considering that different IoT data sources own their specific ingestion methods, users can write their own data ingestion tasks and only need to extend the `DataIngestionTask` class. That is to say, the proposed method can be adaptive to any IoT data sources that can be accessed via web services. Examples of IoT data sources include but are not limited to The Copernicus Open Access Hub (https://scihub.copernicus.eu/, accessed on 22 October 2021), Earth Explorer (https://earthexplorer.usgs.gov/, accessed on 22 October 2021), and Alaska Satellite Facility (https://asf.alaska.edu/, accessed on 22 October 2021).

### 2.3. Data Storage

The data storage component provides a scalable, reliable environment for all data, mainly the ingested IoT data and the processed IoT data, in the system and provides fast access to these data. For primarily ingested IoT data files, they are organized following a multi-level directory tree structure, and directly stored in defined directories in Ceph. For example, Figure 4 presents the storage mechanism of IoT data that were ingested from the Shenzhen Municipal Government Data Open Platform. These primarily ingested files were stored in their raw JSON format in a three-level directory.

![Image](image_url)

**Figure 4.** Primarily ingested IoT data files storage mechanism.

The following two points need to be considered when storing processed IoT data: (1) a column of an IoT dataset records the observation value of an attribute, and contains
repeated values. Additionally, (2) the access pattern on IoT data is mainly column-oriented. Hence, we chose Apache Parquet, a column-oriented data model, to organize processed IoT data. Conceptually, an IoT dataset is composed of a wealth of records, each of which stores the value of a number of observed attributes. Logically, we first split records into many row groups. Each row group is composed of many column chunks, each of which stores the value of one attribute (a column). A column chunk can be further divided into a wealth of pages. A page is the basic unit in a Parquet file. Note that several encoding methods (e.g., run-length encoding) can be used in pages to save storage space of repeated values and compression techniques, i.e., snappy, gzip, etc., can also be used to further save storage space to enable fast data access. In storage, a Parquet file is stored in a defined directory in Ceph as objects.

To realize fast access to IoT datasets, we built a distributed cache layer upon Ceph by adopting Apache Alluxio. In this way, users can load IoT datasets from the underlying distributed object storage system into the cache layer, thus providing fast access to IoT data. Additionally, for processed IoT data stored in Parquet format, we adopted the partitioning mechanism that splits a Parquet file into a number of Parquet files, each of which only stores data of partitioned records. In this way, when accessing specific partitioned IoT data, only a small fraction of Parquet files need to be accessed instead of the whole Parquet file, thus enabling fast data access. Figure 5 illustrates an example of IoT data stored in Parquet format; the IoT dataset is partitioned according to the OBTID attribute. Each partition is stored as a number of Parquet files because they are generated by a number of workers in the Spark cluster.

Figure 5. IoT data storage mechanism based on Apache Parquet.

2.4. Spark SQL-Based Data Processing

The data processing component is responsible for providing users with an SQL-based interactive environment to enable IoT data exploration. An IoT data processing program with Spark SQL includes three steps:

(1) Loading data and creating a view: The first step is to load the users’ desired IoT dataset from the underlying distributed object storage system with interfaces provided by Apache Spark. The IoT dataset is converted to RDD, a unified abstraction in Spark. Then, users can perform their analytics on the temporal view created based on the primary RDD. The following code illustrates an example that loads the IoT dataset ingested from the Shenzhen Municipal Government Data Open Platform from Ceph via the S3 interface. This dataset is composed of a number of JSON files, each of which contains multiple records, one record per line.

```java
// loading data to generate DataSet object
dataSet = sqlContext.read() .format('json') .option('multiline', 'true') .load("s3a://geolake/landing/ingestionTask1/00903515/*.json")

dataSet.createOrReplaceTempView("datasetTV")
```

2.4. Spark SQL-Based Data Processing

The data processing component is responsible for providing users with an SQL-based interactive environment to enable IoT data exploration. An IoT data processing program with Spark SQL includes three steps:

