applied sciences

MDPI

Article

Integrated High-Performance Platform for Fast Query Response in Big Data with Hive, Impala, and SparkSQL: A Performance Evaluation

Bao Rong Chang [1], Hsiu-Fen Tsai [2,*] and Yun-Da Lee [1]

1 Department of Computer Science and Information Engineering, National University of Kaohsiung, Kaohsiung 81148, Taiwan; brchang@nuk.edu.tw (B.R.C.); ff830324@gmail.com (Y.-D.L.)
2 Department of Fragrance and Cosmetic Science, Kaohsiung Medical University, Kaohsiung 80708, Taiwan
* Correspondence: sftsai@kmu.edu.tw; Tel.: +886-7-312-1101 (ext. 2366)

Received: 6 August 2018; Accepted: 26 August 2018; Published: 1 September 2018

Abstract: This paper first integrates big data tools—Hive, Impala, and SparkSQL—which support SQL-like queries for rapid data retrieval in big data. The three introduced tools are not only suitable for operating in business intelligence to serve high-performance data retrieval, but they are also an open-source software solution with low cost for small-to-medium enterprise use. In practice, the proposed approach provides an in-memory cache and an in-disk cache to achieve a very fast response to a query if a cache hit occurs. Moreover, this paper develops so-called platform selection that is able to select the appropriate tool dealing with input query with effectiveness and efficiency. As a result, the speed of job execution of proposed approach using platform selection is 2.63 times faster than Hive in the Case 1 experiment, and 4.57 times faster in the Case 2 experiment.

Keywords: big data tool; SQL-like query; in-memory cache; tool selection; performance index

1. Introduction

The storage cost and data acquisition cost have fallen sharply due to technological advances, which has created the rise of big data in this era. Meanwhile, the growth of data volume in various industries around the world is rising rapidly, for example, in social networks [1]. Also, since obtaining data is no longer the biggest difficulty in scientific research, how to “store” and “dig” massive data and successfully “communicate” the results of analysis has become a new bottleneck and research focus. In big data analytics, the characteristics of data processing are high volume, high velocity, high variety, high veracity, high value, and high visualization (abbreviated as 6 V), and hence data processing is facing an expanding number of incredible challenges. Many issues are emerging in big data manipulation such as network traffic flow, high volume storage, remote backup, resources management, computing performance, package compatibility, data security, service level agreement, disaster recovery, and other abilities. Many practical applications concerning big data have to find the proper way for dealing with huge amounts of data containing a complex structure. Unfortunately, traditional tools cannot solve the problem as mentioned above and thus it is worthwhile to continue to develop emerging tools or tools to tackle the issues in big data environment.

Business intelligence (BI) is defined as a kind of technology and application that is made up of a data warehouse (or data mart), query report, data analysis, data mining, data backup, and recovery to help enterprises make decisions [2]. Today, enterprises treat business intelligence as a tool that transforms existing data in the enterprise into knowledge and helps companies make informed business decisions. The data discussed here includes orders from the enterprise business system, inventory, transaction accounts, customer and supplier profiles, and data from the industry and competitors, as well as various data from other external environments in which the business is located. Business decisions that business intelligence can assist with can be either at the operation level or at the management and policy level. People always like to deliver a query to an easy-to-use business intelligence and analytics that can support a full range of SQL-like commands capabilities such as SAS PROC SQL, Microsoft SQL Server, and Tableau SQL. Today's companies need the ability to turn on BI using SQL commands that scale across big data in BI. Advanced big data analytics requests and reshapes BI with the ability to turn complex data to insight and action for better business decision-making. Furthermore, the technique of in-memory computing is an increasingly viable option; a number of companies are using in-memory computing for many analytical applications [3]. That is why in-memory computing for big data analytics has become a core function of BI to run smarter, more agile, and higher-efficiency data retrieval and analysis. The cost of the establishment and maintenance of big data BI are vital concerns because small-to-medium enterprises may not be able to afford it. A change in attitude towards open-source solutions to create an opportunity for business intelligence is expected.

