

東海大學

資訊工程研究所

碩士論文

指導教授：楊朝棟博士

運用 Spark 於用電大數據湖泊之資料儲存與分析平台
實作

The Implementation of Data Storage and Analytics
Platform for Big Data Lake of Electric Loads Using Spark

研究生：陳子揚

中華民國一零七年六月

東海大學碩士學位論文考試審定書

東海大學資訊工程學系 研究所

研究生 陳子揚 所提之論文

運用 Spark 於用電大數據湖泊之資料儲存

與分析平台實作

經本委員會審查，符合碩士學位論文標準。

學位考試委員會

召集人

許慶賢 簽章

委員

呂芳婷

賴冠州

陳好言

指導教授

楊朝棟 簽章

中華民國 107 年 6 月 8 日

摘要

隨著物聯網大數據技術的快速發展，資料量產生與累積的速度是相當驚人的，傳統架構的資料儲存與分析技術對現在大資料量的處理已漸漸不堪負荷。以本校為例，過去我們將機房用電與校園用電個別儲存在兩個不同的資料庫系統，長久累積下來的資料量是十分龐大的，如果想要將資料取出並用大數據平台分析的話只能透過 JDBC 的連接或是將資料個別輸出，資料的取出就變得相對繁雜，因此如何將現有系統導入資料湖泊與大數據技術是一個趨勢，也是個挑戰。本篇論文提出一個架構能將現有的儲存系統導入至資料湖泊與大數據平台並儲存與分析電能資料，透過 Sqoop 將舊系統的歷史資料轉存到 Hive 上做資料倉儲，即時的串流資料藉由 Kafka 保持資料的完整性且利用 Spark Streaming 的方式將即時產生的電能資料寫入 HBase 做為即時資料的保存，以 Hive 和 HBase 為基底建置資料湖泊以保持資料的完整性，並整合 Impala 與 Phoenix 個別對 Hive 和 HBase 做為搜尋引擎且。本論文也利用 Spark 提出用電預測與斷電判別等分析模組來分析校園用電情形，分析的結果將會儲存在 HBase 上，本論文所有視覺化的呈現都藉由 Apache Superset 完成。

關鍵字: 巨量資料、資料湖泊、資料儲存、資料視覺化、電能資料

Abstract

With the rapid development of the Internet of Things and Big Data technology, the speed of data generation and accumulation is quite alarming. The data storage and analysis technology of the traditional architecture has become not suitable enough by the processing of large amounts of data. Take our campus as example. In the past, we used the power data from data center and campus to be stored separately in two different database systems. The amount of data accumulated is very large over a long period of time. There is no doubt that Big Data technology brings significant benefits such as efficiency and productivity. However, a successful approach to Big data migration requires efficient architecture. How to import existing systems into Data Lake and Big Data technologies is a trend and a challenge. In this work, we proposed an architecture to import existing power data storage system of our campus into Big data platform with Data Lake. We use Apache sqoop to transfer historical data from existing system to Apache Hive for data storage. Apache Kafka is used for making sure the integrity of streaming data and as the input source for Spark streaming that writing data to Apache HBase. To integrate the data, we use the concept of Data Lake which is based on Hive and HBase. Apache Impala and Apache Phoenix are individually used as search engines for Hive and HBase. This thesis uses Apache Spark to analyze power consumption forecasting, and power failure. The results of the analysis will be stored on HBase. All visualizations of this thesis are presented by Apache Superset.

Keywords: Big Data, Data Lake, Data Storage, Data Visualization, Power Data

致謝詞

能夠完成這篇論文必須感謝很多人，首先謝謝我的指導教授楊朝棟博士，不管是大三的畢業專題，或是大四下開始進入 HPC 實驗室，老師藉由各種不同的訓練以及工作讓我碩士在學期間能夠快速的累積自己的實力與培養良好的工作態度，除了學術研究外也教會許多做人處事的態度以及應對，謝謝老師在大學和研究所的指導，相信這段期間的訓練對我往後的人生都有很大的幫助。

謝謝口試委員許慶賢教授、呂芳懌教授、賴冠州教授、陳牧言教授百忙之中抽空參加我的論文口試，每個教授的寶貴意見都能夠讓我的碩士論文更加完善。也謝謝 HPC 的學長姐、學弟們的指教與幫忙。特別謝謝研究所兩年的好夥伴元廷和靜芳，有你們的陪伴和協助才讓我的研究所生涯更加順利。另外也要謝謝好朋友羿群跟健翔，大學畢業後一起留在研究所繼續當同學，如果研究所兩年沒有你們就不會這麼的精彩了。

最後謝謝我的家人，謝謝爸爸、媽媽從小到大對我的栽培、支持與鼓勵，因為有你們各方面的支持，才能讓我沒有後顧之憂的求學，沒有你們的關心我沒有辦法獨自完成碩士學位，也謝謝女朋友筠惠的陪伴，妳完整了我的碩士求學歷程，由衷感謝一路上幫助我和陪伴我的所有人，謝謝你們。

東海大學資訊工程學系 高效能計算實驗室 陳子揚 一零七年六月

Table of Contents

摘要	i
Abstract	ii
致謝詞	iii
Table of Contents	iv
List of Figures	vi
List of Tables	viii
1 Introduction	1
1.1 Motivation	2
1.2 Thesis Contributions	3
1.3 Thesis Organization	4
2 Background Review and Related Works	5
2.1 Data Lake	5
2.2 Hadoop Ecosystem	6
2.2.1 Apache Kafka	6
2.2.2 Apache Spark	8
2.2.3 Spark Streaming	9
2.2.4 Spark MLlib	10
2.2.5 Apache Superset	10
2.3 Big Data Storage	11
2.3.1 Apache Sqoop	12
2.3.2 HDFS	13
2.3.3 Apache Hive	14
2.3.4 Apache HBase	14
2.4 Query Engine	18
2.4.1 Apache Impala	18
2.4.2 Apache Phoenix	19
2.5 Related Works	20
3 System Design and Implementation	23

3.1	System Architecture	23
3.2	System Services	24
3.2.1	Data Transfer	24
3.2.2	Data Collection	25
3.2.3	Data Storage	27
3.2.4	Data Analysis	30
3.2.5	Data Visualization	33
3.3	System Implementation	35
4	Experimental Results	38
4.1	Experimental Environment	38
4.2	The Speed of Transferring Historical Data	39
4.3	Data Lake's Comparison of Different Search Engines	41
4.4	Streaming Data Storage	41
4.5	Power Failure Analysis Results	44
4.6	Verify Power Forecasting Accuracy with MAPE	45
4.7	Superset Visualization	48
5	Conclusions and Future Work	50
5.1	Concluding Remarks	50
5.2	Future Work	51
	References	52
	Appendix	56
	A Cloudera Manager Installation	56
	B Kafka Producer for ICEMS	57
	C Kafka Producer for IGEMS	60
	D Spark Streaming Write ICEMS to HBase	63
	E Spark Streaming Write IGEMS to HBase	65
	F Power Forecast Using HoltWinters	67
	G Power Failure Analysis	71

List of Figures

2.1	The Architecture of Kafka	8
2.2	Spark Library	9
2.3	The Architecture of Spark Streaming	10
2.4	Spark Streaming Workflow	10
2.5	Superset Dashboard	11
2.6	Sqoop Basic Workflow	12
2.7	The Architecture of HDFS	13
2.8	The Architecture of Hive	15
2.9	The Relationship between Hadoop and Hive	15
2.10	The Architecture of HBase Service	16
2.11	Data Model of HBase	17
2.12	The Architecture of Impala	18
2.13	The Architecture of Phoenix	19
3.1	System Architecture	24
3.2	Sqoop Workflow for Data Transferring	25
3.3	Data Collection Workflow	25
3.4	Kafka Producer for IGEMS	26
3.5	Kafka Producer for ICEMS	26
3.6	Spark Streaming Write Data to HBase for IGEMS	26
3.7	Spark Streaming Write Data to HBase for ICEMS	26
3.8	The Architecture of Data Storage	27
3.9	Phoenix Shell	28
3.10	Impala Shell	29
3.11	Hue GUI Interface	29
3.12	Table Format	30
3.13	The Architecture of Superset	33
3.14	The Structure of SQLAlchemy	33
3.15	Editing Database's SQLAlchemy	34
3.16	Word Cloud Chart	34
3.17	Cloudera Manager Web User Interface	36
3.18	Nodes of Cloudera Cluster	36
3.19	HUE Web User Interface	37
3.20	Superset Dashboard for Power Data	37
4.1	Cloudera Cluster	39

4.2	Comparison of m Number for 7G Table	40
4.3	Comparison of m Number for 17G Table	40
4.4	Execution Time of Searching by Hive and Impala	41
4.5	Execution Time of Ordering by Hive and Impala	42
4.6	Execution Time of Searching by Phoenix, Hive and Impala	42
4.7	Execution Time of Searching by Phoenix and Impala	43
4.8	Execution Time of Counting by Phoenix, Hive and Impala	43
4.9	The Table of Power Failure	44
4.10	Bar Chart for Power Failure	44
4.11	Comparison of actual and predicted values of 0312 week	46
4.12	Comparison of actual and predicted values of 0319 week	47
4.13	Time Series Chart for Power Data	48
4.14	Pi Chart for Power Data	48
4.15	Bar Chart for Power Data	49
4.16	Dashboard fro IGEMS	49

List of Tables

2.1	Difference between Current Data Lake and Data Lake with Spark	22
3.1	Software Specifications	35
4.1	Experimental Environment	39
4.2	MAPE error value level	45
4.3	Daily MAPE value for 0312 week	46
4.4	Daily MAPE value for 0319 week	47

Chapter 1

Introduction

In the current era of the rapid flow of information, there is no doubt that Big Data [1] brings significant benefits such as efficiency and productivity. If companies can analyze data in depth, they can mine enormous potential to make decisions more precise, clear, and quick. In the past, companies were able to face increasing data and application problems by importing database, data warehouse [2], and developing business intelligence [3]. With the development of science and technology, the amount of data has continued to increase, and the types have become more complex.

The traditional architecture will not be suitable enough, and it will also lead to the new construction to deal with the challenge of Big Data. It takes a lot of time for relational database and data warehouse system to check for user's confirmation, data model establishment, data import and verification. And many times in the collecting of data, it's not sure how the data model should be established, and high expansion costs and limitations on vertical architectures is a problem also.

With the rise of the Internet of Things [4] and the increase in the speed of the Internet, more and more sensors are deployed around our lives, and the speed and quantity of streaming data generated are quite tremendous. Therefore, how to properly design a system to store streaming and Big Data is an important issue to think. Importing Big Data technology has become a trend.

The traditional architecture of data storage system is insufficient in the search and analysis of Big Data. We can improve the search speed and keep the integrity of data by importing existing systems into Big Data platform with Data Lake, if the traditional architecture data storage system want to use the Big Data platforms for analysis and storage. We can transfer historical data and write real-time data to Data Lake without affecting the existing system. In this work, we aim to implement an architecture that can import existing power data storage system to Big Data platform with Data Lake and provide Big Data storage, analysis and visualization module for power data. Specific goals are listed as follows:

1. To transfer historical data from existing system to Data Lake in Hive.
2. To collect streaming data into Data Lake in HBase
3. To analyze the data from Data Lake with Spark MLlib
4. To utilize Apache Superset to visualize the analysis results by Spark and the data in Data Lake

1.1 Motivation

The Big Data has 4V characteristics: Volume, Velocity, Variety, and Veracity. The traditional data storage architecture has been unable to deal with the current trend of Big Data. Taking our campus as example, we have deployed a lot of smart meters to collect electricity data in all the campus buildings and data center. In the past, we have separately stored data center and campus buildings electricity data in two different database systems. The amount of data accumulated is enormous over a long period of time. Therefore, if we want to do advanced analysis and application of these power data, in the current mainstream platform for processing Big Data is Hadoop [5] and Spark [6] which support the traditional database only with the JDBC to get data. This makes the time and communication costs of getting data sources very expensive. Spark supports two types of HDFS-based data storage and NoSQL [7] databases, Hive and HBase. Therefore, we want to

integrate the campus electricity system and the data center electricity system into our proposed Data Lake system. After the integration, the extraction and use of data will be more convenient for the analysis of the Big Data platform. After collecting the data, the most important thing is analysis. How to find out useful information in the collected data is the main value of Big Data. If we can find out the trend and distribution of power consumption from power data, even predict the power usage and abnormal warnings, it is very helpful for making decision correctly even early when abnormal power usage happened.

In addition to the above mentioned, how to visualize information is also worthy of discussion and research. Currently the mainstream visualization tools in the market need to be charged. For example, Tableau and Power BI are very powerful which support Hive, Impala, etc and are used by many companies for data visualization. But what if there is a budget pressure for who wants to visualize data in a big data database? There is barely no visualization project for big data database in current Apache Software Foundation open source projects, except for Apache Superset developed by Airbnb and contributed to the Apache Software Foundation. Superset is actually a self-service data analysis tool. Its main goal is to simplify our data exploration and analysis operations and provide data analysts with a fast data visualization function. The Data Lake visualization part we proposed is presented by Superset. The entire Data Lake platform targets the data analysts and developers and it is not for an ordinary user. Providing developers with a unified data source for data analysis and retrieval.

1.2 Thesis Contributions

In this work, we propose a system for power data storage and analysis platform with Spark and Data Lake. This system is an open source platform which processes, analyzes, storing streaming and historical data and visualizes data stored in Data Lake. These are the main original contributions:

1. The design, implementation and test of an entirely open source solution integrating state-of-the-art components from the Apache ecosystem. This architecture deals seamlessly with data transfer, collection, storage, analysis and visualization.
2. Demonstrate how to import an existing storage system to the Big Data platform With Data Lake and collect data sources from multiple sources to Data Lake and use Kafka as the message queue to provide data stream integrity.
3. Power failure analysis and power forecasting modules are proposed in this work can help schools make better decisions.
4. Integrate Impala and Phoenix as Data Lake's search engine and provide better search performance
5. Proposed a complete solution of Data Lake and Big Data platform from the data transfer, collection, storage, analysis, visualization to campus electricity environment.

1.3 Thesis Organization

The paper is organized as follows. Chapter 2 introduces the main background and related works. Chapter 3 gives an overview of the system design and its implementation of Big data tools from the Apache ecosystem that are integrated in our framework and shows every component function in the system. Chapter 4 details the experiments and discussion. Finally, in Chapter 5 we provide a conclusion and the future work for this thesis.

Chapter 2

Background Review and Related Works

In Chapter 2, we provide several components that are approaching in this work: Data Lake, Apache Spark, Apache Hive, Apache HBase, Apache Impala, Apache Kafka, and so on. The next sections discuss each component in more detail.

2.1 Data Lake

The most commonly cited definition of Big Data is "The huge amount of data to the database system can not be stored, computed and processed within a reasonable period of time, and it can be analyzed as information that can be interpreted." This definition is relative and will change with the time, industry, and professional field. The data related to the business activities of many companies or organizations, regardless of their types, speed, and quantity, are rapidly growing. Companies are starting to face a lot of data storage problems. In the past, many companies imported databases, imported data warehouses, and even business intelligence. To face the ever-growing data and application problems of the data.

It's no doubt that Big Data has brought huge benefits such as efficiency and productivity. However, as data continues to increase, the traditional architecture will probably not be sufficient. Enterprises understand that if they can analyze data in deep, they will be able to use their enormous data potential to make decisions faster, clearer, and more elaborate. However, the efficiency of data management and analysis tools must increase dramatically first.

The concept of Data Lake [8] [9] first appeared in an article in the 2011 Forbes magazine "Big Data Requires a Big, New Architecture". Data from the data warehouse is generally of high quality and has been pre-processed, but Data Lake store any types and as a storage pool for all data, it facilitates user's analysis and use. The definition of Data Lake is a super large scale storage space with low cost. For example, Hadoop [10] can store any type of data until user needs to do business analysis or data mining. The data stored in Data Lake is the most original form and has not been processed or managed. There are four main features of Data Lake: saving Big Data with less cost(Low Cost), maintaining high fidelity of data(Fidelity), ease of accessibility(Ease of Accessibility), and flexible data analysis(Flexible)

2.2 Hadoop Ecosystem

2.2.1 Apache Kafka

Kafka [11] [12] [13] is a message queue system that is designed to be fast, scalable, and durable. It is an open-source stream processing platform. Apache Kafka originated at LinkedIn and later became an open-source Apache project in 2011, then a first-class Apache project in 2012. Kafka is written in Scala and Java. It aims at providing a high-throughput, low-latency platform for handling real-time data feeds. In Big Data, an enormous volume of data is used. But how are we going to collect this large volume of data and analyze that data? To overcome this,

we need a message queue system. That is why we need Kafka. The functionalities that it provides are well-suited for our requirements, and thus we use Kafka for:

1. Building real-time streaming data pipelines that can get data between systems and applications.
2. Building real-time streaming applications to react to the stream of data.

Kafka has four core APIs as figure 2.1 shows:

- The Producer API allows an application to publish a stream of records to one or more Kafka topics.
- The Consumer API allows an application to subscribe to one or more topics and process the stream of records produced to them.
- The Streams API allows an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.
- The Connector API allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

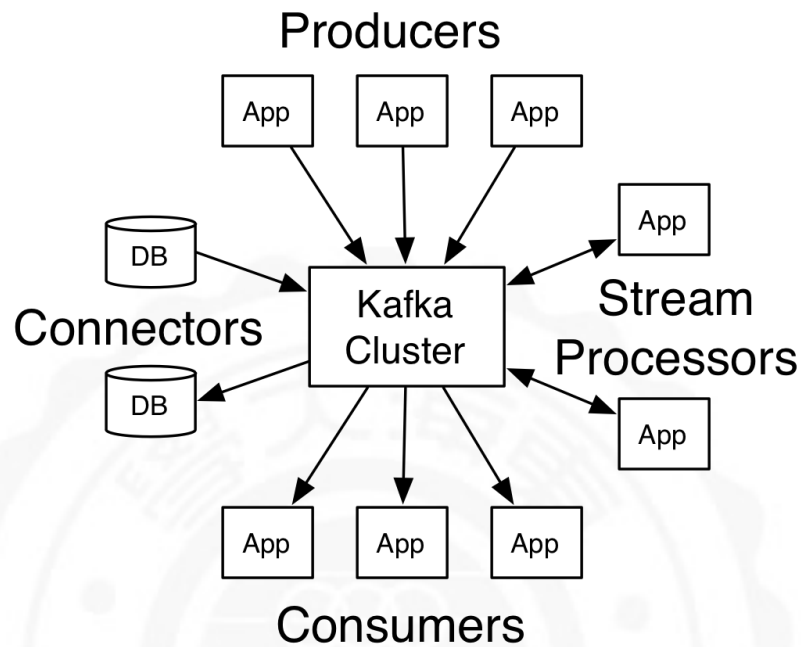


FIGURE 2.1: The Architecture of Kafka

2.2.2 Apache Spark

Apache Spark [6] is an open-source cluster computing framework originally developed in the AMPLab at UC Berkeley. Compared to the two-stage disk-based MapReduce paradigm of Hadoop, Spark's in-memory primitives provide performance up to 100 times faster for certain applications. By permitting user programs to load data into memory of a cluster and repeatedly query it, Spark is well suited for machine learning algorithms. Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster), Hadoop YARN, or Apache Mesos. For distributed storage, Spark can interface with a wide variety of systems, including Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift, and Amazon S3. Spark also supports a pseudo distributed local mode, usually used only for developing or testing purposes, where distributed storage is not required and the local file system can be used instead; in this scenario, Spark is running on a single machine with

one executor per CPU core. In 2014, Spark has more than 465 contributors, making it the most vigorous project in the Apache Software Foundation and Big Data open source projects. Spark provides four main library as the figure 2.2 shown.