(1) Loading data and creating a view: The first step is to load the users’ desired IoT dataset from the underlying distributed object storage system with interfaces provided by Apache Spark. The IoT dataset is converted to RDD, a unified abstraction in Spark. Then, users can perform their analytics on the temporal view created based on the primary RDD. The following code illustrates an example that loads the IoT dataset ingested from the Shenzhen Municipal Government Data Open Platform from Ceph via the S3 interface. This dataset is composed of a number of JSON files, each of which contains multiple records, one record per line.

```java
// loading data to generate DataSet object
Dataset<Row> dataSet = sqlContext.read() .format('json') .option('multiline', 'true') .load("s3a://geolake/landing/ingestionTask1/00903515/*.json")

// Create a temporal view for the DataSet
dataSet.createOrReplaceTempView("datasetTV")
```
To speed up the data loading process, users can load IoT datasets from Ceph into the distributed cache in advance. This can be realized with a simple load command provided by Apache Alluxio. The experiments in Section 3.4 present the effectiveness of caching in processing IoT data.

(2) Spark SQL queries-based IoT data processing: Once a temporal view (e.g., datasetTV in the above example) is created for the IoT dataset, users can express their analytic queries on this temporal view with functions provided by Apache Spark SQL. For example, the following codes find all records that generated in the station OBTID (refers to the observation station) from the IoT dataset. The SQL sentence is then submitted to the Spark cluster. The cluster parses the query, generates the plan, schedules how it is executed by workers, and finally delivers query results resultDS to the client.

```java
Dataset resultDS = sqlContext.sql("select * from dsTView where OBTID='G3558'");
```

Users can also develop their own functions with the DataFrame interfaces provided by Spark to facilitate their explorations on IoT datasets.

(3) Data storage: Users can store their processed data in a specific data format and store them in the desired directory in Ceph. Note that it is recommended to store processed IoT data in Parquet format. In default, the run-length encoding method and snappy compression technique are adopted when storing processed IoT data in Parquet format. In addition, users can use the partitioning mechanism to further organize processed IoT data to enable efficient data access. For example, the following codes store an processed IoT dataset as a number of Parquet files, partitioned by the attribute OBTID, into the directory /geolake/productionZone in Ceph with Amazon’s S3 library.

```java
resultDS.write().partitionBy("OBTID").parquet("s3a://geolake/productionZone/00903515.parquet")
```

The data processing component can be further integrated with web-based notebooks, e.g., Jupyter [29], Apache Zeppelin [30], etc., to enable the visualization of the interactive process and IoT data.

2.5. Metadata Management

Metadata management is an important component of the proposed data management system. It extracts metadata from ingested IoT data into a unified metadata model, stores them into ElasticSearch, and exposes IoT dataset discovery ability to users. Without metadata management, a data lake would only lead to a data swamp [31,32].

The core of the system is the metadata model, which provides a unified description model for IoT data. In this paper, we design the metadata model adopting the widely used ISO 19115 standards [33] considering the following two reasons: (1) Designing a metadata model following widely used standards to enhance the interoperability of metadata when sharing archived IoT data. (2) In this paper, we concentrate on managing IoT data that can be used in geohazards. These IoT data are mainly generated by in situ sensors deployed at specific locations (e.g., geohazards bodies), and thus have spatial and temporal characteristics. Therefore, we chose ISO 19115, a famous standard that provides a detailed description of geographic information resources. Figure 6 illustrates our designed metadata model, which is composed of six elements and five associated classes. Note that the LI_Linkage class is responsible for recording the lineage information of the processed IoT data.