This paper is the first to introduce the open-source solutions to multiple big data tools using (1) Apache Hive [4], (2) Apache Impala [5], and (3) Apache SparkSQL [6] that fully support SQL-like commands. Next, the integration of these tools, as shown in Figure 1, is applied to a data warehouse and they work over underlying supported platforms like Apache Hadoop [7] and/or Apache Spark [8] that commonly share the distributed file system Hadoop Distributed File System(HDFS) [9]. This paper has built three integrated open source SQL-handling tools for the data warehouse part in a commercial business intelligence system, as shown in Figure 2, where a block is surrounded by a dotted square, and thus it is suitable for traditional SQL query interaction. In particular, the proposed approaches can sustain the existing tools relying on relational databases as a data source where the existing data-intensive applications are compatible with a minimum of modifications, or even no modifications at all to the new tool. Finally, this paper has further endeavored to develop a mechanism called platform selection that can choose the appropriate tool to speed up SQL query operation in accordance with the remaining amount of memory instantly available for a computing node. We will deliver complex SQL commands to real world use cases for big data analytics such as OLAP, data mining, and real-time statistics and see how they work.

The following sections of this paper are arranged as follows. In Section 2, related work will be described. The way to tool implementation and performance evaluation are given in Section 3. The experimental results and discussion will be given in Section 4. Finally, we draw a brief conclusion in Section 5.

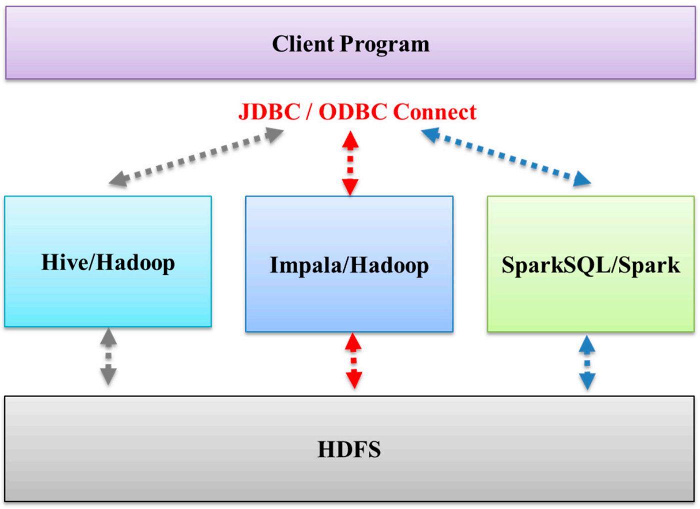


Figure 1. Multiple big data processing tools. JDBC: Java Database Connectivity; ODBC: Open Database Connectivity.

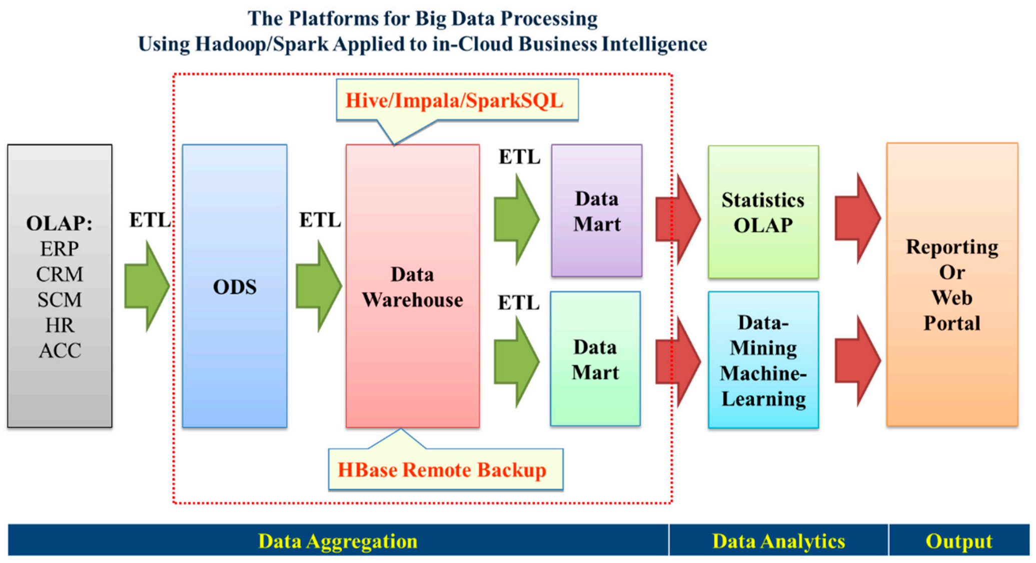


Figure 2. Business intelligence (BI) with multiple big data processing tools.