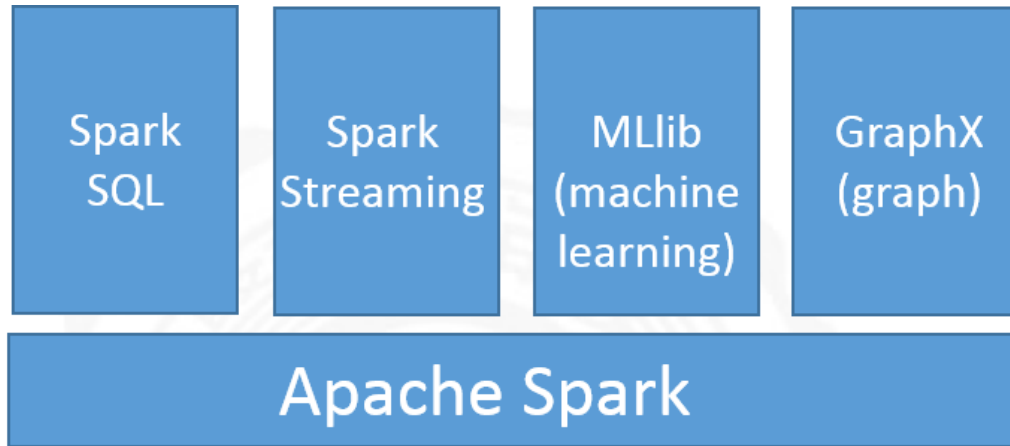


FIGURE 2.2: Spark Library

2.2.3 Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards like figure 2.3 shows. Internally, it works as figure 2.4. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

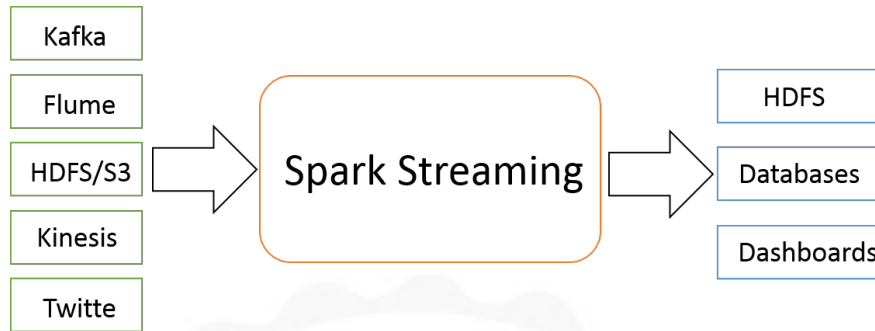


FIGURE 2.3: The Architecture of Spark Streaming

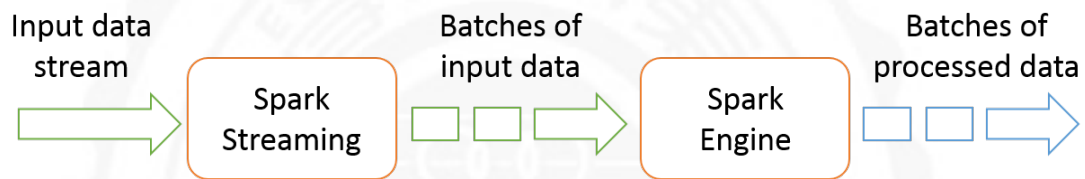


FIGURE 2.4: Spark Streaming Workflow

2.2.4 Spark MLlib

MLlib [14] makes practical machine learning scalable and easy. It consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs. It is divided into 2 packages:

1. mllib contains the original API built on top of RDDs.
2. ml provides higher-level API built on top of dataframes for constructing ML pipelines.

2.2.5 Apache Superset

Superset is an open source data visualization and exploration platform from Airbnb. It was launched in March 2016. The platform provides an intuitive and interactive interface for data visualizations. Superset used to be called Caravel, and Panoramix previously. Superset provides a faster way of visualizing data. It is

highly extensible platform with built in security controls. Apache Superset provides a faster way of creating data visualization and analysis models. Superset is a powerful BI tool that almost matches the power of PowerBI and Tableau. It comes with tons of features for business users. It consists of two primary interfaces:

1. A Rich SQL IDE (Interactive Development Environment) enabling fast and flexible access of data.
2. A Data Exploration Interface that converts data tables into rich visual insights.

The combination of these two interfaces enables users to consume data in a variety of ways. Users can directly visualize data from tables stored a variety of databases including Presto, Hive, Impala[7], Spark[8] SQL, MySQL, Postgres, Oracle, Redshift, and SQL Server. With the addition of a SQL IDE, it provides users with the ability to compose SQL queries to restructure or reduce the size of your data or union data across tables. Additionally, users can immediately visualize their query results using Superset's Visualized flow.

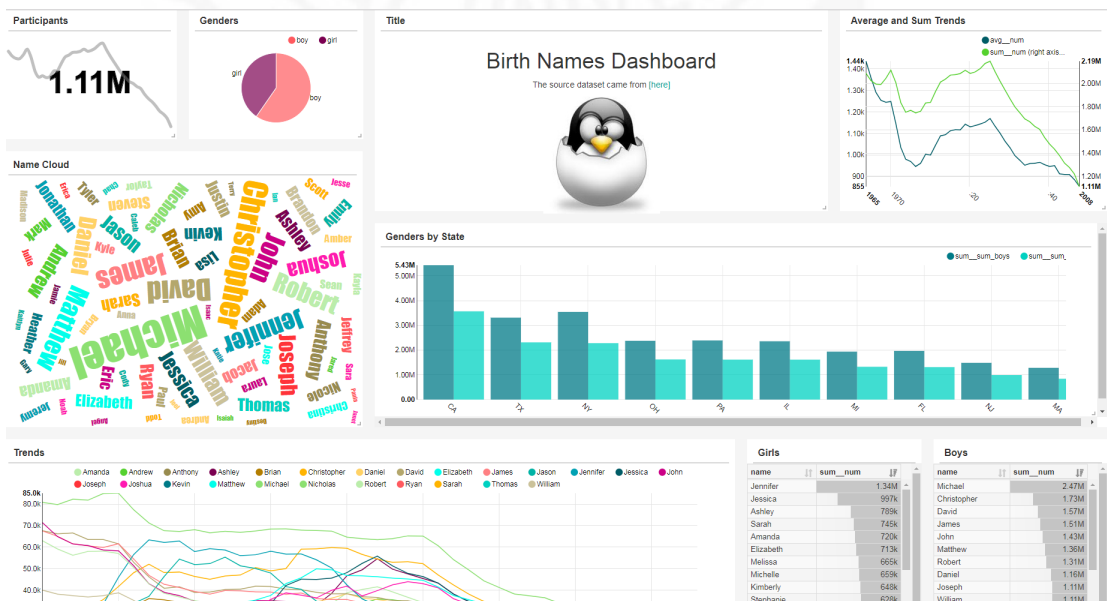


FIGURE 2.5: Superset Dashboard

2.3 Big Data Storage

2.3.1 Apache Sqoop

Apache Sqoop [15] is SQL to Hadoop. Sqoop is a convenient tool that moves data between traditional relational database and NoSQL. Sqoop takes advantage of Hadoop MapReduce parallel feature that accelerates data migration by batch processing.

Sqoop is an import tool that supports data migration from relation database to Hive, HDFS, and Hbase; it also supports full table import and incremental table import. Figure 2.6 shows the basic workflow of Sqoop. When Sqoop imports table data from RDB, it depends on different split-by values to split data; next it lets segmented blocks assigned in different map, and each map will process its block data. Finally, it stores data in the Hadoop distributed storage system.

- Sqoop feature:
 1. High efficiency resources control; parallel processing task to save program execution time.
 2. Data type mapping and transforming can be automatically; users can also define their own data.
 3. Supporting multiple relational databases, such as MySQL, Oracle, SQL Server, DB2.

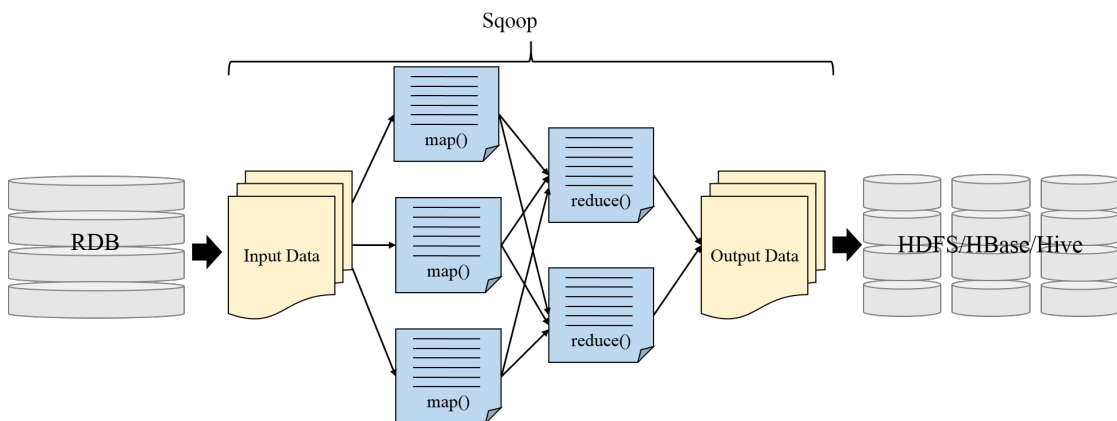


FIGURE 2.6: Sqoop Basic Workflow

2.3.2 HDFS

HDFS [16] is a distributed file system designed to run on commodity hardware. The detection of faults and automated recovery is an important architectural goal of HDFS. HDFS has master-slave architecture with a single Name Node as the master server to manage the file system [1]. Besides, a number of DataNodes, usually one per node in the cluster, manage storage attached to the nodes. HDFS describes a file system namespace and allows user data stored in files. Internally, a file is split into one or more blocks that are stored in a set of Data Nodes. The Name Node executes file system namespace operations such as to open, close, and rename files and directories, and it controls the mapping of blocks to Data Nodes as well. The Data Nodes are responsible for responding read and write requests from clients of the file system. HDFS ensures input distribution and provides the user with an interface whose role is to provide chunks of data files to cluster nodes. Among its chief advantages, HDFS provides input locality by enabling nodes hosting input shards to apply their processing on such chunks, rather than on remotely stored data. Figure 2.7 shows the architecture of HDFS

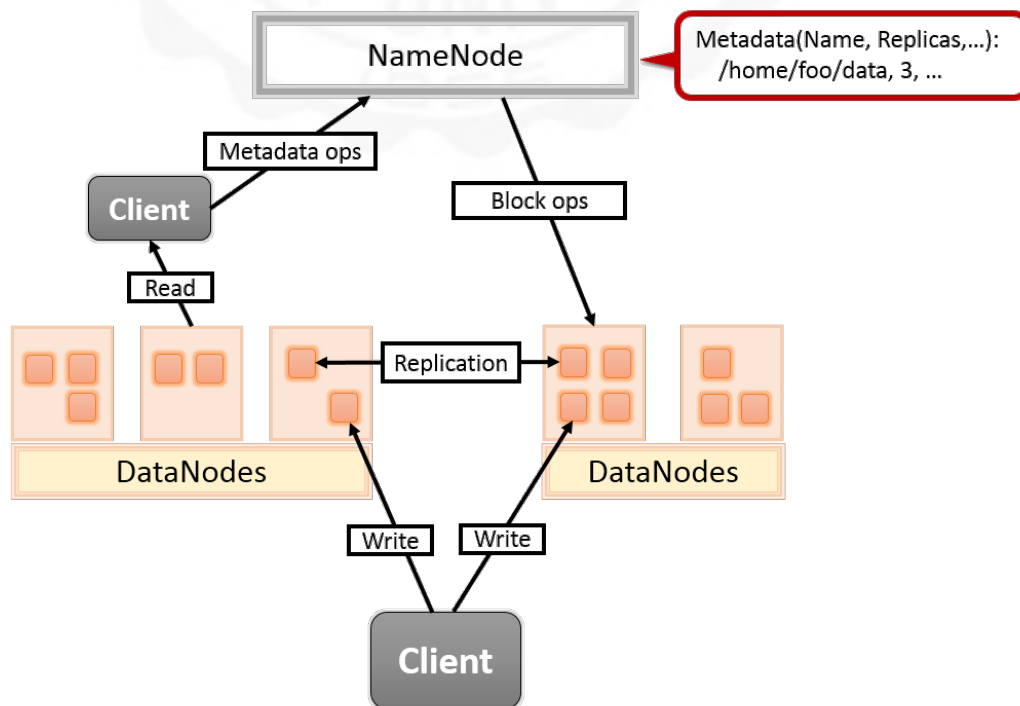


FIGURE 2.7: The Architecture of HDFS

2.3.3 Apache Hive

Apache Hive [17] is a data warehouse solution that has been developed by Apache software Foundation to integrate data storage and to query and manage large datasets. Hive as a data warehouse application on top of Hadoop MapReduce, allows the users to handle the data stored in it as if it was stored in a regular databases. Hive provides a mechanism to project structure onto this data, and query the data using a SQL-like language called HiveQL. Hive enables user with experiences of using traditional RDBMS to do familiar queries on MapReduce. The advantages of Hive are as follows:

- Very powerful.
- Easy to learn and understand.
- Portable and multiple data views.
- Used with and DBMS system with vendor.
- Well defined standards exist and used relational databases.
- High speed, integrating with Java.

Figure 2.8 shows the architecture of Hive, and Figure 2.9 shows the relationship between Hadoop and Hive.

2.3.4 Apache HBase

Apache HBase [18] [19] is a project undertaken by Powerset to deal with the huge amount of data generated by natural language searching. But now it is already a top-level project of the Apache Foundation. HBase runs on HDFS and has attracted widespread attention. Facebook chose HBase to implement its messaging platform in November 2010. HBase is distributed database on HDFS architecture, and is different from general relational database. It is modelled with

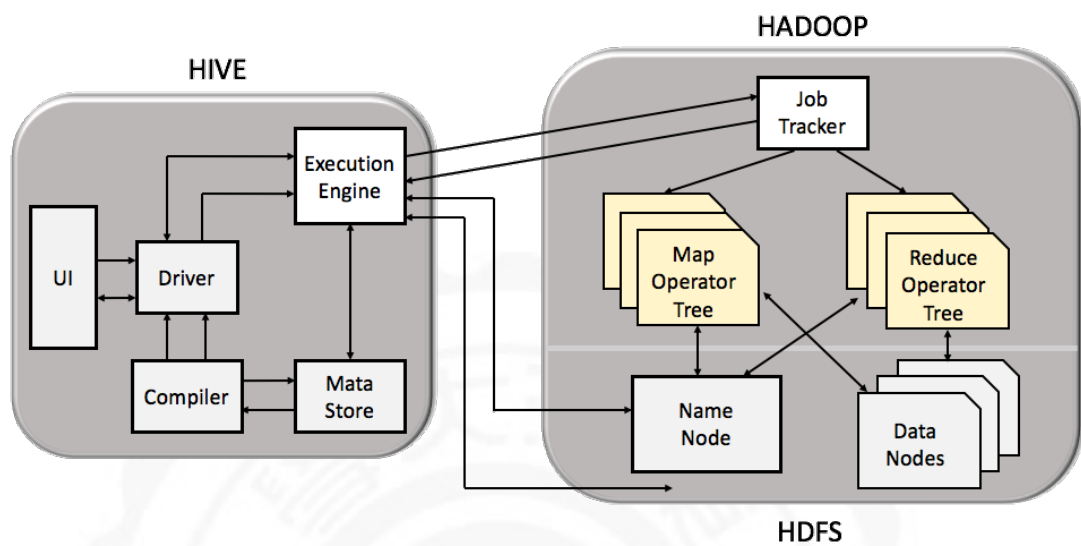


FIGURE 2.8: The Architecture of Hive

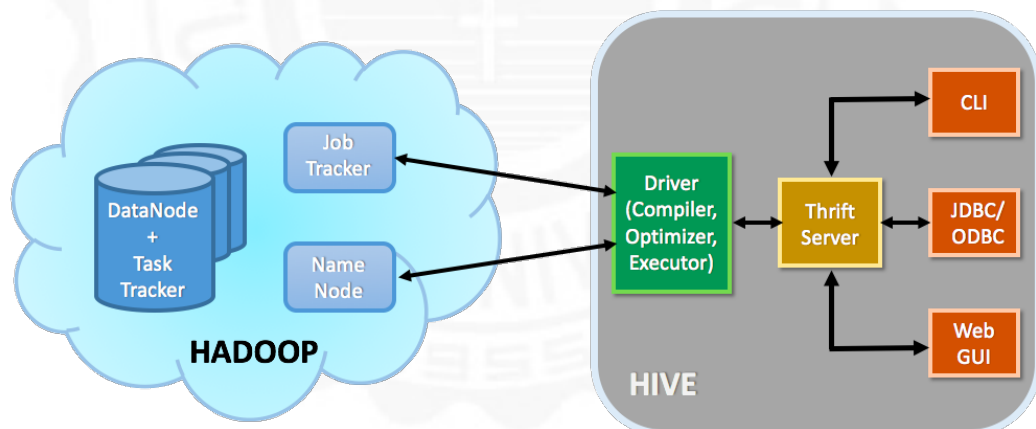


FIGURE 2.9: The Relationship between Hadoop and Hive

reference of Google's BigTable, programmed in Java, and fault-tolerant for storing massive sparse data. The tables from HBase can be used as inputs and outputs in MapReduce tasks. It can be accessed through the Java API, and it also can be accessed by REST, Avro or the Thrift API. Today, it has been used in a number of data-driven sites, including Facebook's messaging platform. In order to conveniently separate data and operation work, the entire data table is divided into many regions. One region is composed of one or more columns, which can be stored in different hosts called as the region servers; master server is used to record a region corresponding to each region server; besides, there is the master server to

record every region server corresponding to every region. The master server will automatically reassign regions on the region server that cannot provide services to another region server. The HBase service architecture is shown in Figure 2.10.

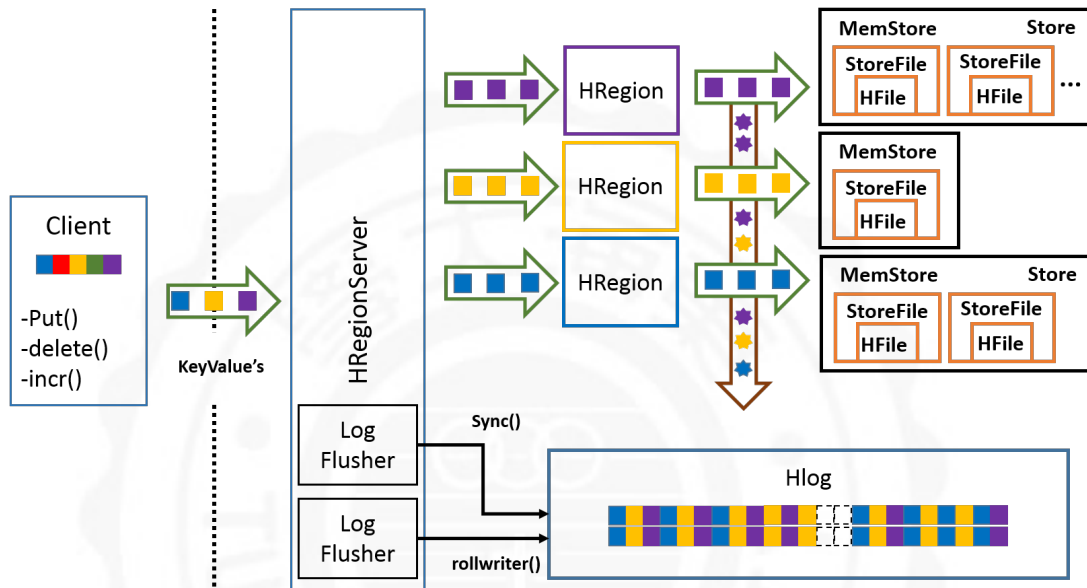


FIGURE 2.10: The Architecture of HBase Service

- **Data Model**

HBase can provide MapReduce programs with data sources or storage space. After HBase version 0.20, it provides TableMapper and TableReducer categories to allow inheritance of the Mapper and Reducer classes. And thus, the key and value in MapReduce can be more easily removed and stored in HBase. HBase uses the row and column as index to access data. It is more like using map container when querying. Another feature of HBase is that each piece of data has a timestamp, so that in a same field there are multiple sets of data of different time. An HBase data table is composed of a number of row and column families; each column has a row key as index. A column family is a set of column labels, which may have many groups of labels. These labels can be added as needed any time without having to reset the entire data table. When accessing data in data table, one usually uses a combination of ('row key', 'family: label') or ('row key', 'family: label', 'timestamp', 'value') to retrieve the required fields. Next, we will introduce the Data Model in HBase, which is shown in Figure 2.11.