Metadata of IoT datasets are extracted by DataIngestionTask during the ingestion phase. Users develop metadata extraction strategies considering the characteristics of IoT datasets in remote repositories, organize them in JSON format, and submit the JSON document to the ElasticSearch cluster. ElasticSearch stores the submitted metadata in a scalable and reliable environment, builds indexes based on the metadata, and supports full-text search to enable IoT dataset discovery [34].
3. Experiments

In this section, we report the experimental results of the proposed management system. We first introduce the experimental settings in Section 3.1. Later, we compare the performance of the proposed multithreading parallel data ingestion method on real data sources in Section 3.2. After that, we demonstrate the efficiency performance of our proposed Parquet-based IoT data storage for geohazards in Section 3.3. Lastly, we test the efficiency performance for our proposed Alluxio-based caching strategy in Section 3.4.

3.1. Settings

(1) Hardware and software: The experimental evaluation was conducted on a seven-node cluster. Each node is a physical machine that has two Intel(R) Xeon(R) eight-core CPUs with version E5-2609, and 16 GB of memory. Each node has three disks whose size is 2 TB and thus provides 6 TB of local storage in total. The operating system running in each node is CentOS 7, version 1511 x86_64.

The infrastructure of the IoT data management system for geohazards is composed of a number of open-source distributed technologies. Table 1 lists details of the software configuration. Note that for each node, two disks are adopted as the Object Storage Devices in the Ceph cluster. The proposed storage system was implemented using Java 8 and Scala 2.11.12.

(2) Data: We evaluated our proposed storage system using five real datasets termed WF10, PM2.5RD, TCOD, ORM, and RMDAS. All datasets listed in Table 2 were provided by the Shenzhen Municipal Government Data Open Platform (https://opendata.sz.gov.cn/, accessed on 25 July 2021).
Table 1. Details of software configuration. Note that ✓ refers to some components of the software installed in this node.

<table>
<thead>
<tr>
<th>Hostname and IP Address</th>
<th>Ceph 14.2.20</th>
<th>Alluxio 2.3.0</th>
<th>Spark 2.4.0</th>
<th>ElasticSearch 7.6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cslave1 10.3.1.11</td>
<td>✓ (monitor)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>cslave2 10.3.1.12</td>
<td>✓ (monitor)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>cslave3 10.3.1.13</td>
<td>✓ (monitor)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>cslave4 10.3.1.14</td>
<td>✓</td>
<td>✓</td>
<td>✓ (monitor)</td>
<td></td>
</tr>
<tr>
<td>cslave5 10.3.1.15</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>cslave6 10.3.1.16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>cmaster 10.3.1.17</td>
<td>✓ (monitor)</td>
<td>✓ (master node)</td>
<td>✓ (master node)</td>
<td></td>
</tr>
</tbody>
</table>

Each dataset records the information of a number of attributes (e.g., wind speed, temperature) observed by monitoring stations deployed in Shenzhen city, and thus is multidimensional. They are updated daily by the platform and can be obtained by sending an HTTP request to the data server. For example, to obtain the data depicted in Figure 1, we need to send the data server an HTTP request (GET or POST) like http://opendata.sz.gov.cn/api/29200_00903515/1/service.xhtml?appKey=xxxxx&page=&rows=1 (accessed on 25 July 2021). Note that three parameters, i.e., appKey, requested page, and the number of requested records (rows) on the desired page, are configured by users. About the five datasets, we refer readers to their website for more details.

Table 2. Details of experimental datasets, which were accessed on 25 July, 2021.

<table>
<thead>
<tr>
<th>Dataset Name</th>
<th>Description</th>
<th>No. of Records</th>
<th>No. of Attributes per Record</th>
<th>Data Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>WF10 1</td>
<td>The weather forecast for the next ten days.</td>
<td>61,881</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>PM2.5RD 2</td>
<td>The PM 2.5 real time observation data.</td>
<td>796,772</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>TCOD 3</td>
<td>The tropical cyclone observation data.</td>
<td>1,387,324</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>ORMS 4</td>
<td>Observation data of regional meteorological observation station.</td>
<td>2,738,519</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>RMDAS 5</td>
<td>Real time meteorological monitoring data of automatic station.</td>
<td>29,069,409</td>
<td>53</td>
<td>5</td>
</tr>
</tbody>
</table>