2. Related Work

2.1. Literature Review and Motivation

Google's MapReduce [10] and the Big Table [11] concepts have been used to implement their own decentralized server infrastructure [12]. In this structure, various network services were developed, such as Google BigQuery, which offers a rented cloud-based high-performance big data processing environment. Amazon has launched DynamoDB [13] services, providing a high-speed NoSQL database storage service for big data. Regarding the issue of SQL-like run-times in big data with NoSQL data sources, the paper [14] defines a lifecycle for big data processing and classifies various available tools and technologies, like Hive and SparkSQL, in terms of the lifecycle phases of big data, which include data acquisition, data storage, data analysis, and data exploitation of the results. In the tutorial [15], this study uses the term SQL-on-Hadoop to refer to systems that provide some level of declarative SQL(-like) processing over HDFS and NoSQL data sources, using architectures that include computational or storage engines compatible with Apache Hadoop. Some SQL-on-Hadoop systems provide their own SQL specific run-times, such as Impala, Big SQL, and Presto, while others exploit a general-purpose run-time such as Hive (MapReduce and Tez) and SparkSQL (Spark). In recent years, academia and industry have started to recognize the limitations of the Hadoop framework and thereafter they have constituted a new wave of mostly domain-specific advanced big data processing tools, such as Impala, SparkSQL, Presto, BigSQL, and Drill, referring to them as Big Data 2.0 processing systems in Reference [16].

In the face of expanding incredible amounts of data in the real world, data-intensive applications always utilize high-performance distributed computers to extract the valuable information from big data. However, the operating system begins to swap data between memory and disk frequently when the amount of data exceeds the size of the allowable memory, and this situation deteriorates the performance significantly, possibly even causing a system crash. A mechanism of an in-memory cache with caching temporal results can relieve the bottleneck of quite frequent data swapping between the memory and disk and is capable of responding with data retrieval within seconds if a cache hit occurs. On the other hand, in-memory computing needs to allocate a large amount of memory to store intermediate data and temporal results that enables a high-speed search, which is suitable for data-intensive applications in particular. Regarding the first issue, this paper introduces two caching approaches, that is, in-memory cache and in-disk cache, for a fast response to high frequency data retrieval when a real-time task is running. As for the second issue, this paper incorporating

SparkSQL and Impala tools into the system aims to provide an in-memory data grid [17] dealing with data-intensive use cases and improving query response significantly, whereas Hive works for batch-mode operation for big data. However, a problem has arisen about how to dispatch an incoming query to an appropriate tool because there are three tools that can handle an SQL command. What we want to do is to make sure a job is done with effectiveness and efficiency. Therefore, this paper discovers critical points that indicate different levels corresponding to the different remaining amounts of memory available in any computing node, builds an integrated system, and finally presents a platform selection dispatching a query to the appropriate tool for execution that prevents a possible system crash due to having insufficient memory left. This is our work advantage and difference from the related work in the literature review.

2.2. Computing Environment

In Figure 3, a cloud computing system capable of high performance, high availability, and high scalability is established for big data processing [18] where the layout shows a server farm at the top layer and storage farm at the bottom layer. As for virtualization in a cloud, a KVM-based virtual machine management (VMM) or hypervisor called Proxmox Virtual Environment (PVE) [19] is employed to deploy a cluster of virtual machines (VMs). PVE can effectively monitor the status of the virtual machine and easily adjust the resource configuration of the virtual machine dynamically [20]. Since the server performance is relative to I/O latency, this cloud needs to improve the efficiency of the access of the hard disk and network and adjust the hardware configuration. Moreover, the cloud has to provide an easy-to-use manner to add/remove a server or storage and select the appropriate server for executing an input query associated with the large amount of data. The speed of reading a huge amount of data is increased by amortizing the I/O delay through a reliable distributed file system like a HDFS is possible. For the hard disk performance improvement, this study used a RAID 5 disk array mode that could improve hard disk read and write speeds, and also guaranteed the security of information. To enhance the network performance, this study used Linux Bonding with Link Aggregation Control Protocol (LACP) to bind two network cards. In such a way, it not only effectively improved network traffic, but also worked normally if there was more than one normal network line where the IPTraf monitors two bound network cards.