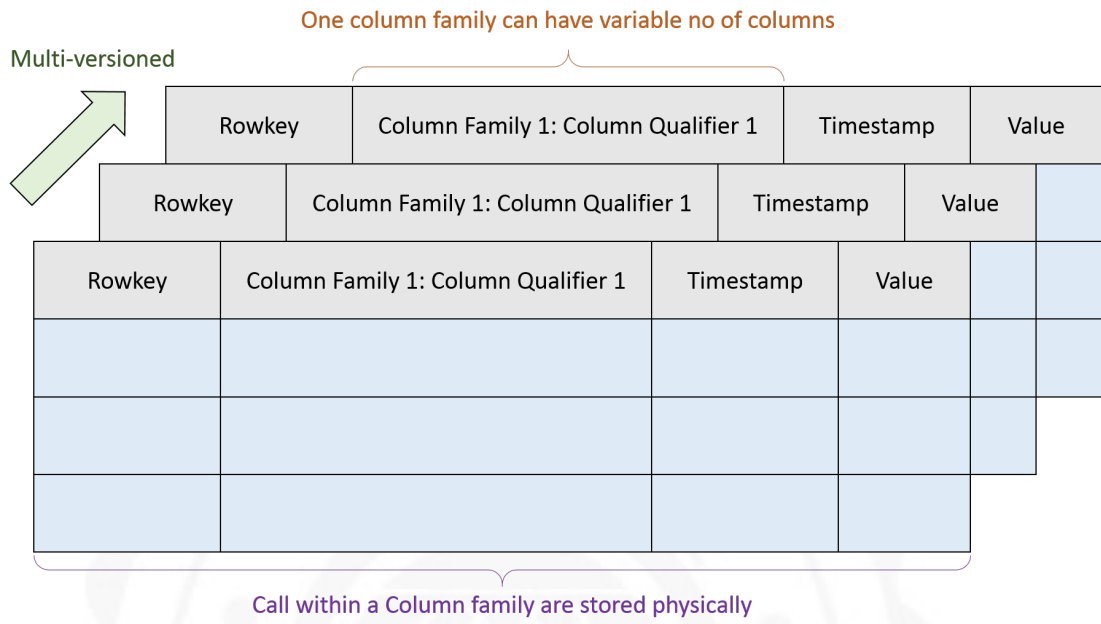


FIGURE 2.11: Data Model of HBase

2.4 Query Engine

2.4.1 Apache Impala

Apache Impala [20] is a real-time SQL query engine that brings scalable parallel database technology for the Hadoop ecosystem. It allows user use SQL to query petabytes of data stored in HDFS and HBase without data movement or transformation. Impala uses Hive metastore, and it can be used to querying data from Hive tables directly. Unlike Hive, Impala SQL does not translate the queries into MapReduce jobs but executes them natively. However, Impala is memory intensive and does not run effectively for heavy data operations like joins because it is not possible to push in everything into the memory. The architecture of Impala is shown in figure 2.12

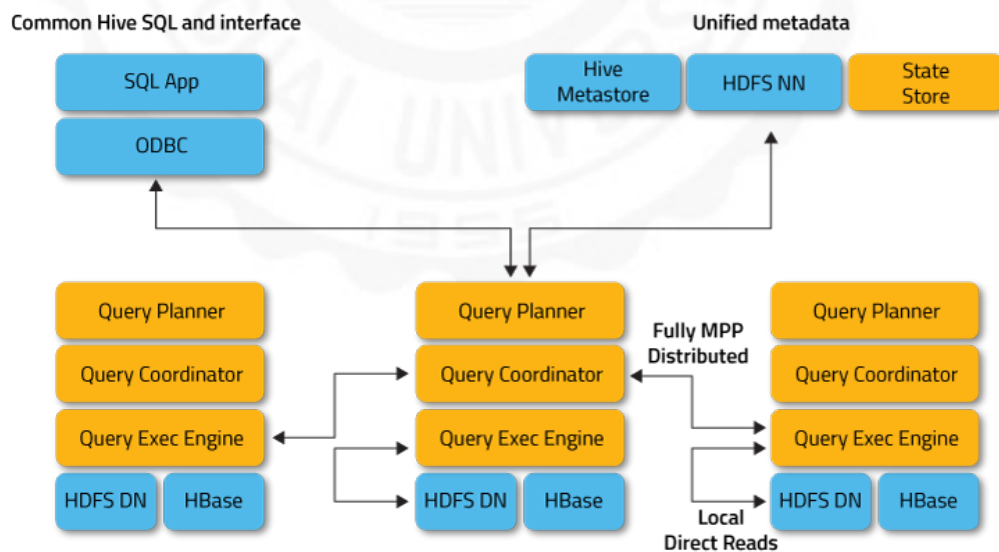


FIGURE 2.12: The Architecture of Impala

2.4.2 Apache Phoenix

Apache Phoenix [21] is an efficient SQL interface for Apache HBase. Many companies are successfully using this technology, including Salesforce.com, where Phoenix first started. Phoenix adds SQL to HBase, the distributed, scalable, Big Data store built on Hadoop. Phoenix aims to ease HBase access by supporting SQL syntax and allowing inputs and outputs using standard JDBC APIs instead of HBase's Java client APIs. It lets you perform all CRUD and DDL operations such as creating tables, inserting data, and querying data. SQL and JDBC reduce the amount of code that users need to write, allow for performance optimizations that are transparent to the user, and opens the door to leverage and integrate lots of existing tooling.

Internally, Phoenix as shown in figure 2.13 takes your SQL query, compiles it into a series of native HBase API calls, and pushes as much work as possible onto the cluster for parallel execution. It automatically creates a metadata repository that provides typed access to data stored in HBase tables. Phoenix's direct use of the HBase API, along with coprocessors and custom filters, results in performance on the order of milliseconds for small queries, or seconds for tens of millions of rows.

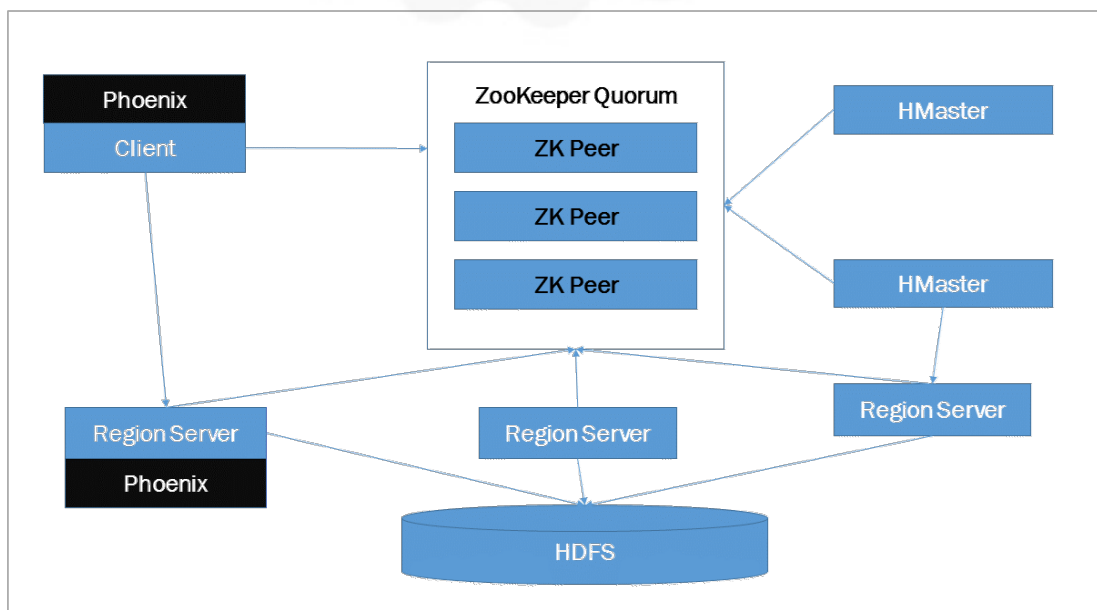


FIGURE 2.13: The Architecture of Phoenix

2.5 Related Works

Sarathkumar Rangarajan et al. [22] proposed an architecture for Personalized Healthcare Service Recommendation using Big Data Lake which just simply uses HDFS as the base of Data Lake. Their data lake architecture does not use any search engine to provide data queries. It is simply using HDFS as a data store. How to manage the data stored on HDFS not explained. Maanak Gupta et al, [23] are about Data Lake, but also a simple description of HDFS as a storage space, and there is no further research on how to manage data. The main focus of this article is on the control of data access rights, which is also very important for this thesis in the future. How to consider information security in the future is also one of the directions of this thesis.

The architecture we proposed is close to Pradeeban Kathiravelu and Ashish Sharma et al. [24] proposed but the difference is that they use Data Café Server to catch real-world biomedical data repository with various data collections and multiple relationships across them to Hive and select Apache Drill as the search engine. In this work we use kafka as the message queue to catch the data generated from the smart meters we deployed in campus and kafka will form the data stream as the input source for spark streaming.

Solaimani et al. [13] presented a novel, generic real-time distributed anomaly detection framework for multi-source stream data. They investigated anomaly detection for a multi-source VMware-based cloud data center, which maintains a large number of virtual machines (VMs). This framework continuously monitors VMware performance stream data related to CPU statistics. It collects data simultaneously from all of the VMs connected to the network and notifies the resource manager to reschedule its CPU resources dynamically when it identifies any abnormal behavior from its collected data. Effective anomaly detection in this case demands a distributed framework with high-throughput and low latency. Distributed streaming frameworks like Apache Storm, Apache Spark. Kafka is well compatible with Spark. It provides guaranteed message delivery with proper

ordering. This means that messages sent by a producer to a particular topic partition will be delivered in the order they are sent, and a consumer also sees messages in the order they are stored. Moreover, a Kafka cluster can be formed to lower processing latency and provide fault tolerance without loss of data. Experimental results show that the use of spark streaming can effectively detect abnormal conditions. Therefore, this thesis refers to this paper also uses the framework of Kafka and Spark Streaming.

Zhang et al. [25] 2013 used HBase to store big data since HBase provides the distributed data storage cluster through HDFS in Hadoop, It is good for big data storage and processing. Their experiments indicate a good performance since their system use HBase as a big data database. Yue Wang et al. [26] 2015. The Apache Hive has been widely used for big data analysis. By providing the SQLlike query language HiveQL, it lower the threshold of big data query. Hive query data efficiently since HiveQL convert into MapReduce tasks. Anja Gruenheid et al. [27] 2011. Hive is a data warehousing solution on top of the Hadoop framework that store data in the Hadoop distributed file system (HDFS). Through using MapReduce assisted SQL query, it more efficient than traditional SQL query. Accordingly, these thesis proposed using HBase or Hive to solve the big data storage.

Chao-Tung Yang et al. [28] they builds a cloud storage system with HBase of Hadoop for storing and analyzing big data of medical records and improves the performance of importing data into database. The data of medical records are stored in HBase database platform for big data analysis. In [29] Ren-Hao Liu et al, proposed a system to collect the electricity usage data in campus buildings through smart meters and environmental sensors, and process the huge amount of data by big data processing techniques. According to the experimental results shows [28] [19], this thesis selects HBase as the storage of streaming data and also this thesis is an extension of the [29]

Their proposed architecture does not support a solution for importing existing systems into Data Lake. This thesis differs from the above papers in that we

propose a complete solution for importing the existing system into the Data Lake and uses the two different natures of the Hive and HBase storage schemes to form a hybrid rather than a single Data Lake. The above-mentioned Data Lake architecture does not propose a data visualization solution. Our architecture can visualize Data Lake's data.

The biggest difference between the proposed architecture and the current solution is to provide search engine, historical data, streaming data import and visualization.

TABLE 2.1: Difference between Current Data Lake and Data Lake with Spark

	Current Data Lake	Data Lake with Spark
HDFS	Y	Y
Historical Data Import	N	Y
Streaming Data Import	N	Y
Search Engine	N	Y
Visualization	N	Y

Chapter 3

System Design and Implementation

This section introduces the system design architecture and implementation of the proposed power data storage and analysis platform with Spark and Data Lake. The system, based on cloud architecture for Big Data, first collects the streaming data generated from smart meters with Apache Kafka and transfers the historical from existing storage system and then processes and analyzes these data to efficiently obtain real-time power information and perform abnormality forecast. Moreover, the proposed system supports historical data queries and behavior analysis.

3.1 System Architecture

This thesis presents an architecture for data collecting, data storage, analysis [13], and data visualization. In the part of data collecting, the data is divided into historical data from existing system and streaming data, and all data stores in Data Lake which is based on Hive and HBase [19]. It also use Apache Phoenix [21] and Apache Impala as search engine to allow user can search and use data quickly. Data Lake stores all the original data. We use Spark MLlib [30] to analyze the

data and store the result to Data Lake and the results are visualized by Apache Superset.

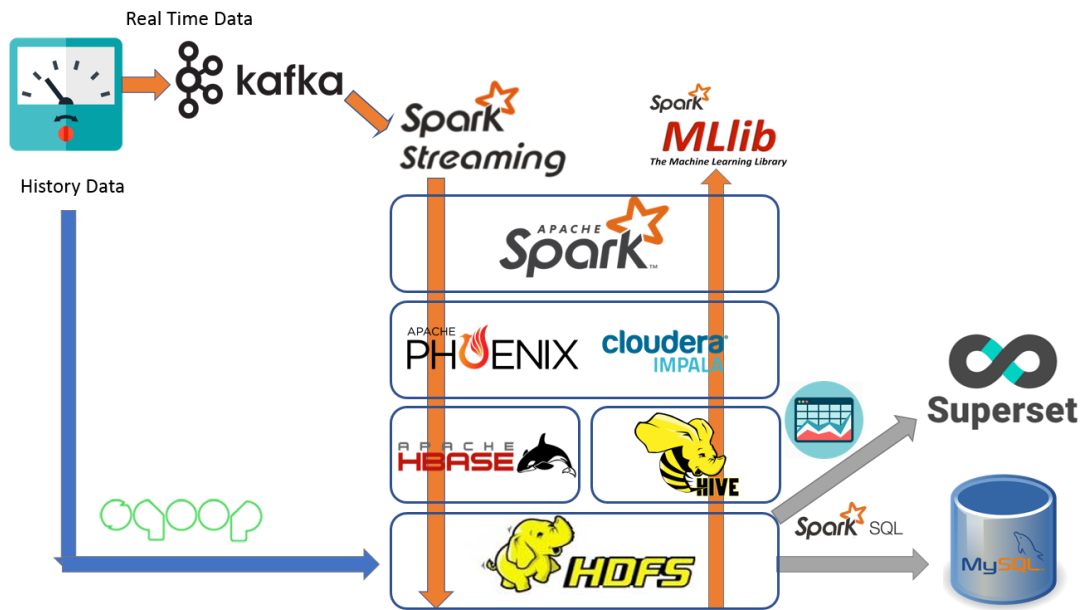


FIGURE 3.1: System Architecture

3.2 System Services

This section introduces the main service provided by the Data Lake system we proposed. Including data transfer, data collection, data storage, data analysis and data visualization.

3.2.1 Data Transfer

In order to transfer the data from the existing storage system to Hive, we use Apache Sqoop [15] to move historical data to the architecture we propose and use the characteristics of parallelism to speed up the overall movement of data. Sqoop is a data transfer tool that can transfer data from a traditional relational database to a Hadoop storage system by using Hadoop MapReduce parallelism to speed up the process of data migration. It supports not only transfer data from MySQL

to HDFS but also Hive and HBase. Figure shows the workflow for transferring historical data from relational data base to Hive table.

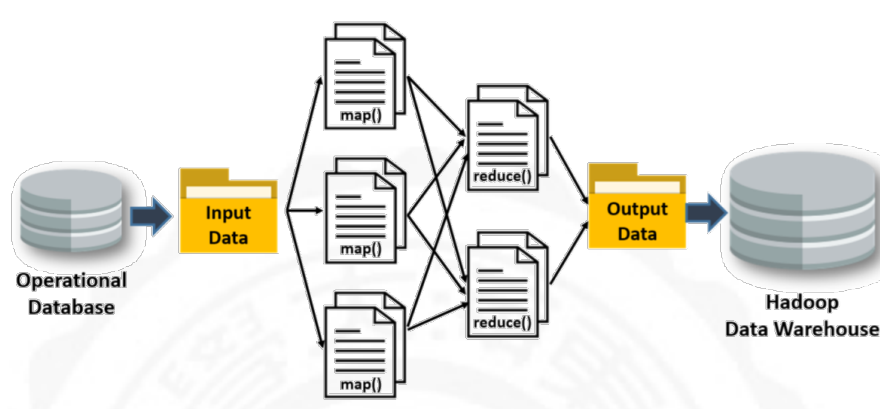


FIGURE 3.2: Sqoop Workflow for Data Transferring

3.2.2 Data Collection

For streaming data collecting generated from smart meters deployed all over the campus, we use Kafka [11] to catch and form the data stream as stream input source for Spark Streaming. The time of Spark Streaming calculation interval will be exactly same as the update speed of the original streaming data. In this way, the data stream formed by Kafka can be perfectly matched with Spark Streaming. The data stream received by Spark Streaming will be presented in the form of DStream. Writing DStream to HBase completes the collection of streaming data. The overall data collection process is shown in the figure 3.3 below.

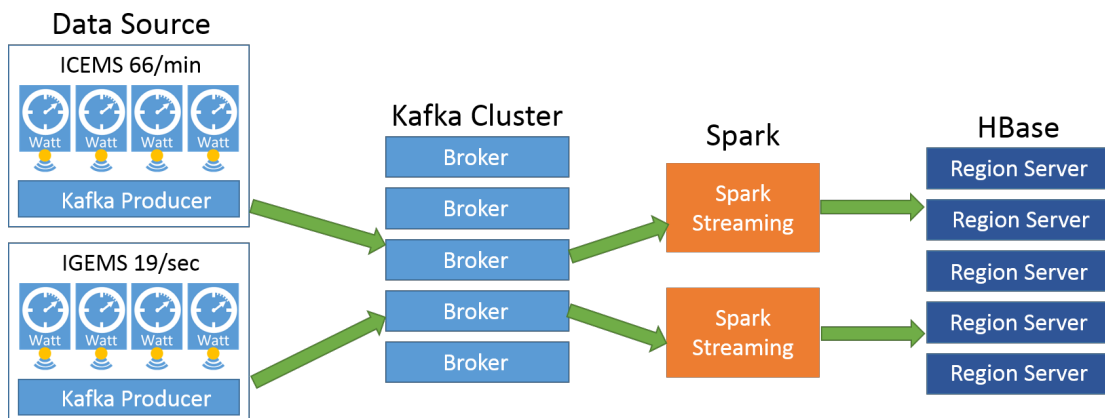


FIGURE 3.3: Data Collection Workflow

Figure 3.4 and Figure 3.5 show that Kafka’s producer catch the source APIs for ICEMS and IGMES and write it to Kafka cluster’s broker.