3.2. Evaluation on Multithreading Parallel Data Ingestion Approach

In this section, we evaluate the proposed multithreading parallel data ingestion method when ingesting data from five real data sources listed in Table 2. All experiments were conducted on cmaster, which has two eight-core CPUs. We repeat each experiment three times and took the average of the results as the final experimental result.
3.2.1. The Impact of the Thread Pool

We compared the proposed multithreading parallel data ingestion method applying a thread pool (denoted by TP) with a default, non-parallel data ingestion method (denoted by WithoutTP), to evaluate how the proposed TP method improves the data ingestion performance. We use the response time after we sent an HTTP GET request to the data server as the metric. We use the speedup ratio as the metric to evaluate the efficiency of the proposed ingestion method. The speedup ratio is calculated based on Equation (1):

\[
\text{Speedup Ratio} = \frac{T_{\text{WithoutTP}} - T_{\text{TP}}}{T_{\text{TP}}}
\]

wherein \( T_{\text{WithoutTP}} \) refers to the time of data ingestion from data servers by sending HTTP GET requests one by one, and \( T_{\text{TP}} \) refers to the time of data ingestion from data servers by sending HTTP GET requests in bulk with a multi-thread pool. Note that the number of threads in the thread pool for the proposed TP method was configured to be 1 in this experiment.

The experimental results are listed in Table 3. It can be observed that, under the condition that both TP and WithoutTP use only one thread, the data ingestion time was significantly alleviated for all datasets except RMDAS when adopting the proposed TP method. The reason is that a data ingestion work includes a large number of HTTP requests, and each request is sent to the data server to obtain desired data as the response. For each HTTP request, the traditional WithoutTP method starts a new thread to send the HTTP request to the data server and shuts down the thread after the data server has responded to the request. Such frequent starting and closing of a thread causes notable time costs. By contrast, the proposed TP method creates a fixed-sized thread pool for a data ingestion task, which includes a number of HTTP requests as we mentioned above. Each thread in the thread pool is assigned to a new HTTP request instead of shutting down once an old HTTP request is accomplished. Once all HTTP requests are handled, the threads in the thread pool are shut down, as well as the release of related resources. The proposed TP method avoids extra overhead in starting and shutting down a thread over and over again, thus expediting the data ingestion time. It can also be observed that with the decrease of data, the ingestion time for RMDAS was not substantial when adopting the TP method. It is interesting and necessary to analyze this phenomenon in depth. The whole data ingestion has three main parts, i.e., sending HTTP requests to the data server, waiting for the response returned by the data server, and analyzing the response result, as we have mentioned in Section 2.2. We have analyzed logs that record information when ingesting the RMDAS dataset from the Shenzhen Municipal Government Data Open Platform. We found that the data ingestion time was significantly alleviated at the beginning of the data ingestion process when adopting the TP method. However, when we continued to send data requests to the data server, the time needed for the data server to return IoT data suddenly became very long. Considering that we do not know how the IoT data are managed in the data server (it is like a black box for us), we guess that the reason could be that the data server limits the processing of requests from the same source to ensure it can provide services for other users.