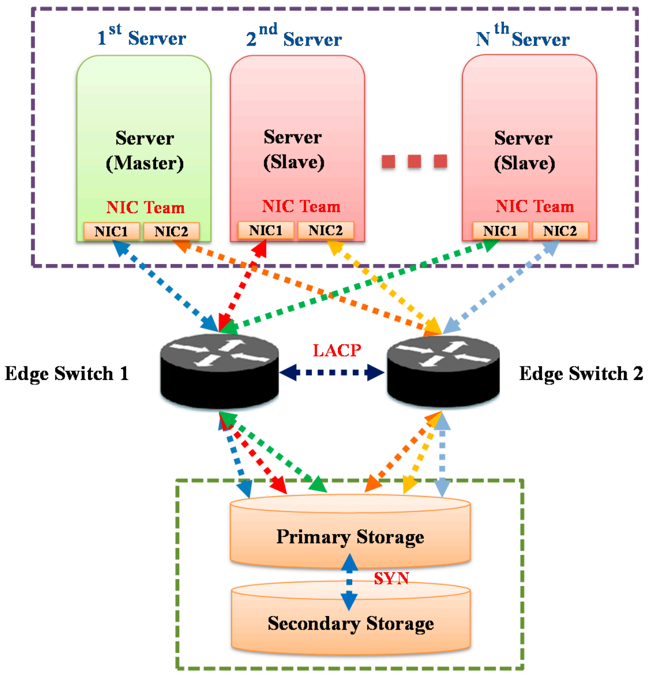
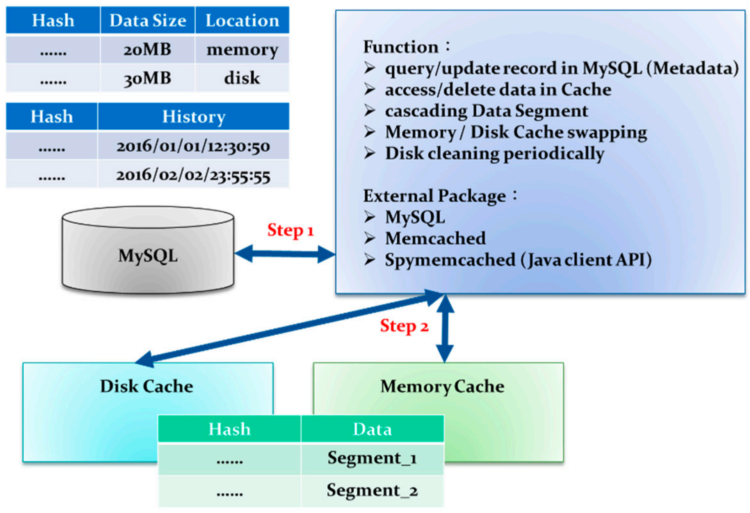


Figure 3. Cloud computing environment.

2.3. Key Computing Components

Hive is a tool of the Apache Hadoop project. When the deployment of Hive system is completed, it will start a HiveServer service to receive and process the client's Hive-QL commands. The client can connect Hive at the Hadoop platform through a command line interface (CLI) or Java database connectivity (JDBC) driver and then it send Hive-QL commands to Hive. After Hive receives a Hive-QL command, the Hive-QL command will be compiled to a Java program, and Hadoop MapReduce executes the job requested from a query. Once a Hive-QL query invokes the table creation, the system will build another Hive's own table (a transfer table) for Hive at the Hadoop platform and creates its metadata at the same time. Metadata will be stored in the relational database MySQL called Metastore [21], and thereafter Hive-QL query is able to get the information in Metastore to look for where data are located and how many data to retrieve, as shown in Figures 4 and 5. Notice that Hive's metastore information is commonly used for Impala and SparkSQL so that Impala/SparkSQL sharing the same Hive's metastore information is individually able to perform the rapid data retrieval from HDFS where the very large amount of data has been stored in HDFS.