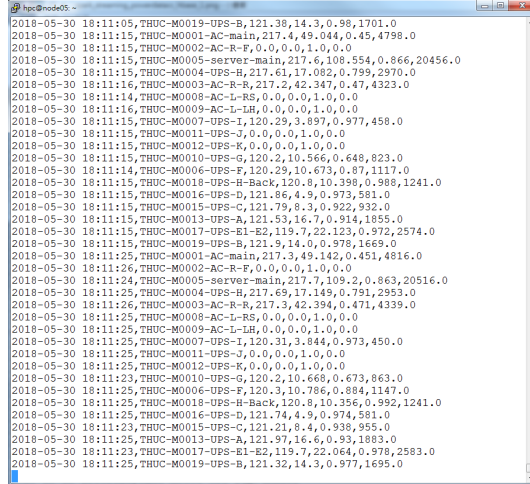


FIGURE 3.4: Kafka Producer for IGEMS

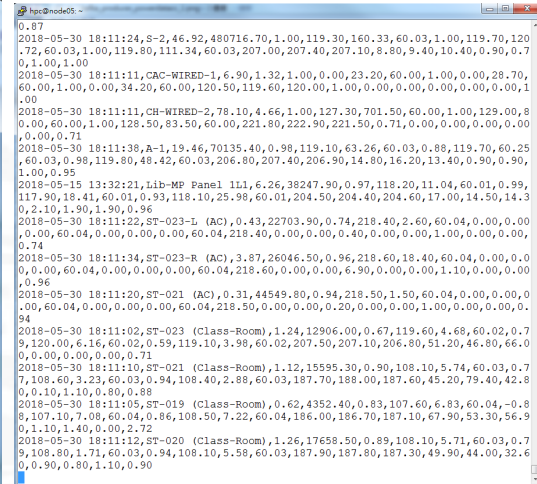


FIGURE 3.5: Kafka Producer for ICEMS

Spark Streaming will receive the data stream from Kafka (Dstream) and write it to HBase table through HBase API as shown in the figure 3.6 and figure 3.7.

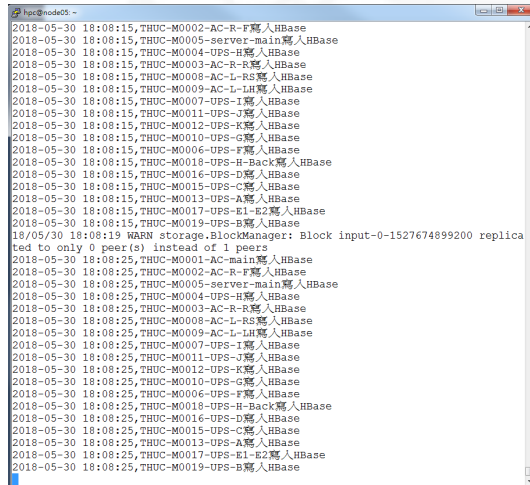


FIGURE 3.6: Spark Streaming Write Data to HBase for IGEMS

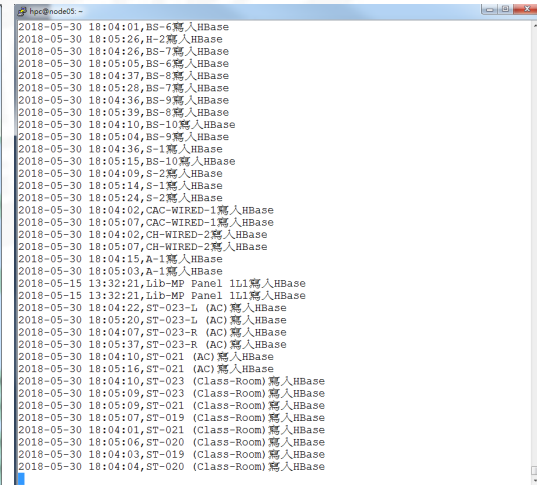


FIGURE 3.7: Spark Streaming Write Data to HBase for ICEMS

3.2.3 Data Storage

We use Hive and HBase as the basis for our Data Lake platform, both of them are based on HDFS. Hive will translates SQL syntax into map-reduce job to search data on HDFS, usually used for Big Data queries offline like shown in figure 3.8. We chose to transfer the historical data of the old system to Hive by Apache sqoop to facilitate the query and analysis of Big Data. HBase is responsible for storing streaming power data on our system because HBase has high throughput and low latency. Beasuse of these features. it is very suitable for faster reading and writing operations in Big Data. In addition, we provide two kinds of search engines for each of these two different feature databases. Two search engines, Apache Impala for Hive and Apache Phoenix for HBase, can provide them with excellent query performance with SQL individually.

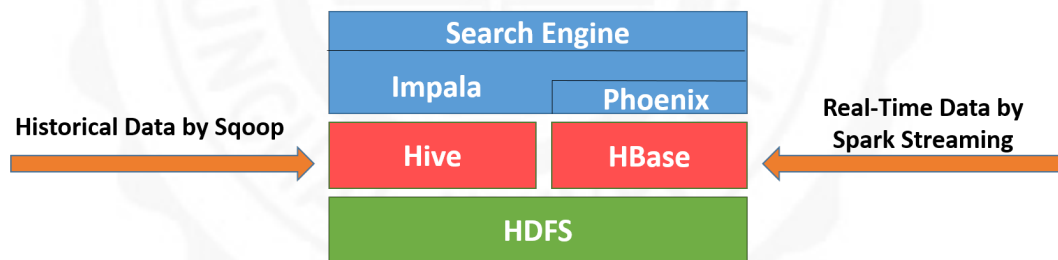
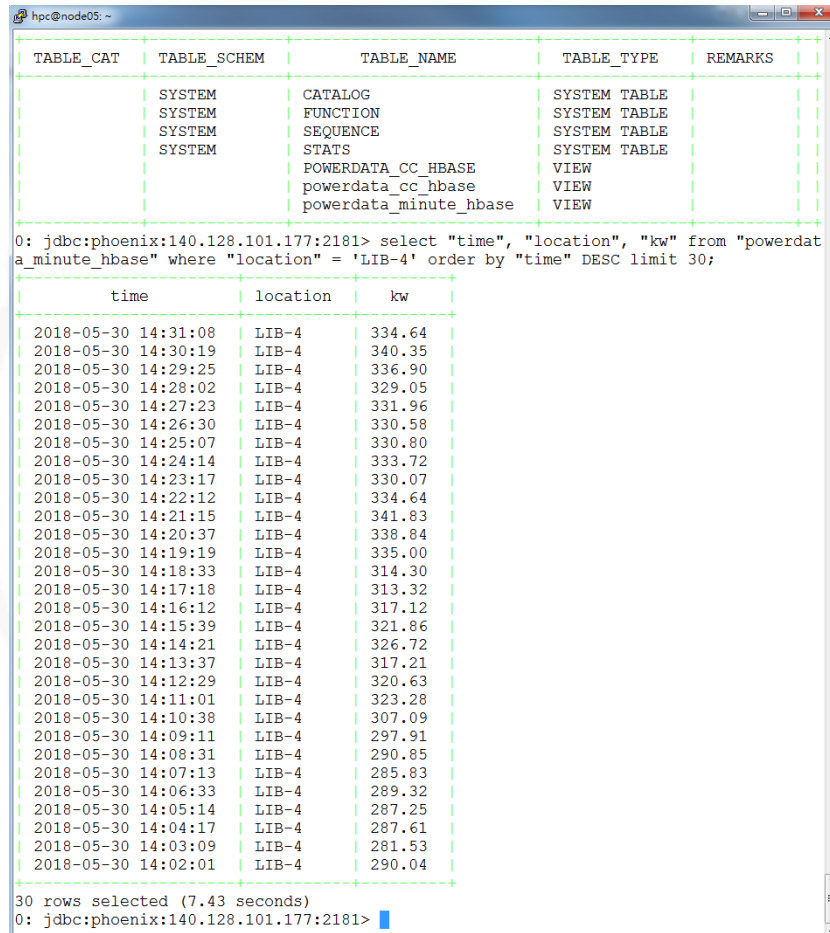


FIGURE 3.8: The Architecture of Data Storage

It is worth noting that Impala also has support for operating HBase, but using HBase native syntax does not support SQL syntax. But in fact, by mapping the HBase table to the Hive table, Impala can use Hive to operate HBase and use SQL syntax. But the reason we chose phoenix is that phoenix is a project specifically designed to provide SQL syntax for HBase. phoenix will translate the SQL syntax into HBase syntax to query. The overall performance is better than via Hive or Impala.

The overall storage architecture is based on HDFS. Data storage is divided into streaming and historical data. Streaming data is stored on HBase and phoenix is used as a search engine that user can use SQL syntax. Figure 3.9 shows the Phoenix-Shell.



```

hpc@node05: ~
┌───┴───┐
│ TABLE_CAT | TABLE_SCHEM | TABLE_NAME | TABLE_TYPE | REMARKS |
├───┬───┬───┬───┬───┘
│      | SYSTEM      | CATALOG     | SYSTEM TABLE |         |
│      | SYSTEM      | FUNCTION    | SYSTEM TABLE |         |
│      | SYSTEM      | SEQUENCE    | SYSTEM TABLE |         |
│      | SYSTEM      | STATS       | SYSTEM TABLE |         |
│      |              | POWERDATA_CC_HBASE | VIEW          |         |
│      |              | powerdata_cc_hbase | VIEW          |         |
│      |              | powerdata_minute_hbase | VIEW          |         |
└───┴───┴───┴───┴───┘

0: jdbc:phoenix:140.128.101.177:2181> select "time", "location", "kw" from "powerdata_cc_hbase" where "location" = 'LIB-4' order by "time" DESC limit 30;
┌───┬───┬───┬───┘
│ time | location | kw |
├───┬───┬───┬───┘
│ 2018-05-30 14:31:08 | LIB-4 | 334.64 |
│ 2018-05-30 14:30:19 | LIB-4 | 340.35 |
│ 2018-05-30 14:29:25 | LIB-4 | 336.90 |
│ 2018-05-30 14:28:02 | LIB-4 | 329.05 |
│ 2018-05-30 14:27:23 | LIB-4 | 331.96 |
│ 2018-05-30 14:26:30 | LIB-4 | 330.58 |
│ 2018-05-30 14:25:07 | LIB-4 | 330.80 |
│ 2018-05-30 14:24:14 | LIB-4 | 333.72 |
│ 2018-05-30 14:23:17 | LIB-4 | 330.07 |
│ 2018-05-30 14:22:12 | LIB-4 | 334.64 |
│ 2018-05-30 14:21:15 | LIB-4 | 341.83 |
│ 2018-05-30 14:20:37 | LIB-4 | 338.84 |
│ 2018-05-30 14:19:19 | LIB-4 | 335.00 |
│ 2018-05-30 14:18:33 | LIB-4 | 314.30 |
│ 2018-05-30 14:17:18 | LIB-4 | 313.32 |
│ 2018-05-30 14:16:12 | LIB-4 | 317.12 |
│ 2018-05-30 14:15:39 | LIB-4 | 321.86 |
│ 2018-05-30 14:14:21 | LIB-4 | 326.72 |
│ 2018-05-30 14:13:37 | LIB-4 | 317.21 |
│ 2018-05-30 14:12:29 | LIB-4 | 320.63 |
│ 2018-05-30 14:11:01 | LIB-4 | 323.28 |
│ 2018-05-30 14:10:38 | LIB-4 | 307.09 |
│ 2018-05-30 14:09:11 | LIB-4 | 297.91 |
│ 2018-05-30 14:08:31 | LIB-4 | 290.85 |
│ 2018-05-30 14:07:13 | LIB-4 | 285.83 |
│ 2018-05-30 14:06:33 | LIB-4 | 289.32 |
│ 2018-05-30 14:05:14 | LIB-4 | 287.25 |
│ 2018-05-30 14:04:17 | LIB-4 | 287.61 |
│ 2018-05-30 14:03:09 | LIB-4 | 281.53 |
│ 2018-05-30 14:02:01 | LIB-4 | 290.04 |
└───┴───┴───┴───┘

30 rows selected (7.43 seconds)
0: jdbc:phoenix:140.128.101.177:2181>

```

FIGURE 3.9: Phoenix Shell

The reason we don't use Hive to store streaming data is if we use spark streaming to write streaming data to Hive in this work, each write will create a folder in Hive's directory. As the time past, the overall Hive performance will be very slow, so we choose HBase as a database for high-speed write of streaming data.

Hive is used as a database of existing system data. Because the existing system data tables are large, Hive is suitable for cleaning and processing large amounts of data. In this work, Impala was used as Hive's search engine. Figure 3.10 shows the Phoenix-Shell.

```

hpc@node05: ~
[node05:21000] > select `time`, `meter_id`, `p`/1000 from powerdata_cc_hbase limit 20;
Query: select `time`, `meter_id`, `p`/1000 from powerdata_cc_hbase limit 20
Query submitted at: 2018-05-30 14:36:19 (Coordinator: http://node05:25000)
Query progress can be monitored at: http://node05:25000/query_plan?query_id=494e914b6c5a494b:ec28c995000000000
+-----+-----+-----+
| time           | meter_id       | p / 1000 |
+-----+-----+-----+
| 2018-03-15 22:06:10 | THUC-M0005-server-main | 19.522 |
| 2018-03-15 22:06:20 | THUC-M0005-server-main | 19.57 |
| 2018-03-15 22:06:30 | THUC-M0005-server-main | 19.525 |
| 2018-03-15 22:06:40 | THUC-M0005-server-main | 19.443 |
| 2018-03-15 22:06:50 | THUC-M0005-server-main | 19.535 |
| 2018-03-15 22:07:00 | THUC-M0005-server-main | 19.463 |
| 2018-03-15 22:07:10 | THUC-M0005-server-main | 19.387 |
| 2018-03-15 22:07:20 | THUC-M0005-server-main | 19.462 |
| 2018-03-15 22:07:30 | THUC-M0005-server-main | 19.532 |
| 2018-03-15 22:07:40 | THUC-M0005-server-main | 19.652 |
| 2018-03-15 22:07:50 | THUC-M0005-server-main | 19.431 |
| 2018-03-15 22:08:00 | THUC-M0005-server-main | 19.606 |
| 2018-03-15 22:08:10 | THUC-M0005-server-main | 19.478 |
| 2018-03-15 22:08:20 | THUC-M0005-server-main | 19.427 |
| 2018-03-15 22:08:30 | THUC-M0005-server-main | 19.305 |
| 2018-03-15 22:08:40 | THUC-M0005-server-main | 19.415 |
| 2018-03-15 22:08:50 | THUC-M0005-server-main | 19.456 |
| 2018-03-15 22:09:00 | THUC-M0005-server-main | 19.391 |
| 2018-03-15 22:09:10 | THUC-M0005-server-main | 19.296 |
| 2018-03-15 22:09:20 | THUC-M0005-server-main | 19.22 |
+-----+-----+-----+
Fetched 20 row(s) in 0.31s
[node05:21000] >

```

FIGURE 3.10: Impala Shell

Query History | Saved Queries | Results (1,024+) | Search

key	kw	location	time	totalkwh	
1	L-WIRED-1_2018-03-17 21:51	3.1	L-WIRED-1	2018-03-17 21:51:00	4.04
2	L-WIRED-1_2018-03-17 21:52	3.1	L-WIRED-1	2018-03-17 21:52:00	4.04
3	L-WIRED-1_2018-03-17 21:53	3.1	L-WIRED-1	2018-03-17 21:53:00	4.04
4	L-WIRED-1_2018-03-17 21:54	3.1	L-WIRED-1	2018-03-17 21:54:00	4.04
5	L-WIRED-1_2018-03-17 21:55	3.1	L-WIRED-1	2018-03-17 21:55:00	4.04
6	L-WIRED-1_2018-03-17 21:57	3.1	L-WIRED-1	2018-03-17 21:57:00	4.04
7	L-WIRED-1_2018-03-17 21:58	3.2	L-WIRED-1	2018-03-17 21:58:00	4.04
8	L-WIRED-1_2018-03-17 22:00	3.2	L-WIRED-1	2018-03-17 22:00:00	4.04
9	L-WIRED-1_2018-03-17 22:01	3.8	L-WIRED-1	2018-03-17 22:01:00	4.04
10	L-WIRED-1_2018-03-17 22:02	3.8	L-WIRED-1	2018-03-17 22:02:00	4.04
11	L-WIRED-1_2018-03-17 22:03	4.4	L-WIRED-1	2018-03-17 22:03:00	4.04
12	L-WIRED-1_2018-03-17 22:04	3.5	L-WIRED-1	2018-03-17 22:04:00	4.04
13	L-WIRED-1_2018-03-17 22:05	3.4	L-WIRED-1	2018-03-17 22:05:00	4.04
14	L-WIRED-1_2018-03-17 22:06	3	L-WIRED-1	2018-03-17 22:06:00	4.04
15	L-WIRED-1_2018-03-17 22:07	3	L-WIRED-1	2018-03-17 22:07:00	4.04
16	L-WIRED-1_2018-03-17 22:08	3	L-WIRED-1	2018-03-17 22:08:00	4.04
17	L-WIRED-1_2018-03-17 22:09	3	L-WIRED-1	2018-03-17 22:09:00	4.04
18	L-WIRED-1_2018-03-17 22:10	3	L-WIRED-1	2018-03-17 22:10:00	4.04
19	L-WIRED-1_2018-03-17 22:11	3.1	L-WIRED-1	2018-03-17 22:11:00	4.04
20	L-WIRED-1_2018-03-17 22:12	3.1	L-WIRED-1	2018-03-17 22:12:00	4.04
21	L-WIRED-1_2018-03-17 22:13	3.1	L-WIRED-1	2018-03-17 22:13:00	4.04
22					

FIGURE 3.11: Hue GUI Interface

In addition, the HBase table can be mapped to Hive to operate HBase through Hive. Hue is a GUI interface that can operate Hive like Figure 3.11 shows, HBase, and Impala through web pages. Finally, analysts can use impala and phoenix's api to obtain data when intercepting data, or connect Hive and HBase directly through spark SQL.

3.2.4 Data Analysis

In terms of data analysis, provides three analysis modules for this thesis.

1. Power failure analysis
2. Power forecasting(time-series)

Using Spark MLlib for machine learning of power data, and also using the time series model for electricity forecasting. Our proposed Algorithm 1 can calculate the situation of power failure on campus. First, organize the large amount of historical data stored on the Hive table into a time and meter ID format as shown in the Figure 3.12. Then In Algorithm 1, use Spark SQL to read the Hive table and sort the time difference between the two, as long as more than 5 minutes and less than 180 minutes, it is determined that the power-off or power failure occurs and compares the status of these two meters to eliminate the single meter failuration. The results of the analysis are written to HBase and visualized by Superset.

	_c0	meter_id
1	1516269960	THUC-M0019-UPS-B
2	1516269900	THUC-M0019-UPS-B
3	1516269840	THUC-M0019-UPS-B
4	1516269780	THUC-M0019-UPS-B
5	1516269720	THUC-M0019-UPS-B
6	1516269660	THUC-M0019-UPS-B
7	1516269600	THUC-M0019-UPS-B
8	1516269540	THUC-M0019-UPS-B
9	1516269480	THUC-M0019-UPS-B
10	1516269420	THUC-M0019-UPS-B
11	1516269360	THUC-M0019-UPS-B
12	1516269300	THUC-M0019-UPS-B

FIGURE 3.12: Table Format

Algorithm 1 Power Failure Algorithm

```

1: meter1  $\leftarrow$  array for Meter1's power data
2: meter2  $\leftarrow$  array for Meter2's power data
3: timeGap  $\leftarrow$  meter(0) – meter(1)
4: for  $i = 1; i < \text{meter1.length}; i++$  do
5:   if (five minute < timeGap < three hours) then
6:     ArrayBuffer1  $\leftarrow$  power failure time for meter1
7:   end if
8: end for
9: for  $i = 1; i < \text{meter2.length}; i++$  do
10:  if (five minute < timeGap < three hours) then
11:    ArrayBuffer2  $\leftarrow$  power failure time for meter2
12:  end if
13: end for
14: Power Failure Time  $\leftarrow$  ArrayBuffer1  $\cap$  ArrayBuffer2

```

In terms of predicting electricity use, taking into account that the characteristics of electricity consumption belong to the time series, it may be quite stable or present a certain trend over time and have seasonal characteristics. Due to the above characteristics, this paper uses Holt Winters algorithm to predict the power trend. After experiments confirmed that the forecast method is to use the power data of the previous two days as training data, the data of the next day forecasting will be the most accurate. In addition, due to the relationship between campus routines, training data set for Monday, Tuesday, Saturday and Sunday were selected at the same time for last week. HoltWinters provides addition and multiplication methods. Additive method is preferred when seasonal variations are roughly constant through the series. Multiplicative method is preferred when the seasonal variations are changing. However, because we have only one day to predict, we don't have too significant seasonal changes and we chose Holt Winters' additive method.