Table 3. Time of data ingestion from data servers with a bulk of HTTP GET requests in different ways.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Records</th>
<th>( T_{\text{WithoutTP}} ) (ms)</th>
<th>( T_{\text{TP}} ) (ms)</th>
<th>Speedup Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WF10</td>
<td>61,881</td>
<td>7074</td>
<td>5775</td>
<td>22.49</td>
</tr>
<tr>
<td>PM2.5RD</td>
<td>798,772</td>
<td>151,625 (&gt;2 min)</td>
<td>106,906 (&gt;1 min)</td>
<td>41.83</td>
</tr>
<tr>
<td>TCOD</td>
<td>1,387,324</td>
<td>529,038 (&gt;8 min)</td>
<td>422,860 (&gt;7 min)</td>
<td>25.11</td>
</tr>
<tr>
<td>ORMS</td>
<td>2,738,519</td>
<td>1,648,214 (&gt;27 min)</td>
<td>944,633 (&gt;15 min)</td>
<td>74.48</td>
</tr>
<tr>
<td>RMDAS</td>
<td>29,069,409</td>
<td>114,884,844 (&gt;31 h 54 min)</td>
<td>113,344,482 (&gt;31 h 29 min)</td>
<td>1.36</td>
</tr>
</tbody>
</table>
3.2.2. Varying the Number of Threads

To take a closer observation of how the number of threads in the fixed-size thread pool affects the data ingestion time, we further increased the number of threads from 1 to 10. The results are shown in Figure 7, where the x-axis represents the number of threads and the y-axis indicates the time spent on ingesting data from the specific data source. The data ingestion time quickly decreased and showed a downward trend as the number of threads in the thread pool grew. The reason is twofold: the thread pool avoids frequently starting and shutting down a thread as we mentioned above, and a number of data ingestion tasks are executed in parallel instead of one by one.

![Figure 7](image-url). Comparison of time spent on data ingestion from data servers with a bulk of HTTP GET requests with a different number of threads in a multi-thread pool. (a) WF10 containing 61,881 historical records. A total of 62 requests were sent to server. (b) PM2.5RD containing 798,772 historical records. A total of 80 requests were sent to server. (c) TCOD containing 1,387,324 historical records. A total of 139 requests were sent to server. (d) ORMS containing 2,738,519 historical records. A total of 274 requests were sent to server. (e) RMDAS containing 29,069,409 historical records. A total of 2907 requests were sent to server.
Figure 8 shows the effectiveness of the proposed multithreading parallel data ingestion method by calculating the saved time compared to 1 thread according to Equation (2). $T_1$ refers to the time spent on ingesting IoT data with only 1 thread in the thread pool. $T_i$ refers to the time spent on ingesting IoT data with $i$-th ($i = 2, 3, ..., 10$) threads in the thread pool. It can be observed from Figure 8 that when the number of threads in a fixed-size thread pool is continuously increasing from 2 to 10, the saved time for data ingestion shows an upward trend.

\[
ST = \frac{T_1 - T_i}{T_1}, i = 2, 3, ..., 10.
\]  

Figure 8. Saved time comparing to 1 thread for ingesting data with a fix-size thread pool.

3.3. Evaluation on Parquet-Based IoT Data Storage

In this section, we evaluate the efficiency of the proposed Parquet-based data model for the organization and storage of IoT data.

3.3.1. Storage Space Consumption

We first start with the evaluation of the storage space consumption of IoT data ingested from the above-mentioned data sources. Note that for each HTTP GET request to a data source in a data ingestion job, there will be a file storing IoT data in the JSON format once the request is responded to successfully. Therefore, there are a number of IoT data files stored in JSON format for each data source, and the number of data files is determined by the number of HTTP GET requests in a data ingestion job for the data source. In this experiment, all data files were organized into a folder, one data source into one folder, and were archived in the Ceph cluster. We used Spark to read all data files stored in JSON format in a directory, compress data, and generate the files in Parquet format. We used the default snappy and run-length encoding (RLE) as the compression code and encoding method in this experiment, respectively.

The results of time cost and storage space cost on generating Parquet-based data files are given in Table 4. The effectiveness of the storage space was calculated according to Equation (3):

\[
\text{Improvement} = \frac{S_{\text{json}} - S_{\text{parquet}}}{S_{\text{json}}}
\]

where $S_{\text{json}}$ refers to the data size of the total primary json files, $S_{\text{parquet}}$ represents the data size of generated parquet files, and $T_{\text{parquet}}$ is the time spent on generating Parquet-based files with Apache Spark. The experimental results are listed in Table 4. It can be observed that the storage space was significantly reduced when using the Parquet as the format to store massive IoT data, by comparing to storing it with the JSON format. The reason is...
that the IoT data within a column are encoded with the RLE method, which efficiently compresses a large number of successive, repeated values. Therefore, the storage overhead was significantly reduced. Meanwhile, the time cost \( T_{\text{parquet}} \) was acceptable.