Hash	Data Size	Location
……	20MB	memory
……	30MB	disk

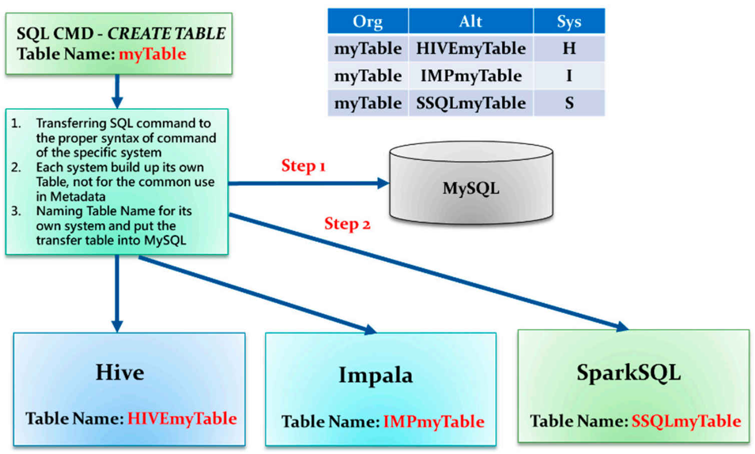
Hash	History
……	2016/01/01/12:30:50
……	2016/02/02/23:55:55

Function：
- query/update record in MySQL (Metadata)
- access/delete data in Cache
- cascading Data Segment
- Memory / Disk Cache swapping
- Disk cleaning periodically

External Package：
- MySQL
- Memcached
- Spymemcached (Java client API)

Hash	Data
……	Segment_1
……	Segment_2

Figure 4. Creating table and storing the information in the metastore.



SQL CMD - *CREATE TABLE*
Table Name: myTable

1. Transferring SQL command to the proper syntax of command of the specific system
2. Each system build up its own Table, not for the common use in Metadata
3. Naming Table Name for its own system and put the transfer table into MySQL

Org	Alt	Sys
myTable	HIVEmyTable	H
myTable	IMPmyTable	I
myTable	SSQLmyTable	S

Figure 5. SQL query command to the tools.

Impala, developed by Cloudera, is used to speed up this kind of SQL-Like query statement; it uses its own MPP (massively parallel processing) query engine. Impala uses LLVM (low level virtual machine, written in C++) [22] to compile these statements. Using an LLVM can significantly reduce the compiling cost. Impala is capable of handling Hive-like SQL commands, and fortunately is able to access Hive's metastore information as well. The main function of Spark is the same as Hadoop MapReduce, but it uses in-memory cluster computing. The storage system is also compatible with HDFS. SparkSQL is a Hive-like system based on Spark MapReduce. It is fully compatible with Hive as well as being capable of accessing Hive's metastore information like metadata stored in the MySQL database.

Hue is a web-based graphical interface [23] for Hadoop and Impala. Hue in Hadoop or Impala is just like phpMyAdmin in MySQL. This study employed Hue as a user-machine interface because users can easily perform operations and observations in addressing the problem of the Hadoop ecosystem's difficulty of operation.

Memcached [24] is a key-value type of distributed memory caching system. The key length is limited to 250 characters and a single datum cannot exceed 1MB. Currently, it is often used in websites and database search result caches. In Figure 6, the system has provided Spymemcached metastore information to locate the data in the memory cache or in the disk cache so that Spymemcached is able to update data between the memory cache and the disk cache asynchronously. This paper intends to implement rapid data retrieval the in-memory cache and the in-disk cache if there is no need to start Hive, Impala, or SparkQSL for dealing with an SQL query.

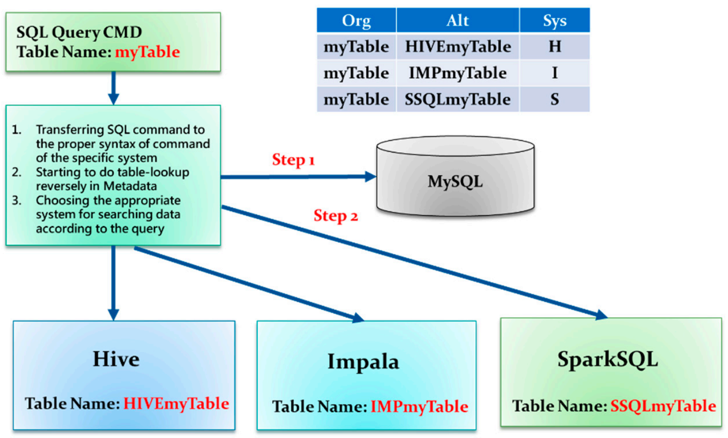

SQL Query CMD
Table Name: myTable
Org | Alt | Sys
myTable | HIVEmyTable | H
myTable | IMPmyTable | I
myTable | SSQLmyTable | S
1. Transferring SQL command to the proper syntax of command of the specific system
2. Starting to do table-lookup reversely in Metadata
3. Choosing the appropriate system for searching data according to the query
Step 1
MySQL
Step 2
Hive
Table Name: HIVEmyTable
Impala
Table Name: IMPmyTable
SparkSQL
Table Name: SSQLmyTable