Algorithm 2 HoltWinters forecasting Algorithm

```

1: getTrainData ← get training data from DB
2: ts ← transform training data to dense vector
3: PredictArray ← array for writing forecasting data to DB
4: model = HoltWinters.fitModel(ts, Period, "Method", "BOBYQA")
5: forecast = model.forecast(ts, ts)
6: for i = 0; i < 23; i ++ do
7:   PredictArray(i)(0) = Date
8:   PredictArray(i)(1) = Hour
9:   PredictArray(i)(2) = Location
10:  PredictArray(i)(3) = Predicted Value
11: end for
12: forecastDF = (PredictArray.toRDD, schema)
13: forecastDF.write.mode("append").(DB)

```

In algorithm 2, First, the data to be trained is obtained from the database. The data type is dataframe. Then it is transformed into a dense vector and HoltWinters training model is used. The algorithm has three parameters that can be optimized. The first is TS, TS is the training data set, and the second is Period that means seasonality of data i.e period of time before behavior begins to repeat itself. The third parameter is modeltype, two variations differ in the nature of the seasonal component. Additive method is preferred when seasonal variations are roughly constant through the series, Multiplicative method is preferred when the seasonal variations are changing proportional to the level of the series. Then the power forecasting function is used to predict the power consumption trend in the next 24 hours, and the result is converted into RDD combined with schema and converts to dataframe and writes back to database.

3.2.5 Data Visualization

Superset supports direct query of multiple data sources, including Hive, Impala and other data sources. However, HBase is not supported, but by mapping HBase table to Hive table, Superset can visualize HBase data through Impala. The visualization part of the system in the thesis is connected to the data source with Impala. Superset extracts real-time data from HBase and historical data from Hive. The result of any analysis from this system are stored in HBase.

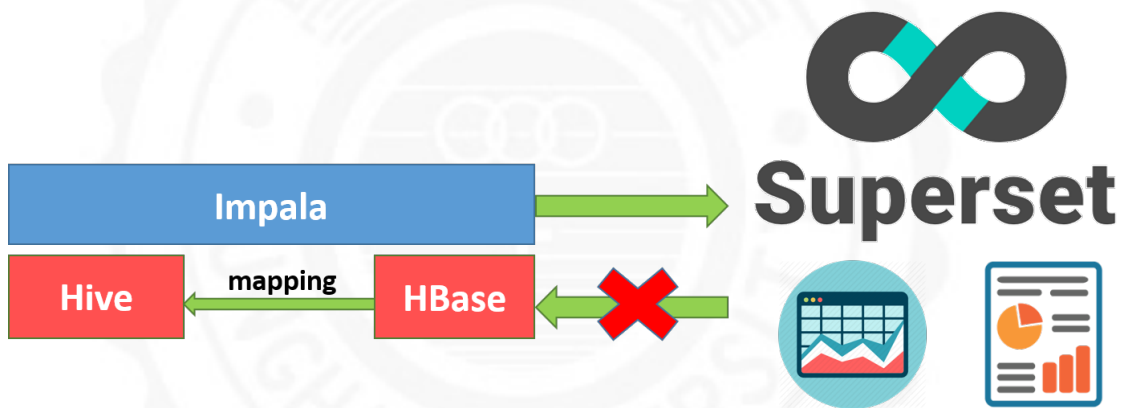


FIGURE 3.13: The Architecture of Superset

SQLAlchemy URI is the way that Superset connects to the database, The Engine is the starting point for any SQLAlchemy application. It is “home base” for the actual database and its DBAPI, delivered to the SQLAlchemy application through a connection pool and a Dialect, which describes how to talk to a specific kind of database/DBAPI combination. The general structure can be illustrated as 3.14.

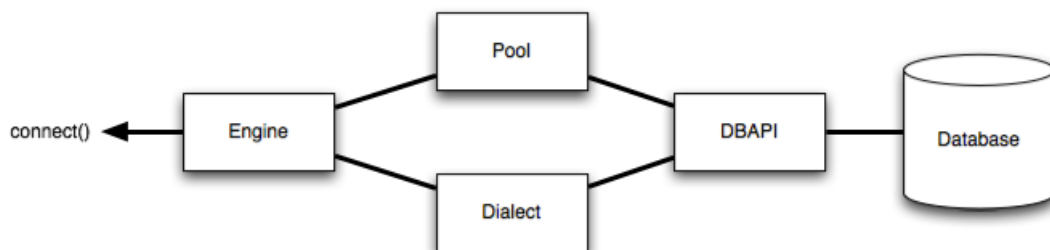


FIGURE 3.14: The Structure of SQLAlchemy

Therefore, as long as the specified URI is set, the specified database is connected and further data exploration is performed. Taking this thesis as example, Impala's URI is `impala://140.128.101.177:21050/` and tests whether the connection connection is ok or not like 3.15.

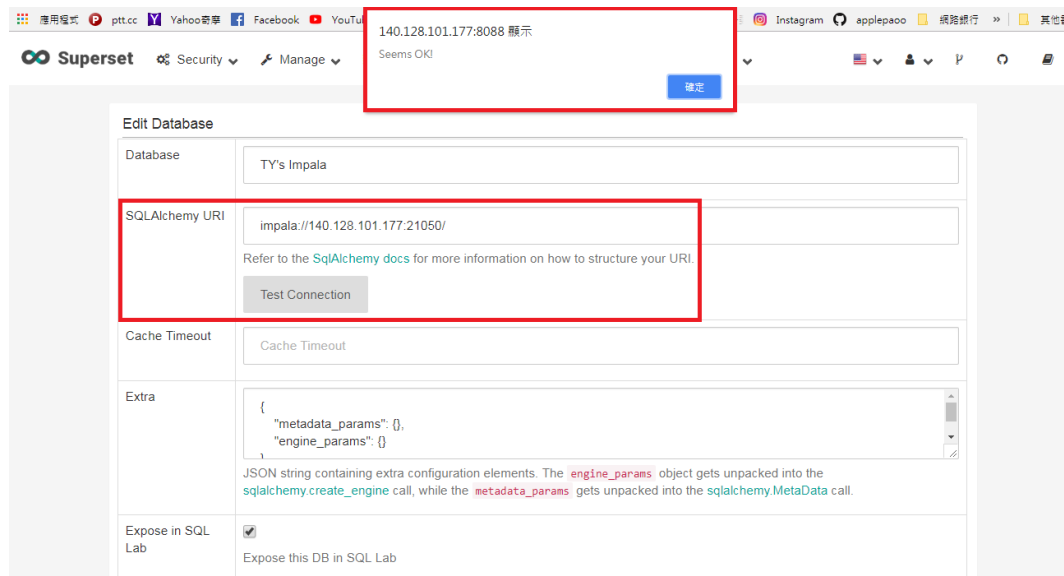


FIGURE 3.15: Editing Database's SQLAlchemy

After completing the connection setting, you can display a lot of charts on the data of the database. Taking the 3.16 as an example, the electricity consumption of each meter is displayed in Word Cloud. The larger the font is, the greater the power consumption is. There are some detailed adjustments on the left side that can be selected like time, font size range, and so on.



FIGURE 3.16: Word Cloud Chart

3.3 System Implementation

In this work, we have established the Big Data cluster for our Data Lake system through thirteen physical machines, one node as master, twelve nodes as the computing node to set up Cloudera Big Data platform that including CDH (Cloudera Distribution Including Apache Hadoop), Apache Spark, Apache Kafka, Apache Sqoop, Apache Hive, Apache Hive, Apache Phoenix and Apache Impala. Table 3.1 shows the software specification of thirteen cluster nodes.

TABLE 3.1: Software Specifications

	Version
Cloudera Manager	5.14.0
Hadoop	hadoop-2.6.0+cdh5.14.0+2714
HDFS	hadoop-2.6.0+cdh5.14.0+2714
Spark	spark-2.0.0+cdh5.14.2+543
Sqoop	sqoop2-1.99.5+cdh5.14.0+47
Hive	hive-1.1.0+cdh5.14.0+1330
HBase	hbase-1.2.0+cdh5.14.0+440
Impala	impala-2.11.0+cdh5.14.0+0
Phonenix	Phoenix 4.7.0 on CDH5.7
HUE	hue-3.9.0+cdh5.14.0+7830
Kafka	0.11.0-kafka-3.0.0

The Data Lake platform was built through cloudera. The monitoring platform provided by cloudera allows us to clearly understand the health status, CPU usage, memory usage, cluster network IO, and other related information of cluster services as the figure 3.17 shows.

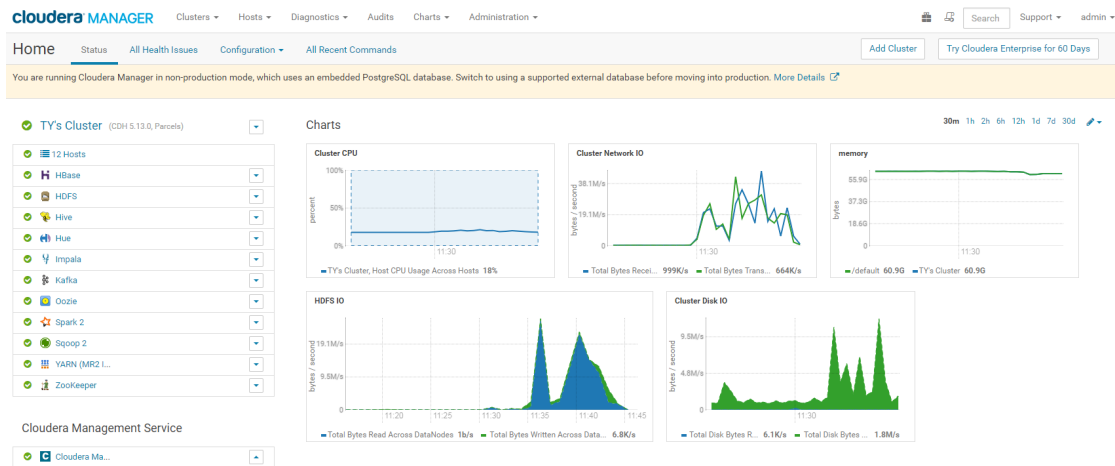


FIGURE 3.17: Cloudera Manager Web User Interface

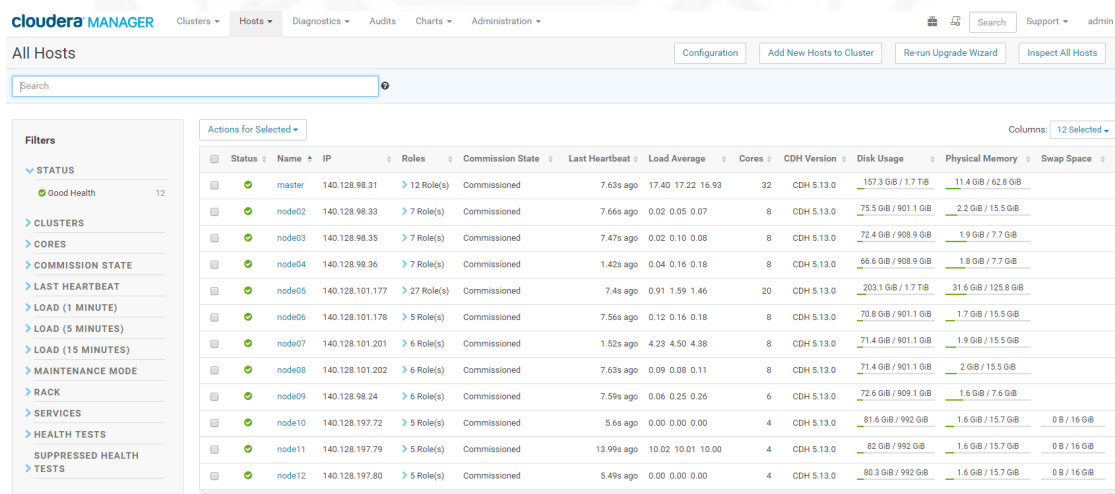


FIGURE 3.18: Nodes of Cloudera Cluster

We use HUE to provide a UI interface for Hive, HBase, and Impala. Hue not only supports to manipulate data in Apache Hadoop ecosystem, but also provides corresponding dynamical search dashboard with Solr. The most important is that it support interactive query of HiveQL, Impala and HBase/ In figure 3.19 we can see the visualization interface of Hue.

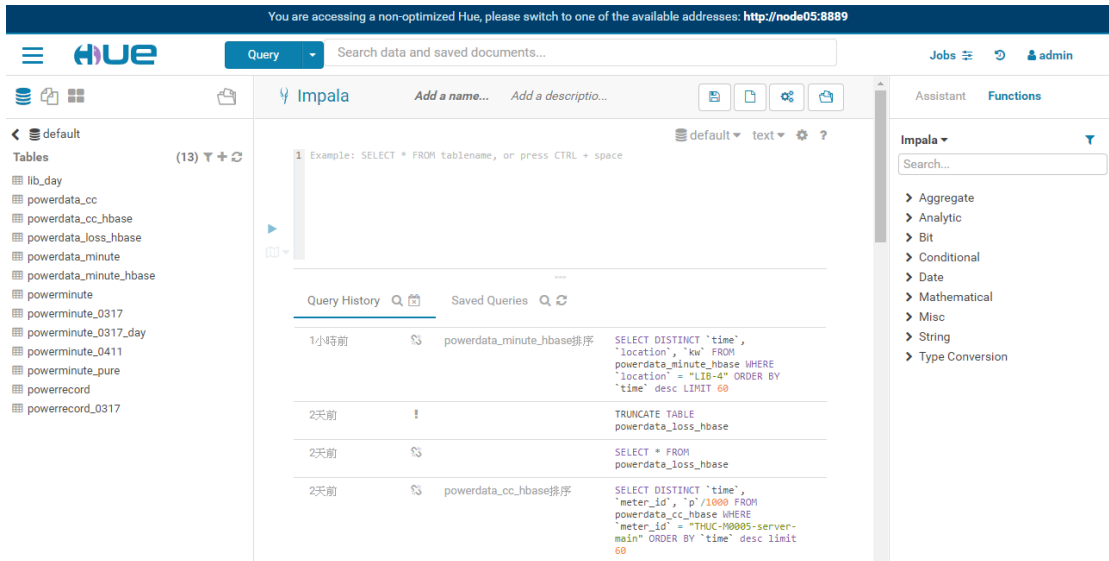


FIGURE 3.19: HUE Web User Interface

The visualization part of this system is to use Apache Superset as our solution. We integrate Hive, HBase, Impala as the data source of Superset. Superset provides many kinds of charts to user to choose whatever they want to use for their data like Distribution-Bar Chart, Pie Chart, Dual Axis Line Chart, etc. Each chart is called a slice, a dashboard can be composed of multiple slices like figure 3.20.

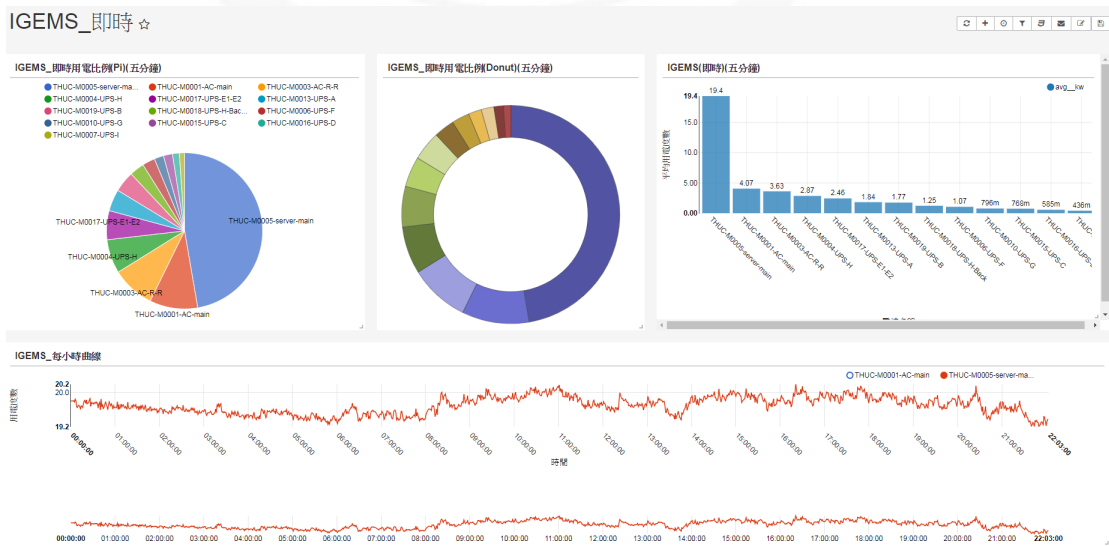


FIGURE 3.20: Superset Dashboard for Power Data

Chapter 4

Experimental Results

In this section, the experimental environment and results of the Power Data Storage and Analysis Platform with Spark and Data Lake are described. In section 4.1, we introduce the experimental environment and implementation of the proposed system. Sections 4.2 to 4.5 show the experiment of performance tests for verifying the efficiency of the system.

4.1 Experimental Environment

This section presents our hardware experimental environment. The proposed system is implemented with 13 physical servers connected by Gigabit Ethernet to build a computing cluster. Each physical server consists of Intel Core i7 CPU with 16 GB Memory and 1TB HD. Besides, Ubuntu 14.04 is adopted as our operating system. Also, Hadoop 2.6.0-cdh5.14.0, Spark 2.0.0, Sqoop 1.4.5, Hive 1.2.1, and HBase 1.0.0 are installed, as shown in Table 4.1 and Figure 4.1

TABLE 4.1: Experimental Environment

	CPU	RAM	DISK
Master	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz	64G	2T
Node01	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	16G	1T
Node02	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	16G	1T
Node03	Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz	8G	1T
Node04	Intel(R) Xeon(R) CPU E3-1230 v3 @ 3.30GHz	8G	1T
Node05	Intel(R) Core(TM) i7-6950X CPU @ 3.00GHz	128G	2T
Node06	Intel(R) Core(TM) i7-4770 CPU @ 3.40GH	16G	1T
Node07	Intel(R) Core(TM) i7-4770 CPU @ 3.40GH	16G	1T
Node08	Intel(R) Core(TM) i7-4770 CPU @ 3.40GH	16G	1T
Node09	AMD Phenom(tm) II X6 1055T Processor	8G	1T
Node10	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz	16G	1T
Node11	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz	16G	1T
Node12	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz	16G	1T



FIGURE 4.1: Cloudera Cluster

4.2 The Speed of Transferring Historical Data

For the transfer of historical data, we use Apache Sqoop to transfer 7G and 17G data separately for the two data tables of the existing storage system, and

used the characteristics of parallelism to speed up the overall movement of data. From the experiment as Figure 4.2 and Figure 4.3 show, we can find that increasing the number of maps can effectively reduce the time for moving, but we must notice that when the number of maps is too large, the data will take more time on the network communication to reduce the data.