### Table 4. Comparison of storage space consumption.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Records</th>
<th>No. of JSON Files</th>
<th>( S_{\text{json}} ) (MB)</th>
<th>( S_{\text{parquet}} ) (MB)</th>
<th>Effectiveness (%)</th>
<th>( T_{\text{parquet}} ) (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WF10 1</td>
<td>61,881</td>
<td>7</td>
<td>20.74</td>
<td>0.97</td>
<td>95.32</td>
<td>1.954</td>
</tr>
<tr>
<td>PMZ.5RD 2</td>
<td>798,772</td>
<td>80</td>
<td>119.01</td>
<td>29.56</td>
<td>75.16</td>
<td>2.770</td>
</tr>
<tr>
<td>TCOD 3</td>
<td>1,387,324</td>
<td>139</td>
<td>491.68</td>
<td>53.46</td>
<td>89.13</td>
<td>8.889</td>
</tr>
<tr>
<td>ORMS 4</td>
<td>2,738,519</td>
<td>274</td>
<td>372.93</td>
<td>70.62</td>
<td>81.06</td>
<td>9.456</td>
</tr>
<tr>
<td>RMDAS 5</td>
<td>29,069,409</td>
<td>2907</td>
<td>12,998.43</td>
<td>1626.09</td>
<td>87.49</td>
<td>95.640</td>
</tr>
</tbody>
</table>


There was a difference for datasets TCOD and ORMS: it can be observed that the data size \( S_{\text{json}} \) of the original IoT data stored in JSON format for ORMS was smaller than it was for TCOD. However, the data size (Sparquet) of IoT data stored in Parquet format for ORMS was larger than for TCOD. The reason is that TCOD contains a substantial portion of NULL values.

### 3.3.2. Evaluation of Efficiency on Different Query Scenarios

Next, we further evaluated the performance of our storage mechanism in the scenario of diverse groups of queries on different datasets. Table 5 defines six queries (Q1–Q6) on three datasets (TCOD, ORMS, and RMDAS) as the benchmark to conduct our experiments. Note that Q1, Q3, and Q5 query the whole dataset to records satisfying one predicate. In addition, Q2, Q4, and Q6 query the whole dataset to find records satisfying two predicates, one of which is used to partition each dataset in Parquet-based data organization. \( R_{\text{total}} \) refers to the number of records in total. \( R_{\text{QH}} \) represents the number of records of query hits. Each query is composed of two stages, i.e., loading data from Ceph to generate a Dataset in Spark and executing query with SparkSQL on the Dataset. Note that there are two types of time for the first stage, i.e., the time spent on transferring data from the disk to a distributed cache (denoted by \( T_{\text{diskToCache}} \)), and the time spent on loading data from the cache to the query engine (denoted by \( T_{\text{cacheToQuery}} \)). Additionally, the time spent on executing the query with Apache Spark is denoted by \( T_{\text{queryExecution}} \). The sum of \( T_{\text{diskToCache}} \) and \( T_{\text{queryExecution}} \) was used to measure the total time for JSON, Parquet and PParquet. The sum of \( T_{\text{cacheToQuery}} \) and \( T_{\text{queryExecution}} \) was applied to measure the total time for CJSON, CParquet, and CPParquet.