Figure 6. Hive metastore applied to the in-memory cache and the in-disk cache.

2.4. Critical Point Discovery

The recipe of open-source packages with their respective versions used in this paper is as follows: Hadoop 2.5.0, Hive 0.13.1, Impala 2.1.2, Spark 1.2.0, Hue 3.7.0, Oracle Java (JDK) 7u67, and Scala 2.11.1. There are seven servers in a farm along with Switch Cisco SG200-26 26-port Gigabit Smart Switch × 2 and the specification of each server is IBM System × 3650 M4 with CPU Intel Xeon E5-2620@2.0 GHz × 2, Memory DDR3-8 G × 16, Disk 7200 rpm-300 GB × 8, and Network Intel-1GbEx2. Hive, Impala, and SparkSQL are quite similar in function. They use the same metastore and support for similar SQL syntax. The experiment herein is to find the critical points of performance in these different tools with various amounts of remaining memory. In order to implement the mechanism of automatically selecting the appropriate tool, this experiment will test the execution performances of different SQL commands and different scales of data by adjusting the virtual machine's amount of remaining memory. Test items are listed in Tables 1–3 along with ten different memory allocations from 2 GB to 20 GB increased by 2 GB every time. So there will be totally 270 tests and the number of run experiments is five times for each test. The comparison charts are shown in Figures 7–10. In this experiment, the testing environment is designed by this study. Different results may exist with different hardware tools. This section for discovering critical points intends to find what size of memory where Impala or SparkSQL may not work successfully and it probably enforces the program terminated. As for how big the amount of data that is given that will affect the performance of a tool or cause a computing node crash, it depends on whether a data-intensive application is running on either real-time or batch mode within a big data tool.

The results show that the execution speed of Impala was higher than that of Hive when the remaining amount of memory was between 2 GB and 4 GB. When the remaining amount of memory was between 14 GB and 16 GB, the execution speed of SparkSQL was higher than that of Impala. Therefore, the platform selection appropriately set the remaining amount of memory with 3 GB and 15 GB as critical point 1 (denoted as L1) and critical point 2 (denoted as L2), respectively. Notice that the execution speed of Impala was higher than that of Hive and SparkSQL as the remaining amount of memory was available between 3 GB and 15 GB; the execution speed of SparkSQL was higher than that of Hive and Impala as the remaining amount of memory was bigger than 15 GB, as shown in Figures 7–11. This implied that platform selection should choose Impala for a query as the remaining amount of memory allocated was between L1 and L2, SparkSQL as the remaining amount of memory allocated is over L2, and Hive as the remaining amount of memory allocated was less L1. These critical points marked here were based on the allocated memory size of 20 GB to each computing node (virtual machine), and these settings are not globally applicable because the alternative critical points are subject to changes in the different versions of the software packages and allocated memory size.

Table 1. Test files.

Group	Data Size
Test file I	100 MB—About 400 thousand records
Test file II	500 MB—About two million records
Test file III	1 GB—About 4 million records

Table 2. Test SQL commands.

Category	Function
SQL command I	Only search specific fields
SQL command II	Search for a particular field and add the comparison condition
SQL command III	Execute the commands containing Table JOIN
SQL command IV	Execute the commands containing GROUP BY
SQL command V	Execute the commands containing INSERT INTO

Table 3. Test tools.