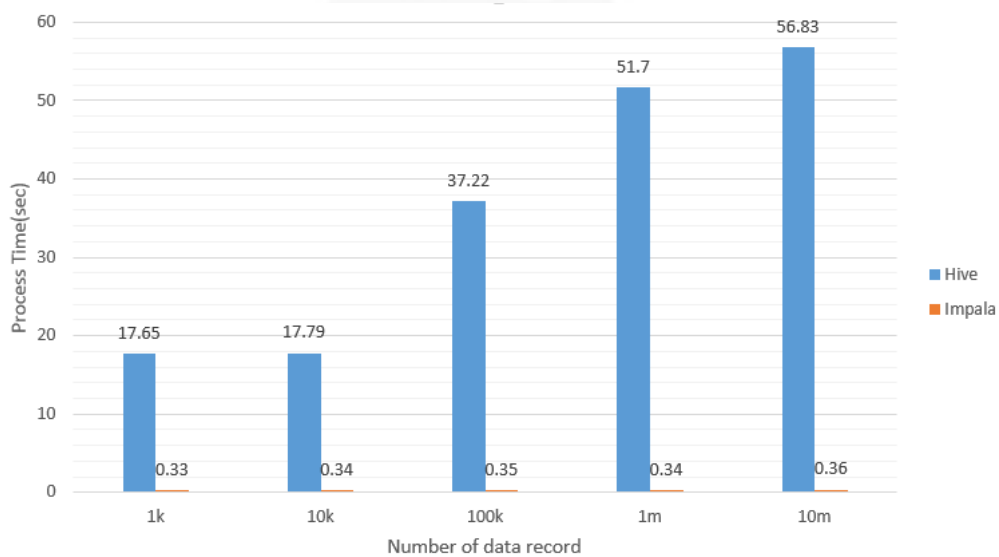


FIGURE 4.2: Comparison of m Number for 7G Table

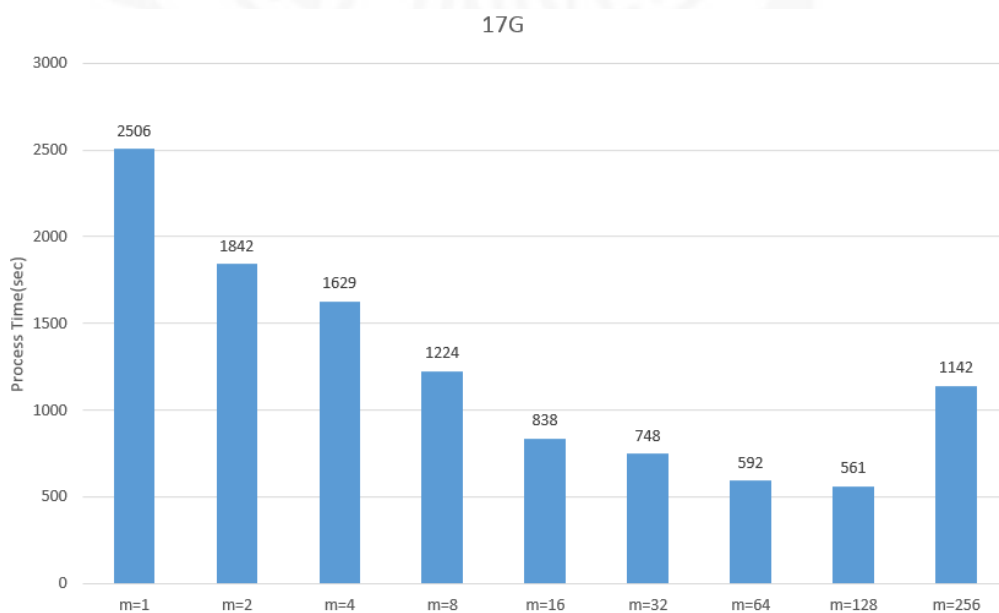


FIGURE 4.3: Comparison of m Number for 17G Table

4.3 Data Lake's Comparison of Different Search Engines

We choose Apache Impala as Hive's search engine in our system. Figure 4.4 shows that Impala is much better than hive in the performance of searching data because Hive converts each SQL query to map-reduce job to search data on HDFS and each stage of reading and writing will access disk. However, Impala reduces the phase of accessing disk reads and writes and uses a large amount of memory characteristics to calculate data. So using Impala as a search engine for Hive is proper.

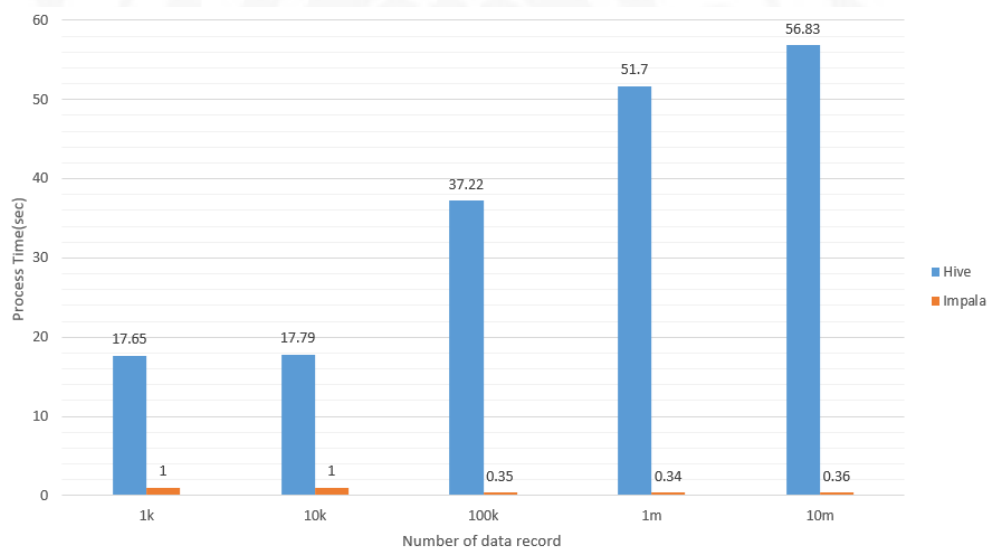


FIGURE 4.4: Execution Time of Searching by Hive and Impala

Figure 4.5 shows that Hive is suitable for batch processing with complex data, and Impala can analyze the results of Hive in real time. The combination of these two component are very effective.

4.4 Streaming Data Storage

For the storage of streaming data, we choose HBase as our system's database. However, HBase does not support SQL-like syntax to query data. Instead, it use its own shell commands to manipulate the data stored in HBase but it's unfriendly

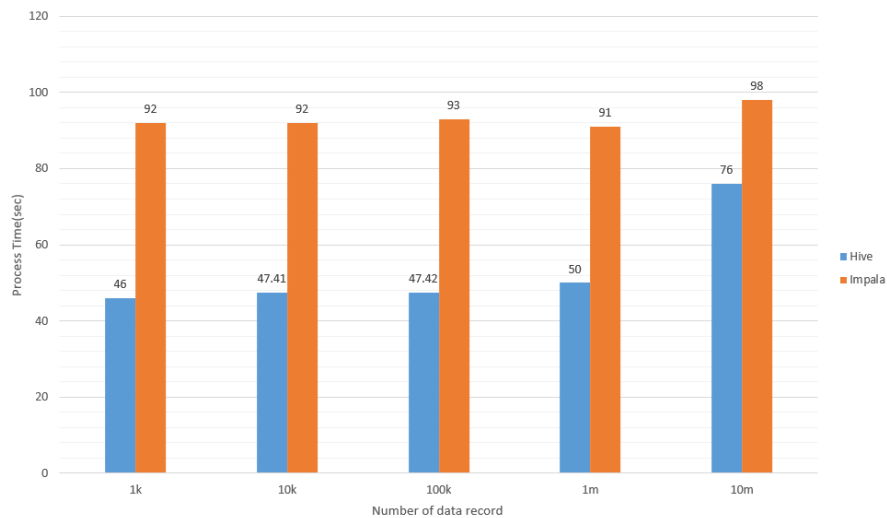


FIGURE 4.5: Execution Time of Ordering by Hive and Impala

for the user who is familiar with SQL-like syntax. So in our architecture we use Apache Phoenix as the searching engine for SQL processing over HBase and also compare Impala, Hive, Phoenix to manage HBase table. Figure 4.6 shows that Hive takes much more time than the other three because Hive converts the task of query into the map-reduce type. Both Phoenix and Impala have excellent performance in query speed in searching one data condition.

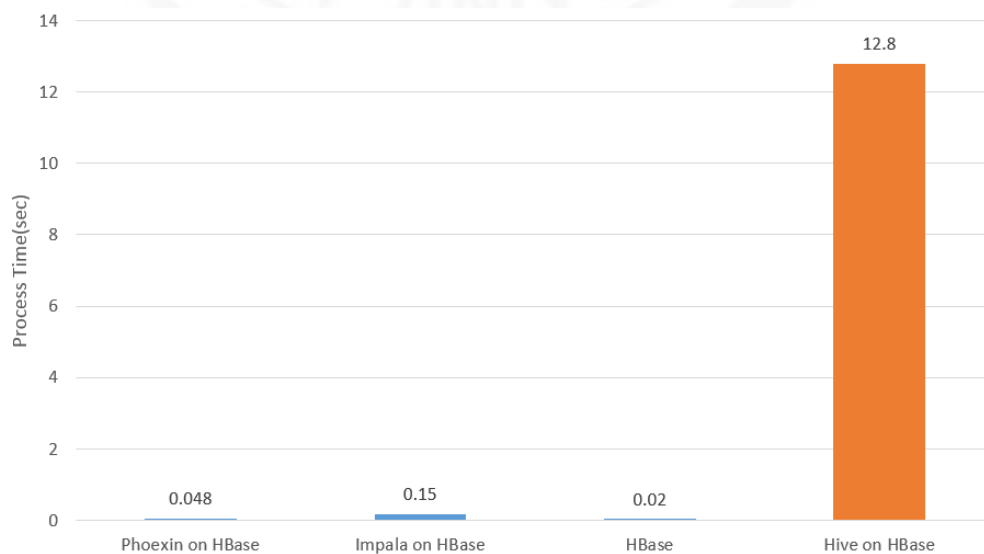


FIGURE 4.6: Execution Time of Searching by Phoenix, Hive and Impala

Comparing Phoenix and Impala execution times to search for meter's data of different unit times. Figure 4.7 shows that no matter what the unit time, phoenix performance is better than Impala, because phoenix's query syntax will be converted into HBase statement, the optimization of HBase is better

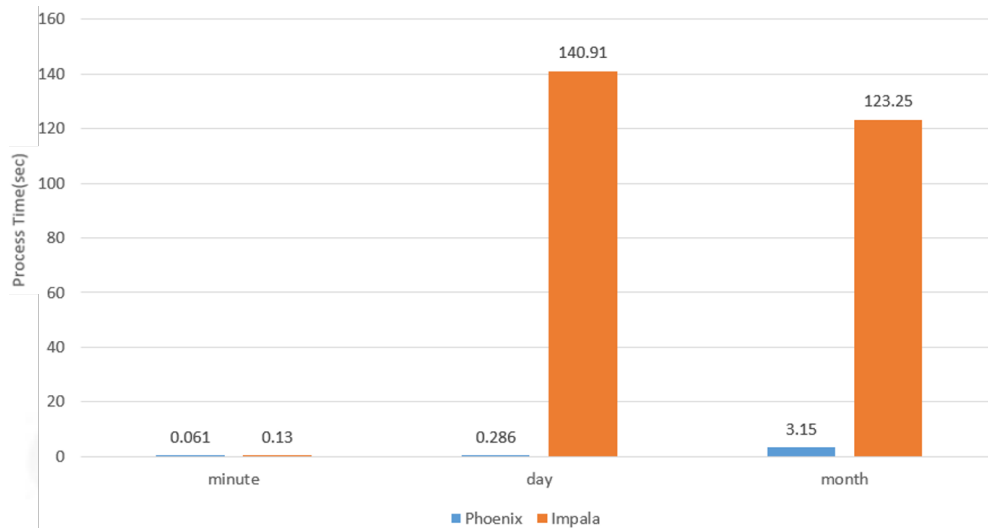


FIGURE 4.7: Execution Time of Searching by Phoenix and Impala

Figure 4.8 shows the speed comparison for counting data in HBase. The experimental result shows that Apache Phoenix has the best performance than others because Phoenix converts SQL query to HBase-specific grammar for query, so it can provide the better performance on HBase.

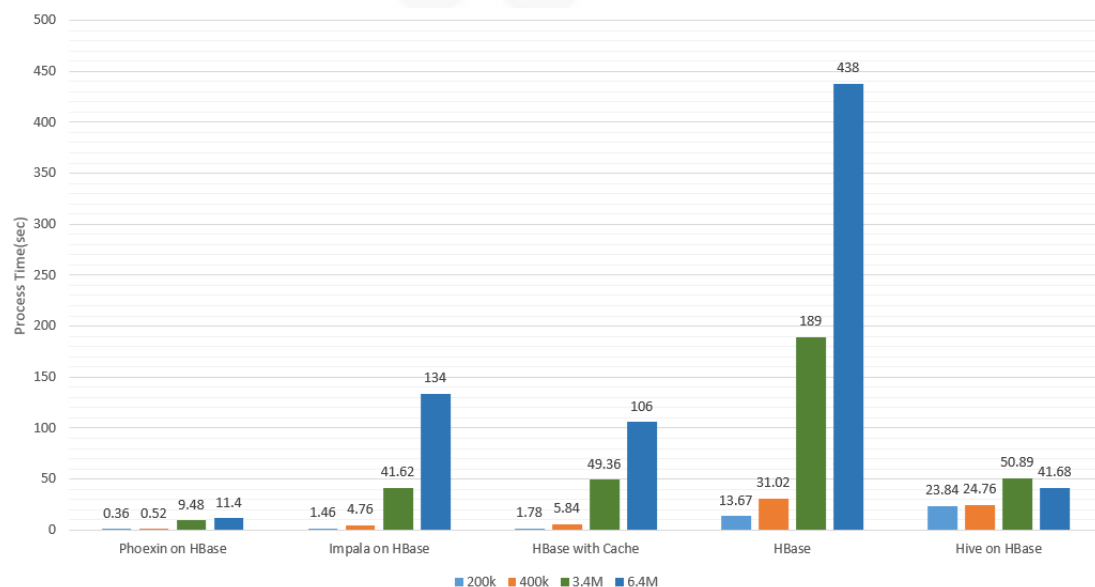


FIGURE 4.8: Execution Time of Counting by Phoenix, Hive and Impala

4.5 Power Failure Analysis Results

Through the algorithm of power failure we proposed in previous section, we can calculate the number of power failure from historical data and visualize analysis results by Apache Superset into table and the chart of Columnar distribution

斷電分析表 ☆ [🔗](#) 00:00:01.73 🔍 <> 📄 json 📄 csv View Query

date	recovery_date	avg_total_time
2018-04-02 07:04:00	2018-04-02 07:29:00	25.0
2018-01-19 14:03:00	2018-01-19 15:02:00	59.0
2018-01-04 19:35:00	2018-01-04 19:43:00	8.00
2018-01-04 18:23:00	2018-01-04 18:35:00	12.0
2018-01-04 00:50:00	2018-01-04 00:59:00	9.00
2017-12-25 17:34:00	2017-12-25 18:17:00	43.0
2017-12-18 11:25:00	2017-12-18 11:35:00	10.0
2017-12-18 10:31:00	2017-12-18 11:00:00	29.0
2017-11-03 09:35:00	2017-11-03 09:59:00	24.0
2017-09-10 18:14:00	2017-09-10 18:30:00	16.0
2017-07-06 15:21:00	2017-07-06 15:35:00	14.0
2017-03-16 14:15:00	2017-03-16 14:27:00	12.0
2017-02-28 19:31:00	2017-02-28 19:44:00	13.0
2017-01-18 23:23:00	2017-01-18 23:30:00	7.00
2016-12-28 22:20:00	2016-12-28 22:31:00	11.0
2016-12-28 20:28:00	2016-12-28 20:39:00	11.0
2016-12-27 17:49:00	2016-12-27 19:59:00	130
2016-12-26 03:27:00	2016-12-26 03:34:00	7.00
2016-09-23 16:20:00	2016-09-23 19:15:00	175
2016-08-03 15:50:00	2016-08-03 16:02:00	12.0

FIGURE 4.9: The Table of Power Failure

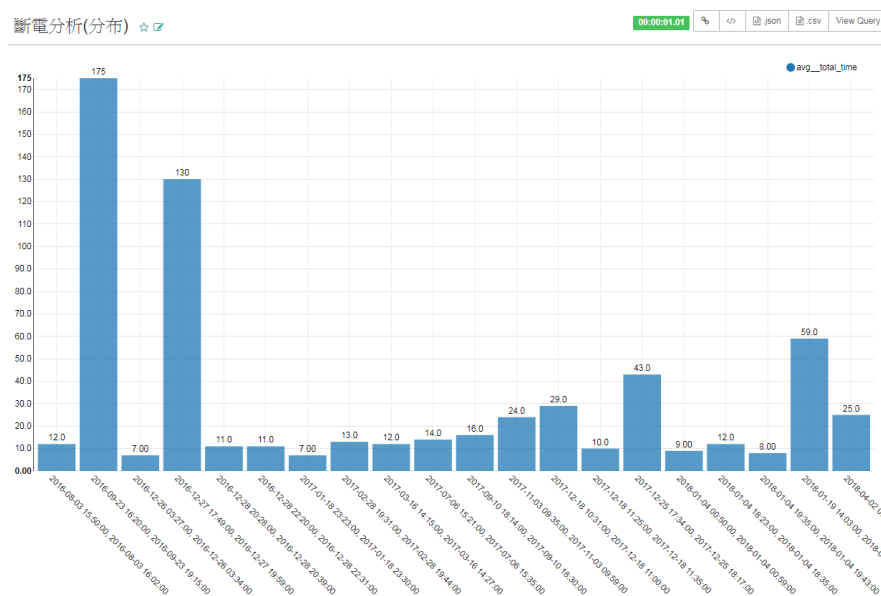


FIGURE 4.10: Bar Chart for Power Failure

4.6 Verify Power Forecasting Accuracy with MAPE

In this work, the HoltWinters algorithm is used for forecasting the power consumption situation in a day. In order to verify the accuracy of the prediction and actual values, we use the MAPE test to verify the accuracy of prediction. Assuming that n group actual values are: $v_1 \cdot v_2 \cdot v_3 \dots v_n$ and the predicted values are $p_1 \cdot p_2 \cdot p_3 \dots p_n$.

Calculate the percentage error of the actual value and the predicted value first from i group. The percentage error is the absolute value of p_i minus v_i , and then divided by the i group of actual values v_i .

$$error_i = \left| \frac{p_i - v_i}{v_i} \right| \quad (4.1)$$

Find each group's percentage errors: $error_1 \cdot error_2 \cdot error_3 \dots error_i \dots error_n$. the n group's percentage errors are averaged to find $MAPE$:

$$MAPE = \frac{\sum_{i=1}^n \left| \frac{p_i - v_i}{v_i} \right|}{n} \quad (4.2)$$

From the above formula 4.1, 4.2 can find out that MAPE is the average of the percentage of the predicted value to the actual value. So the smaller the MAPE value is, the higher the accuracy is. The higher the MAPE value is, the lower the accuracy is. If MAPE is greater than 50 %, there is no reference value for this group of data.

TABLE 4.2: MAPE error value level

MAPE	Ability to predict
<10%	high accuracy
10% – 20%	good
20% – 50%	reasonable
>50%	incorrect

Figure 4.11 shows that 3/12 full week power data trend used HoltWinters for forecasting. Red is the predicted value and blue is the actual value. It can be

seen that the overall trend from Monday to Friday can be predicted precisely. However, due to the prediction on Saturday and Sunday, the model can only be trained through historical data from last week and the number of people may also change on Saturday and Sunday because of the uncertainty in the weekend.