### Table 5. Description of query benchmark.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SQL Description</th>
<th>( R_{\text{QH}}/R_{\text{total}} )</th>
<th>Time (s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCOD</td>
<td>( Q1. ) select * from TCOD where WIND = 20</td>
<td>14562/1387324</td>
<td>15.64</td>
<td>ISSUETYPE and WIND refer to the observation site identifier and the forecasting wind speed at the current moment, respectively.</td>
</tr>
<tr>
<td></td>
<td>( Q2. ) select * from TCOD where ISSUETYPE = 'PGTW' and WIND = 20</td>
<td>9/1387324</td>
<td>0.09</td>
<td></td>
</tr>
<tr>
<td>ORMS</td>
<td>( Q3. ) select * from ORMS where T = 150</td>
<td>6615/2738519</td>
<td>0.24</td>
<td>OBTID and T refer to the observation site identifier and the current temperature, respectively.</td>
</tr>
<tr>
<td></td>
<td>( Q4. ) select * from ORMS where OBTID = 'G3558' and T = 150</td>
<td>128/2738519</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>RMDAS</td>
<td>( Q5. ) select * from RMDAS where U &gt; 90</td>
<td>3384510/29069409</td>
<td>0.56</td>
<td>OBTID and U refer to the observation site identifier and relative humidity, respectively.</td>
</tr>
<tr>
<td></td>
<td>( Q6. ) select * from RMDAS where OBTID = 'G3539' and U &gt; 90</td>
<td>46923/29069409</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>
The experimental results are depicted in Figure 9. It can be observed that loading data into Spark dominates the query, by comparing it to the query execution with Spark-SQL. Different storage mechanisms affect the time cost of loading data from Ceph to a Spark Dataset.

For Q1, Q2, and Q3, each needs to scan the whole dataset to find records that satisfy their single query predicates. The time cost spent on loading data stored in the Parquet format (Parquet in Figure 9a–c) is significantly alleviated by comparing it to loading data stored in primarily JSON format (JSON in Figure 9a–c). The reason is that we re-organized IoT data based on Parquet, a column-oriented data format, with the snappy data compression technique and RLE method to reduce the data size of IoT data. Therefore, the data loading time was alleviated.

For Q2, Q4, and Q6, each needs to scan the whole dataset to find records that satisfy two query predicates. Note that a query predicate was adopted to partition the IoT data store in the Parquet format. It can be observed that the time cost spent on loading data stored in the Parquet format (Parquet and PParquet in Figure 9d–f) was significantly alleviated by comparing it to loading data stored in non-partitioned JSON format (JSON in Figure 9d–f). Additionally, the time cost spent on loading data on partitioned Parquet files (PParquet in Figure 9d–f) was reduced compared to loading data stored in non-partitioned Parquet files (Parquet in Figure 9d–f). The reason is that loading data from partitioned Parquet files only needs to load a portion of data, i.e., data chunks storing the partitioned
column consecutively, as a Dataset in Spark. Then, a query is executed on the Dataset with another query predicate. Hence, the query time is alleviated due to the data loading time being decreased.

3.4. Evaluation on Apache Alluxio-Based Caching Strategy

In this section, we evaluate the efficiency of the proposed Alluxio-based caching strategy with the same six queries (Q1–Q6) mentioned above.

The experimental results are shown in Figure 9. It can be observed that the time cost spent on loading data was alleviated after the caching strategy was adopted in all storage mechanisms (CJ SON vs. JSON, CP arquet vs. Parquet, and CP Parquet vs. P Parquet). The reason is that the caching strategy caches IoT datasets in Alluxio, a distributed memory-based orchestration layer. Hence, Spark directly loads data from memory instead of Ceph, and thus decreases the data loading time.

4. Discussion

In this study, we investigated the efficiency of a big data technologies turbocharged data lake architecture in managing distributed IoT big data accessed via web services. It can be observed from experimental results that the proposed multithreading parallel data ingestion method alleviates the time spent on ingesting IoT datasets from distributed data sources. Additionally, the designed IoT data organization and storage mechanism and the adoption of the distributed caching technique reduces the IoT data access time and thus improves the efficiency of IoT data processing when applying Apache Spark. However, this work still has some parts that can be further improved in our future works.