Tools	Command
Hive	Use "enforced hive" command to run the tool
Impala	Use "enforced impala" command to run the tool
SparkSQL	Use "enforced sparksql" command to run the tool

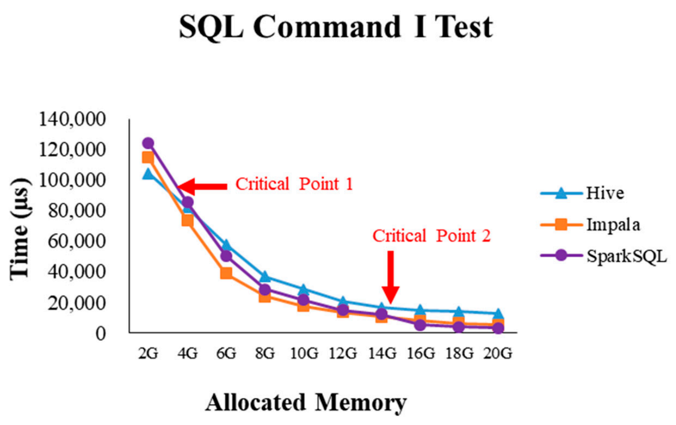


Figure 7. The average execution time of one record by SQL command I.

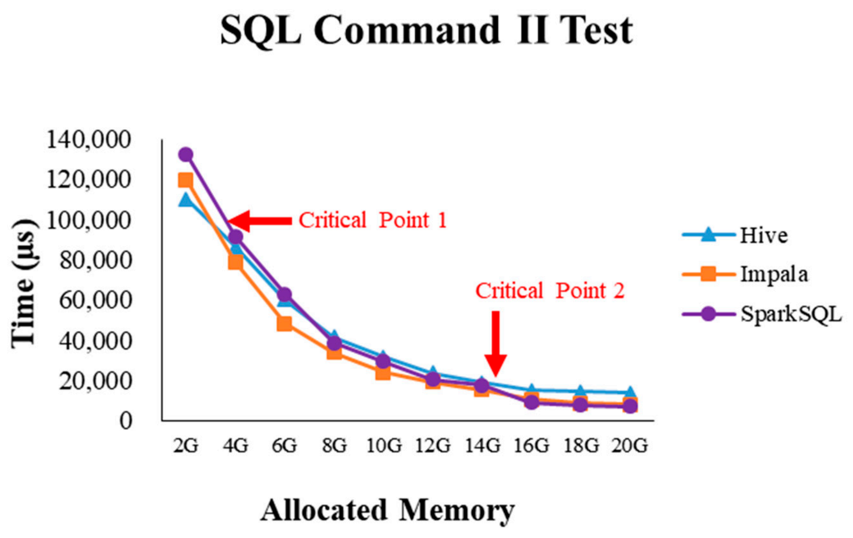


Figure 8. The average execution time of one record by SQL command II.

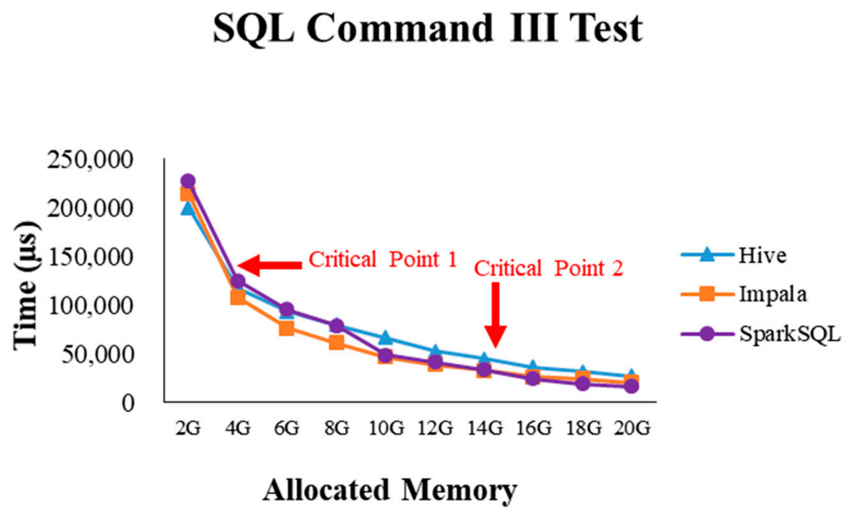


Figure 9. The average execution time of one record by SQL command III.