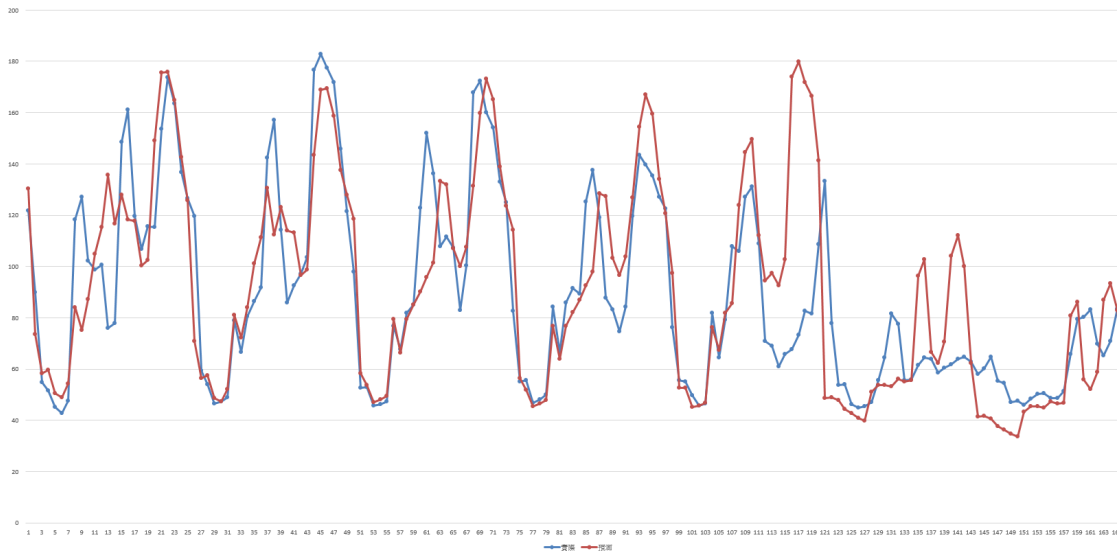


FIGURE 4.11: Comparison of actual and predicted values of 0312 week

Table 4.3 shows the daily MAPE values and the weekly average MAPE values. From the table, we can see that the MAPE values all fall between 10% and 35%, and the average MAPE value of the week is 20.17%. Overall, the forecasts are in the range between reasonable and good.

TABLE 4.3: Daily MAPE value for 0312 week

date	MAPE
3/12	17.96%
3/13	11.16%
3/14	11.10%
3/15	13.98%
3/16	35.78%
3/17	25.95%
3/18	25.28%
average	20.17%

Figure 4.12 only shows the actual values and predicted values from Monday to Friday. Red is the predicted value and blue is the actual value. From 4.12 we can see that Monday to Friday can still accurately predict the trend of power consumption.

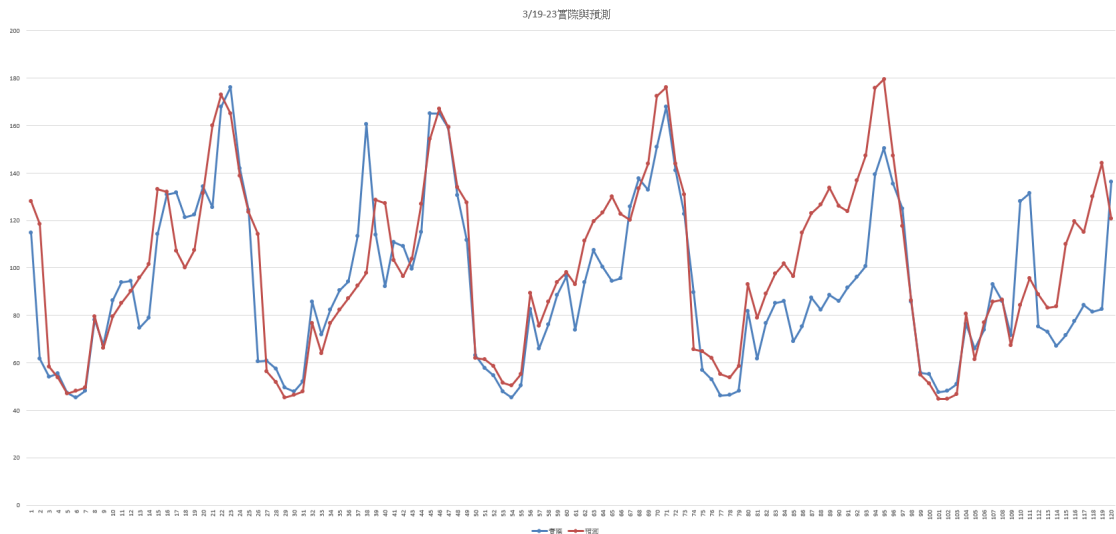


FIGURE 4.12: Comparison of actual and predicted values of 0319 week

It can also be seen from the Table 4.4 that the MAPE values from Monday to Friday are between 11% and 28%, and the average MAPE value is 17.24%.

TABLE 4.4: Daily MAPE value for 0319 week

date	MAPE
3/19	13.04%
3/20	13.44%
3/21	11.78%
3/22	28.14%
3/23	19.80%
average	17.24

From the above two experiments, we can infer that the forecast used by HoltWinters algorithm has a good accuracy of power consumption. The average MAPE value is between 10% and 20%, which is a good prediction type. The proposed platform enables highly accurate trend prediction of power usage data.

4.7 Superset Visualization

We map the HBase table to Hive, so Superset can visualize the data from HBase through Impala. The following figures are the results of analysis written to HBase and visualized by superset. Figure 4.13 shows the times series data collected by the two power meters of the library in hours.

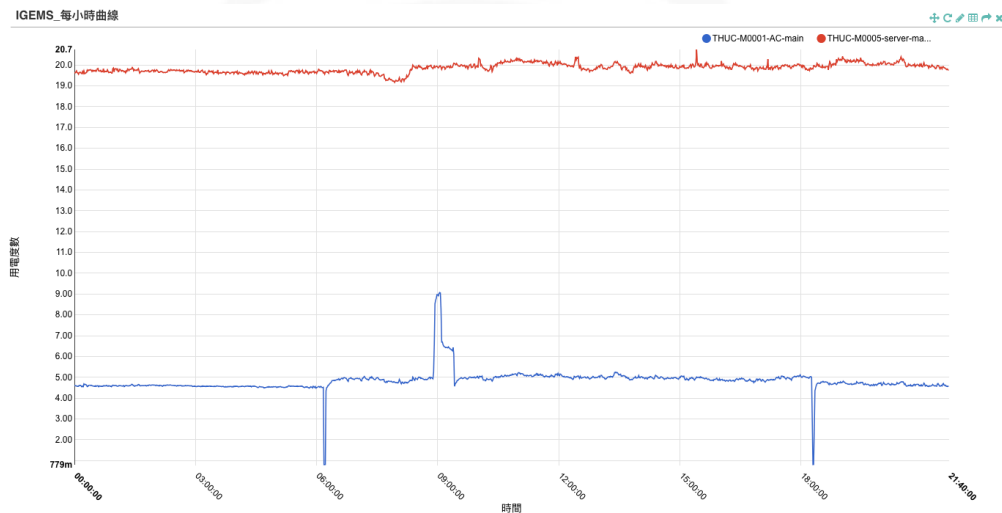


FIGURE 4.13: Time Series Chart for Power Data

Figure 4.14 shows the power data collected by Smart Meters deployed over the campus in the form of Pi Chart.

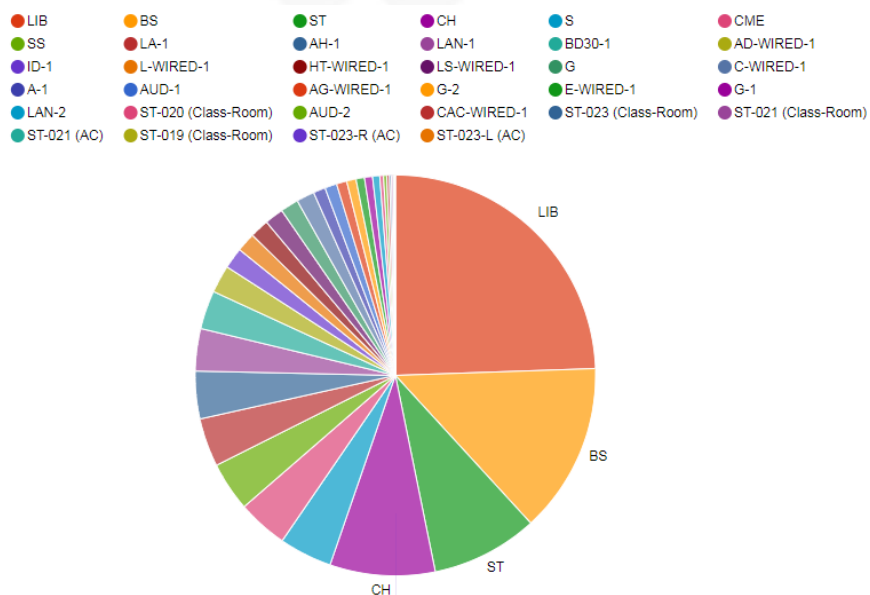


FIGURE 4.14: Pi Chart for Power Data

Figure 4.15 shows the rank of the top 10 power consumption ratios for campus buildings

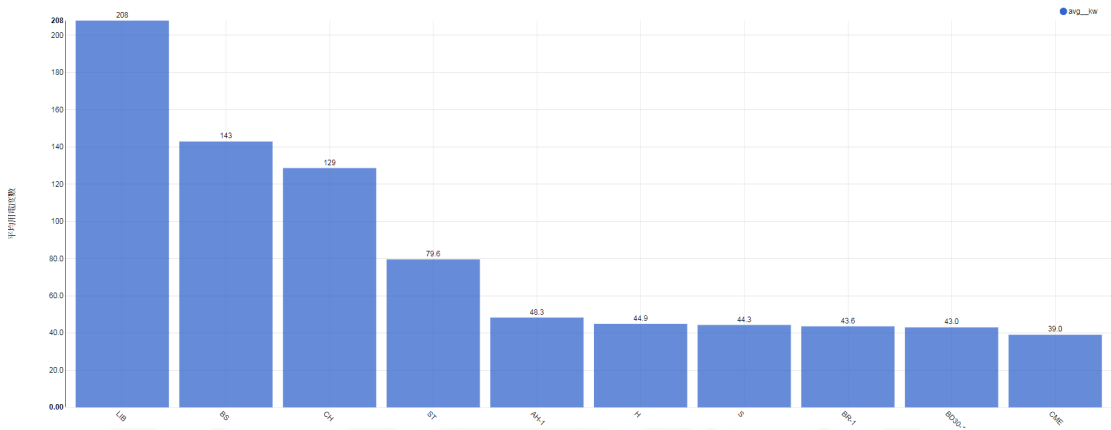


FIGURE 4.15: Bar Chart for Power Data

We also made Superset dashboards for IGEMS and ICEMS individually.

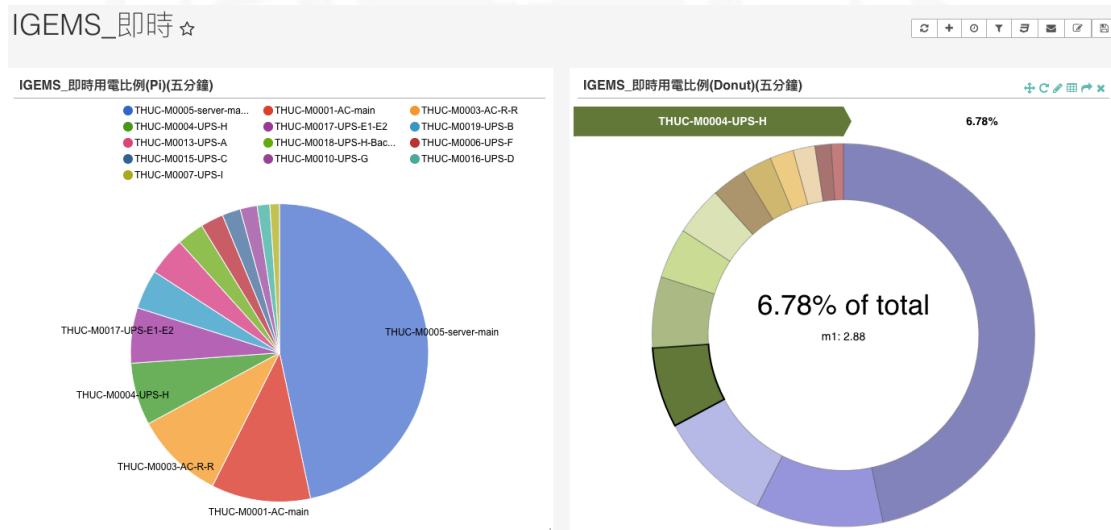


FIGURE 4.16: Dashboard fro IGEMS

Chapter 5

Conclusions and Future Work

A lot of smart meters to collect electricity data in the campus buildings and data center have been deployed. In the current mainstream platform for processing Big Data is Hadoop and Spark which support the traditional database only with the JDBC to get data, the cost of communication to obtain data for Hadoop or Spark is too high. To solve this problem, we proposed an architecture to import existing storage system to Big Data platform with Data Lake.

5.1 Concluding Remarks

This work presents an architecture of entirely open source solution integrating state-of-the-art components from the Apache ecosystem. It can efficiently import existing power data storage system into Big Data platform with Data Lake. The existing system's historical data is transferred to Apache Hive by Sqoop for the data warehouse. The streaming data is written into HBase to save the streaming data. The Data Lake is based on Hive and HBase to keep the integrity of the original data. We Integrate Impala and Phoenix as Data Lake's search engine and provide better search performance and power forecasting models are proposed in this work by Spark can help schools make better decisions. In conclusion, this work proposes a complete solution of Data Lake and Big Data platforms from

the data transfer, collection, storage, analysis, visualization to campus electricity environment.

5.2 Future Work

Constrained by HBase Scan performance, Superset searches for large data in HBase and causes the chart to display slowly, failing to reach a second-level response. In the future, we hope to speed up Superset's ability to retrieve data from HBase. In addition, the deployment of the physical machine environment may also be one of the reasons that affect the overall cluster performance. We hope that we can unify the machine specifications and cooperate with high-speed networks to improve the overall performance of Data Lake. In addition, the power forecasting module can add parameters such as temperature and number of people to achieve more accurate predictions.

References

- [1] Saint John Walker. Big data: A revolution that will transform how we live, work, and think, 2014.
- [2] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.
- [3] Hsinchun Chen, Roger HL Chiang, and Veda C Storey. Business intelligence and analytics: from big data to big impact. *MIS quarterly*, pages 1165–1188, 2012.
- [4] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101, 2012.
- [5] Tom White. *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.”, 2012.
- [6] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
- [7] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2), 2010.
- [8] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the*

- 2017 ACM International Conference on Management of Data*, pages 51–63. ACM, 2017.
- [9] Huang Fang. Managing data lakes in big data era: What’s a data lake and why has it become popular in data management ecosystem. In *Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2015 IEEE International Conference on*, pages 820–824. IEEE, 2015.
- [10] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.
- [11] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *Proceedings of the VLDB Endowment*, 8(12):1654–1655, 2015.
- [12] Rajiv Ranjan. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014.
- [13] Mohiuddin Solaimani, Mohammed Iftekhhar, Latifur Khan, Bhavani Thuraisingham, Joe Ingram, and Sadi Evren Seker. Online anomaly detection for multi-source vmware using a distributed streaming framework. *Software: Practice and Experience*, 46(11):1479–1497, 2016.
- [14] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.
- [15] Liu Chen, Junghyun Ko, and Jeongmo Yeo. Analysis of the influence factors of data loading performance using apache sqoop. *KIPS Transactions on Software and Data Engineering*, 4(2):77–82, 2015.
- [16] Amrit Pal, Kunal Jain, Pinki Agrawal, and Sanjay Agrawal. A performance analysis of mapreduce task with large number of files dataset in big data using

- hadoop. In *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*, pages 587–591. IEEE, 2014.
- [17] Aditya Bhardwaj, Ankit Kumar, Yogendra Narayan, Pawan Kumar, et al. Big data emerging technologies: A casestudy with analyzing twitter data using apache hive. In *Recent Advances in Engineering & Computational Sciences (RAECS), 2015 2nd International Conference on*, pages 1–6. IEEE, 2015.
- [18] Apache HBase Team. Apache hbase reference guide. *Apache, version, 2(0)*, 2016.
- [19] Chao-Tung Yang, Shuo-Tsung Chen, Walter Den, Yun-Ting Wang, and Endah Kristiani. Implementation of an intelligent indoor environmental monitoring and management system in cloud. *Future Generation Computer Systems*, 2018.
- [20] Devadutta Ghat, David Rorke, and Dileep Kumar. New sql benchmarks: Apache impala (incubating) uniquely delivers analytic database performance, 2016.
- [21] Kunal Gupta, Astha Sachdev, and Ashish Sureka. Empirical analysis on comparing the performance of alpha miner algorithm in sql query language and nosql column-oriented databases using apache phoenix. *arXiv preprint arXiv:1703.05481*, 2017.
- [22] Sarathkumar Rangarajan, Huai Liu, Hua Wang, and Chuan-Long Wang. Scalable architecture for personalized healthcare service recommendation using big data lake. In *Service Research and Innovation*, pages 65–79. Springer, 2015.
- [23] Maanak Gupta, Farhan Patwa, James Benson, and Ravi Sandhu. Multi-layer authorization framework for a representative hadoop ecosystem deployment. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pages 183–190. ACM, 2017.

- [24] Pradeeban Kathiravelu and Ashish Sharma. A dynamic data warehousing platform for creating and accessing biomedical data lakes. In *VLDB Workshop on Data Management and Analytics for Medicine and Healthcare*, pages 101–120. Springer, 2016.
- [25] Chen Zhang and Xue Liu. Hbasemq: A distributed message queuing system on clouds with hbase. In *INFOCOM, 2013 Proceedings IEEE*, pages 40–44. IEEE, 2013.
- [26] Yue Wang, Yingzhong Xu, Yue Liu, Jian Chen, and Songlin Hu. Qmapper for smart grid: Migrating sql-based application to hive. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 647–658. ACM, 2015.
- [27] Anja Gruenheid, Edward Omiecinski, and Leo Mark. Query optimization using column statistics in hive. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, pages 97–105. ACM, 2011.
- [28] Chao-Tung Yang, Jung-Chun Liu, Shuo-Tsung Chen, and Hsin-Wen Lu. Implementation of a big data accessing and processing platform for medical records in cloud. *Journal of medical systems*, 41(10):149, 2017.
- [29] Ren-Hao Liu, Chan-Fu Kuo, Chao-Tung Yang, Shuo-Tsung Chen, and Jung-Chun Liu. On construction of an energy monitoring service using big data technology for smart campus. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pages 81–86. IEEE, 2016.
- [30] Fabrizio Carcillo, Andrea Dal Pozzolo, Yann-Aël Le Borgne, Olivier Caelen, Yannis Mazzer, and Gianluca Bontempi. Scarff: a scalable framework for streaming credit card fraud detection with spark. *Information fusion*, 41:182–194, 2018.