• The current data ingestion method adopts a thread pool, a CPU-level parallelism technique, and sends a bulk of web services (e.g., HTTP, FTP) to the IoT data server. This method has unsatisfactory performance when the data server limits the number of requests from the same machine. It can be observed from the experimental results that the reduction of IoT data ingestion time is flattened even if the number of threads in the thread pool is increased. To overcome this limitation, in our future works, we will consider first distributing data ingestion tasks to each machine in a cluster. Each machine applies the proposed multithreading parallel data ingestion method to further accelerate the data acquisition process.

• The current work lacks an interactive analysis platform for researchers to perform on-demand IoT data processing. In our future works, we will consider combining Jupyter with our proposed data lake-based IoT data management system to provide researchers with the ability to discover IoT datasets archived in either underlying Ceph or a distributed cache, so as to write an IoT data processing code with APIs provided by Apache Spark.

5. Conclusions

This paper focuses on how to help researchers efficiently making use of physically distributed IoT datasets accessible via web services with current big data technologies to meet the requirements of various IoT applications. To do that, we designed an IoT data management system that follows the data lake architecture. This system integrates state-of-the-art computer technologies, i.e., multi-threading technique for fast IoT dataset ingestion, Apache Parquet for efficient IoT data organization, Apache Alluxio for building a distributed cache layer to speed up IoT data access, Apache Spark for high-performance IoT data processing, etc. Experiments on real IoT data sources show the surprising benefits of big data technologies in storing and processing big IoT data. Experimental results indicate that the proposed system performs well on metrics, including the following aspects: the maximum speed-up ratio of data ingestion is 174.5%, and the effectiveness of saving storage space consumption is up to 95.32%. Additionally, the IoT data processing abilities are good according to the results obtained by executing six queries on different IoT datasets.
Our expectation of this work is to guide researchers to quickly build such a system to utilize IoT datasets with current big data technologies efficiently. Our designed system can be applied to various applications that need to analyze IoT datasets. For example, it can be integrated into the early warning system on geohazards to help stakeholders quickly ingest and analyze IoT data to assist in geological hazard evaluations. In the future, we plan to add more data ingestion techniques to ingest IoT datasets not only from distributed IoT data sources accessible via web services but also from IoT data from in situ sensors. Additionally, we also plan to integrate commonly-used notebooks such as Jupyter to provide researchers with an interactive analysis environment.

Author Contributions: Conceptualization, Xiaohui Huang and Lizhe Wang; methodology, Xiaohui Huang and Ze Deng; software, Xiaohui Huang and Jiabao Li; validation, Junqing Fan and Ze Deng; formal analysis, Xiaohui Huang and Lizhe Wang; investigation, Jining Yan; resources, Xiaohui Huang; data curation, Xiaohui Huang; writing—original draft preparation, Xiaohui Huang; writing—review and editing, Lizhe Wang; visualization, Xiaohui Huang; supervision, Lizhe Wang; project administration, Lizhe Wang; funding acquisition, Lizhe Wang. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by the National Natural Science Foundation of China (U1711266 and No. 41925007) and the Strategic Priority Research Program of the Chinese Academy of Sciences, grant number XDA19090128.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Yu, M.; Yang, C.; Li, Y. Big Data in Natural Disaster Management: A Review. Geosciences 2018, 8, 165. [CrossRef]
6. Piccialli, F.; Cuomo, S.; Bessis, N.; Yoshimura, Y. Data Science for the Internet of Things. IEEE Internet Things J. 2020, 7, 4342–4346. [CrossRef]
8. Maritza, G.Z.A.; Esteban, T.C.; Israel, C.V.; León-Salas, W.D. Synchronization of chaotic artificial neurons and its application to secure image transmission under MQTT for IoT protocol. Nonlinear Dyn. 2021, 104, 4581–4600. [CrossRef]