Appendix A

Cloudera Manager Installation

Set host

```
$ sudo vim /etc/hostname  
$ sudo vim /etc/hosts
```

Install and set ntp

```
$ sudo apt-get install ntp  
$ sudo ntpdate -s ntp.ubuntu.com pool.ntp.org
```

Download Cloudera

```
$ wget http://archive.cloudera.com/cm5/installer/latest/cloudera-manager-installer.bin
```

Give the access permission

```
$ sudo chmod 775 cloudera-manager-installer.bin
```

Install Cloudera

```
$ sudo ./cloudera-manager-installer.bin
```

Login Cloudera Browser

```
$ http://IP-Address:7180/
```

Appendix B

Kafka Producer for ICEMS

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.codehaus.jettison.json.JSONArray;
import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;
import java.io.IOException;
import java.net.URL;
import java.net.URLConnection;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Properties;
import java.util.Scanner;

public class Producer_PowerData_Minute_HBase {
    public static void main(String[] args) throws JSONException, IOException {

        Properties props = new Properties();
        props.put("bootstrap.servers", "140.128.98.31:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("enable.auto.commit", "false");
        props.put("auto.offset.reset", "earliest");
        Producer<String, String> producer = new KafkaProducer<>(props);
        System.out.println("準備傳送");
```

```
URLConnection connection =
new URL("http://140.128.197.129:8080/rest/buildingMeter/powerUsage/")
.openConnection();
Scanner scanner = new Scanner(connection.getInputStream());

String PowerData = scanner.useDelimiter("\\\\A").next();
System.out.println(PowerData);

JSONArray k;
JSONObject i;

k = new JSONArray(PowerData);
System.out.println("傳送開始");

for (int p = 0; p < k.length(); p++) {

i = k.getJSONObject(p);

long unixSeconds = Long.parseLong(k.getJSONObject(p).getString("time_stamp"));
Date date = new Date(unixSeconds);
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String formattedDate = sdf.format(date);
producer.send(new ProducerRecord<String, String>("PowerData_Minute_HBase",
i.getString("location"), formattedDate + ","
+ i.getString("location") + ","
+ i.getString("KW") + ","
+ i.getString("totalKWH") + ","
+ i.getString("ch1_pf") + ","
+ i.getString("ch1_voltage") + ","
+ i.getString("ch1_current") + ","
+ i.getString("ch1_hz") + ","
+ i.getString("ch2_pf") + ","
+ i.getString("ch2_voltage") + ","
+ i.getString("ch2_current") + ","
+ i.getString("ch2_hz") + ","
+ i.getString("ch3_pf") + ","
+ i.getString("ch3_voltage") + ","
+ i.getString("ch3_current") + ","
+ i.getString("ch3_hz") + ","
+ i.getString("voltage12") + ","
+ i.getString("voltage23") + ","
+ i.getString("voltage31") + ","
+ i.getString("ch1_THDi") + ","
+ i.getString("ch2_THDi") + ","
+ i.getString("ch3_THDi") + ","
+ i.getString("ch1_THDv") + ","
+ i.getString("ch2_THDv") + ",")
```

```
+ i.getString("ch3_THDv") + ","
+ i.getString("total_pf")
));

System.out.println(new ProducerRecord<String, String>("PowerData_Minute_HBase",
    i.getString("location"), formattedDate + ","
+ i.getString("location") + ","
+ i.getString("KW") + ","
+ i.getString("totalKWH") + ","
+ i.getString("ch1_pf") + ","
+ i.getString("ch1_voltage") + ","
+ i.getString("ch1_current") + ","
+ i.getString("ch1_hz") + ","
+ i.getString("ch2_pf") + ","
+ i.getString("ch2_voltage") + ","
+ i.getString("ch2_current") + ","
+ i.getString("ch2_hz") + ","
+ i.getString("ch3_pf") + ","
+ i.getString("ch3_voltage") + ","
+ i.getString("ch3_current") + ","
+ i.getString("ch3_hz") + ","
+ i.getString("voltage12") + ","
+ i.getString("voltage23") + ","
+ i.getString("voltage31") + ","
+ i.getString("ch1_THDi") + ","
+ i.getString("ch2_THDi") + ","
+ i.getString("ch3_THDi") + ","
+ i.getString("ch1_THDv") + ","
+ i.getString("ch2_THDv") + ","
+ i.getString("ch3_THDv") + ","
+ i.getString("total_pf")
));
}

System.out.println("傳送結束");
producer.close();
System.out.println("Message sent successfully");
}
}
```

Appendix C

Kafka Producer for IGEMS

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.codehaus.jettison.json.JSONArray;
import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;

import java.io.IOException;
import java.net.URL;
import java.net.URLConnection;
import java.text.DecimalFormat;
import java.util.Properties;
import java.util.Scanner;

public class Producer_PowerData_CC_HBase {
public static void main(String[] args) throws JSONException, IOException {

Properties props = new Properties();
props.put("bootstrap.servers", "140.128.98.31:9092");
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("enable.auto.commit", "false");
props.put("auto.offset.reset", "earliest");
Producer<String, String> producer = new KafkaProducer<>(props);
System.out.println("準備傳送");
```

```
URLConnection connection =
new URL("http://icems.thu.edu.tw/config/getpower.php").openConnection();
Scanner scanner = new Scanner(connection.getInputStream());

String PowerData_CC = scanner.useDelimiter("\\\\A").next();
//System.out.println(PowerData_CC);

JSONArray k;
JSONObject i;

k = new JSONArray(PowerData_CC);

for (int p = 0; p < k.length() - 1; p++) {

i = k.getJSONObject(p);

if (i.getDouble("11") == 0 || i.getDouble("14") == 0 || i.getDouble("8") == 0) {

double v = 0;

producer.send(new ProducerRecord<String, String>("PowerData_CC_HBase", i.getString("0"),
i.getString("1").substring(0, 19) + ","
+ i.getString("0") + ","
+ v + ","
+ i.getString("8") + ","
+ i.getString("14") + ","
+ i.getString("11")
));

System.out.println(new ProducerRecord<String, String>("PowerData_CC_HBase",
i.getString("0"),
i.getString("1").substring(0, 19) + ","
+ i.getString("0") + ","
+ v + ","
+ i.getString("8") + ","
+ i.getString("14") + ","
+ i.getString("11")
));

} else {

double v = i.getDouble("11") / i.getDouble("14") / i.getDouble("8");
DecimalFormat df = new DecimalFormat("##.00");
v = Double.parseDouble(df.format(v));
```

```
producer.send(new ProducerRecord<String, String>("PowerData_CC_HBase", i.getString("0"),
i.getString("1").substring(0, 19) + "," 間
+ i.getString("0") + ","
+ v + ","
+ i.getString("8") + ","
+ i.getString("14") + ","
+ i.getString("11")
));

System.out.println(new ProducerRecord<String, String>("PowerData_CC_HBase",
i.getString("0"),
i.getString("1").substring(0, 19) + "," //時間
+ i.getString("0") + "," //電表ID
+ v + "," //V值
+ i.getString("8") + "," //I
+ i.getString("14") + "," //PF
+ i.getString("11") //P
));

}
}

int count_data = k.length() - 1;
System.out.println("共傳送了" + count_data + "筆資料");
producer.close();

}
}
```


Appendix D

Spark Streaming Write ICEMS to HBase

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.sql.Row
import org.apache.spark.sql._
import org.apache.hadoop.hbase._
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.HTable
import org.apache.hadoop.hbase.TableName
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util._
import scala.collection.JavaConversions._
import org.apache.hadoop.hbase.io.ImmutableBytesWritable

val ssc = new StreamingContext(sc, Seconds(60))
val topicMap = "PowerData_Minute_HBase".split(":").map((_, 1)).toMap
val zkQuorum = "140.128.98.31:2181";
val group = "test-consumer-group"
val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map(_._2)
val lines_split = lines.map(x => x.split(",")).map(x => {(x(0), x(1), x(2), x(3))})

lines_split.foreachRDD(rdd => {
  rdd.foreachPartition(partitionRecords => {
    val conf = HBaseConfiguration.create();
    //val conf = new HBaseConfiguration
    conf.set("hbase.zookeeper.property.clientPort", "2181")
    conf.set("hbase.zookeeper.quorum", "140.128.98.31")
    val connection = ConnectionFactory.createConnection(conf);
```

```
partitionRecords.foreach(s => {
  val table = connection.getTable(TableName.valueOf("powerdata_minute_hbase"))
  val put = new Put(Bytes.toBytes(s._2.toString + "_" + s._1.toString.substring(0, 16)))
  put.addColumn(Bytes.toBytes("powerdata_minute"), Bytes.toBytes("time")
    , Bytes.toBytes(s._1.toString))
  put.addColumn(Bytes.toBytes("powerdata_minute"), Bytes.toBytes("location")
    , Bytes.toBytes(s._2.toString))
  put.addColumn(Bytes.toBytes("powerdata_minute"), Bytes.toBytes("kw")
    , Bytes.toBytes(s._3.toString))
  put.addColumn(Bytes.toBytes("powerdata_minute"), Bytes.toBytes("totalkwh")
    , Bytes.toBytes(s._4.toString))
  table.put(put)
  table.close()
  println(s._1.toString + ", "+ s._2 + " 寫入HBase")
})
})
})
ssc.start()
ssc.awaitTermination()
```

Appendix E

Spark Streaming Write IGEMS to HBase

```
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.sql.Row
import org.apache.spark.sql._
import org.apache.hadoop.hbase._
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.HTable
import org.apache.hadoop.hbase.TableName
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util._
import scala.collection.JavaConversions._
import org.apache.hadoop.hbase.io.ImmutableBytesWritable

val ssc = new StreamingContext(sc, Seconds(10))
val topicMap = "PowerData_CC_HBase".split(":").map((_, 1)).toMap
val zkQuorum = "140.128.98.31:2181";
val group = "test-consumer-group"
val lines = KafkaUtils.createStream(ssc, zkQuorum, group, topicMap).map(_._2)
val lines_split = lines.map(x => x.split(","))
    .map(x => {(x(0), x(1), x(2), x(3), x(4), x(5))})

lines_split.foreachRDD(rdd => {
  rdd.foreachPartition(partitionRecords => {
    val conf = HBaseConfiguration.create();
    //val conf = new HBaseConfiguration
    conf.set("hbase.zookeeper.property.clientPort", "2181")
```

```
conf.set("hbase.zookeeper.quorum", "140.128.98.31")
val connection = ConnectionFactory.createConnection(conf);
partitionRecords.foreach(s => {
val table = connection.getTable(TableName.valueOf("powerdata_cc_hbase"))
val put = new Put(Bytes.toBytes(s._2.toString + "_" + s._1.toString
.substring(0, 18)+"0"))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("time")
, Bytes.toBytes(s._1.toString))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("meter_id")
, Bytes.toBytes(s._2.toString))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("v")
, Bytes.toBytes(s._3.toString))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("i")
, Bytes.toBytes(s._4.toString))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("pf")
, Bytes.toBytes(s._5.toString))
put.addColumn(Bytes.toBytes("powerdata_cc"), Bytes.toBytes("p")
, Bytes.toBytes(s._6.toString))
table.put(put)
table.close()
println(s._1.toString + "," + s._2 + " 寫入HBase")
})
})
})
ssc.start()
ssc.awaitTermination()
```

Appendix F

Power Forecast Using HoltWinters

```
package Data_Lake
import java.text.SimpleDateFormat
import java.util.{Calendar, Date, Properties}
import com.cloudera.sparkts.models.HoltWinters
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.sql.types._
import org.apache.spark.{SparkConf, SparkContext}

object HoltWintersDate {
def main(args: Array[String]): Unit = {

val conf = new SparkConf().setAppName("Simple Application")
val sc = new SparkContext(conf)
val spark = SparkSession
.builder()
.appName("Spark Hive Example")
.getOrCreate()

val jdbcDF = spark.read.format("jdbc")
.option("url", "jdbc:mysql://120.109.150.175:3306/power")
.option("driver", "com.mysql.jdbc.Driver")
.option("dbtable", "PowerHour").option("user", "hpc")
.option("password", "hpcverygood").load()
jdbcDF.createOrReplaceTempView("PowerHour_test")

def getNowDate(): String = {
var now: Date = new Date()
```

```
var dateFormat: SimpleDateFormat = new SimpleDateFormat("yyyy-MM-dd")
var hehe = dateFormat.format(now)
hehe
}

def determineDayOfTheWeek(a: Int): Int = {
var dayForWeek = 0
if (a == 1) {
dayForWeek = 7
return dayForWeek
} else {
dayForWeek = a - 1
return dayForWeek
}
}

def trainAndPredict(a: String) = {
var test = spark.sql(a)
println("query:" + a)
println("訓練集")
test.show(48)

val dataTrain = test.select("x").rdd.map(r => r(0)).map(_.toString).map(_.toDouble)
.collect()

val ts = Vectors.dense(dataTrain)
val hModel = HoltWinters.fitModel(ts, 24, "Additive", "BOBYQA")
//Multiplicative, Additive
val forecast = hModel.forecast(ts, ts)
val forecastArray = forecast.toArray
println("開始預測")
forecastArray.foreach(println)
println("共預測了" + forecastArray.length + "筆資料")

val writeToMySQLArray = Array.ofDim[String](24, 4)

for (i <- 0 to 23) {
writeToMySQLArray(i)(0) = getNowDate()
writeToMySQLArray(i)(1) = i.toString
writeToMySQLArray(i)(2) = "LIB-4"
writeToMySQLArray(i)(3) = forecastArray(i).toString
}

println("寫入開始...")
val predictRDD = spark.sparkContext.parallelize(writeToMySQLArray)
val schema = StructType(List(StructField("date", StringType, true)
, StructField("hr", IntegerType, true), StructField("Meter_id", StringType, true)
, StructField("P", DoubleType, true)))
val rowRDD = predictRDD
.map(p => Row(p(0).toString, p(1).toInt, p(2).toString, p(3).toDouble))
```

```
val predictDF = spark.createDataFrame(rowRDD, schema)
val prop = new Properties()
prop.put("user", "hpc")
prop.put("password", "hpcverygood")
prop.put("driver", "com.mysql.jdbc.Driver")
predictDF.write.mode("append")
.jdbc("jdbc:mysql://120.109.150.175:3306/power", "power.PowerHourPredict", prop)
println("已寫入24筆資料至MySQL")
}

val sdf = new SimpleDateFormat("yyyy-MM-dd")
val cal = java.util.Calendar.getInstance();
val cal_1 = java.util.Calendar.getInstance();
val cal_2 = java.util.Calendar.getInstance();
val cal_6 = java.util.Calendar.getInstance();
val cal_8 = java.util.Calendar.getInstance();

cal.setTime(sdf.parse(getNowDate()))
cal.add(java.util.Calendar.DATE, -7)

cal_1.setTime(sdf.parse(getNowDate()))
cal_1.add(java.util.Calendar.DATE, -1)

cal_2.setTime(sdf.parse(getNowDate()))
cal_2.add(java.util.Calendar.DATE, -2)

cal_6.setTime(sdf.parse(getNowDate()))
cal_6.add(java.util.Calendar.DATE, -6)

cal_8.setTime(sdf.parse(getNowDate()))
cal_8.add(java.util.Calendar.DATE, -8)

val day = determineDayOfTheWeek(cal.get(Calendar.DAY_OF_WEEK))
println("現在日期:" + getNowDate() + " 星期" + day)
println("上週日期:" + sdf.format(cal.getTime))

day match {
case 1 =>
var sqlDate = "" + sdf.format(cal.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_6.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate + ", " + sqlDate_1 + ")"
trainAndPredict(sqlQuery)

case 2 =>
var sqlDate = "" + sdf.format(cal.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_8.getTime) + ""
```

```
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate_1 + ", " + sqlDate + ")"
trainAndPredict(sqlQuery)
case 3 =>
var sqlDate = "" + sdf.format(cal_1.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_2.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate_1 + ", " + sqlDate + ")"
trainAndPredict(sqlQuery)

case 4 =>
var sqlDate = "" + sdf.format(cal_1.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_2.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate_1 + ", " + sqlDate + ")"
trainAndPredict(sqlQuery)

case 5 =>
var sqlDate = "" + sdf.format(cal_1.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_2.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate_1 + ", " + sqlDate + ")"
trainAndPredict(sqlQuery)

case 6 =>
var sqlDate = "" + sdf.format(cal.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_6.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate + ", " + sqlDate_1 + ")"
trainAndPredict(sqlQuery)

case 7 =>
var sqlDate = "" + sdf.format(cal.getTime) + ""
var sqlDate_1 = "" + sdf.format(cal_8.getTime) + ""
var sqlQuery = "select round(`p`/1000, 1) as x from PowerHour_test
where `Meter_id` = 'LIB-4' and `p`/1000 > 10
and `date` in " + "(" + sqlDate_1 + ", " + sqlDate + ")"
trainAndPredict(sqlQuery)
}
}
}
```


Appendix G

Power Failure Analysis

```
import org.apache.hadoop.hbase._
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.HTable
import org.apache.hadoop.hbase.TableName
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util._
import scala.collection.JavaConversions._
import org.apache.hadoop.hbase.io.ImmutableBytesWritable

val df = sql("SQLQUERY")
val df_1 = sql("SQLQUERY")
val data_RDD = df.rdd
val data_RDD_1 = df_1.rdd
val data = data_RDD.map(_.mkString(",")).take(data_RDD.count().toInt)
.map(x => x.split(","))
val data_1 = data_RDD_1.map(_.mkString(",")).take(data_RDD_1.count().toInt)
.map(x => x.split(","))

import java.text.SimpleDateFormat
import scala.collection.mutable.ArrayBuffer
val data_ArrayBuffer = new ArrayBuffer[Long]()
val data_0_ArrayBuffer = new ArrayBuffer[Long]()
val data_1_ArrayBuffer = new ArrayBuffer[Long]()
// val data_2_ArrayBuffer = new ArrayBuffer[Long]()

for( i <- 0 to data.length-2) {

if(data(i)(0).toLong - data(i+1)(0).toLong > 300
&& data(i)(0).toLong - data(i+1)(0).toLong < 10800){
```

```
val df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = df.format(data(i+1)(0).toLong * 1000L)
data_ArrayBuffer += data(i+1)(0).toLong

data_0_ArrayBuffer += data(i+1)(0).toLong
data_0_ArrayBuffer += data(i)(0).toLong

println(date + "," + data(i+1)(0) + "斷電")
}
}
println("化學系館預估共斷電了" + data_0_ArrayBuffer.length/2+ "次")

for( i <- 0 to data_1.length-2) {

if(data_1(i)(0).toLong - data_1(i+1)(0).toLong > 300
&& data_1(i)(0).toLong - data_1(i+1)(0).toLong < 10800){
val df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = df.format(data_1(i+1)(0).toLong * 1000L)
data_1_ArrayBuffer += data_1(i+1)(0).toLong

println(date + "," + data_1(i+1)(0) + "斷電")
}
}
println("圖書館預估共斷電了" + data_1_ArrayBuffer.length + "次")

val final_data = data_ArrayBuffer.intersect(data_1_ArrayBuffer)

for( i <-0 to final_data.length-1){
val df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
val date = df.format(final_data(i) * 1000L)
val recovery_date =
df.format(data_0_ArrayBuffer(data_0_ArrayBuffer.indexOf(final_data(i))+1) * 1000L)
val total_time = {(data_0_ArrayBuffer(data_0_ArrayBuffer.indexOf(final_data(i))+1))
-data_0_ArrayBuffer(data_0_ArrayBuffer.indexOf(final_data(i)))} / 60

val conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.property.clientPort", "2181")
conf.set("hbase.zookeeper.quorum", "140.128.98.31")
val connection = ConnectionFactory.createConnection(conf);
val table = connection.getTable(TableName.valueOf("powerdata_loss_hbase"))
val put = new Put(Bytes.toBytes(date.toString + "_" + recovery_date.toString
.substring(0, 16)))
put.addColumn(Bytes.toBytes("powerdata_loss"), Bytes.toBytes("date"),
Bytes.toBytes(date.toString))
put.addColumn(Bytes.toBytes("powerdata_loss"), Bytes.toBytes("recovery_date")
, Bytes.toBytes(recovery_date.toString))
put.addColumn(Bytes.toBytes("powerdata_loss"), Bytes.toBytes("total_time"),
Bytes.toBytes(total_time.toString))
```

```
table.put(put)
table.close()
println(date + "," + "發生斷電," + recovery_date + "復電,共斷電了" + total_time +
"分鐘")
}

println("全校預估共斷電了" + final_data.length + "次")
```