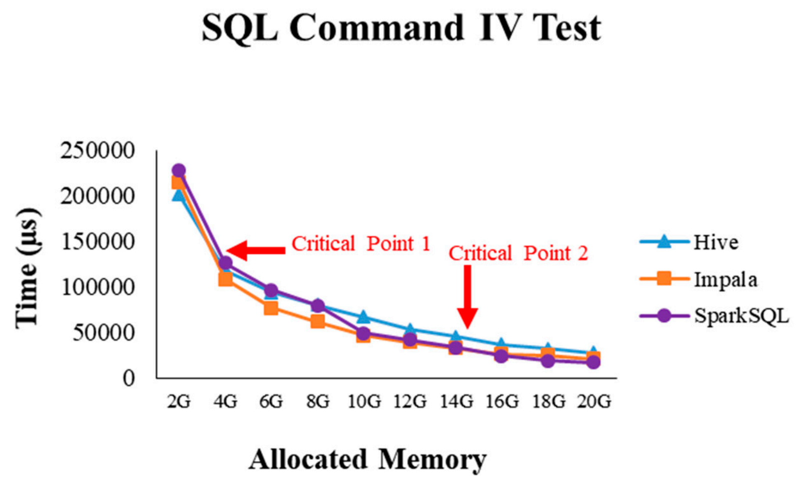


Figure 10. The average execution time of one record by SQL command IV.

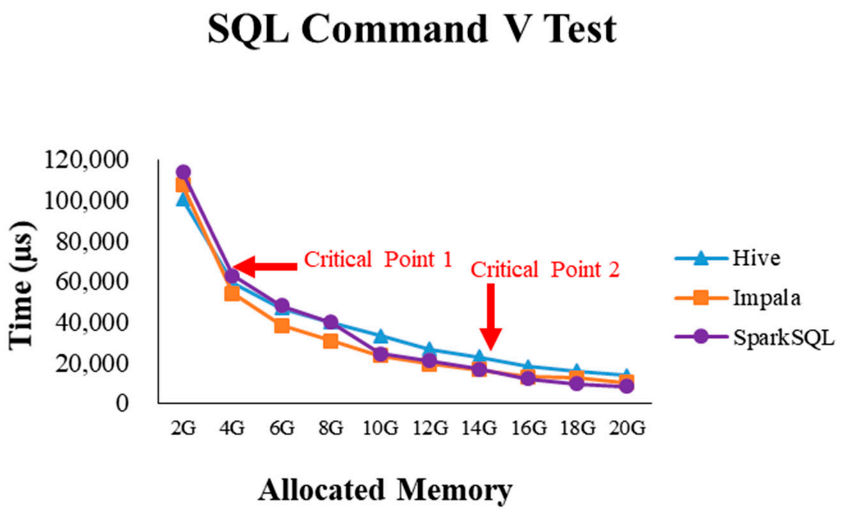


Figure 11. The average execution time of one record by SQL command V.

3. System Implementation and Performance Evaluation

The integration of three big data processing tools—Hive, Impala, and SparkSQL—can be enhanced through the following two ways: (1) the rapid data retrieval from in-memory caching or in-disk caching using Memcached if data have been cached earlier; otherwise (2) platform selection for choosing the appropriate tool to speed up the query/response operation at HDFS. It should provide a metric to indicate the system efficiency and thus a performance index has been introduced in this paper to show the performance evaluation among different approaches.

3.1. Multi-System Compatible Solution

Hive requires relatively few memory resources, but it will write temporary files to HDFS frequently, resulting in long computation times and poor performance. In contrast, SparkSQL needs much more memory because most of the operations are done in memory and uses less write files to hard disk. SparkSQL, however, may risk a routine crash as the memory is not sufficient. On the other hand, Impala supports most standard SQL statements better for more advanced SQL query requirements in business intelligence. Since each tool has their own specialized features, people can only choose a suitable analytics tool for the specific data size or type. Therefore, the use of a multi-system compatible solution is proposed in this paper, as shown in Figure 12, and is able to balance the strengths and weaknesses of the above tools. With small-scale data and real-time requirements, SparkSQL is the best choice, while large-scale data and batch processing requirements will best be met using Apache Hive. If business intelligence or analytical tools have more advanced SQL Query requests, Impala will support most standard SQL statements. Thus, this study will also design a tool selection for this feature. Once the tool selection has received the SQL command, it starts to check the cluster status [25] and the current memory usage among computing nodes, i.e., virtual machines [26]. After that, the tool selection chooses the appropriate tool—Hive, Impala, or SparkSQL—to execute the SQL command.

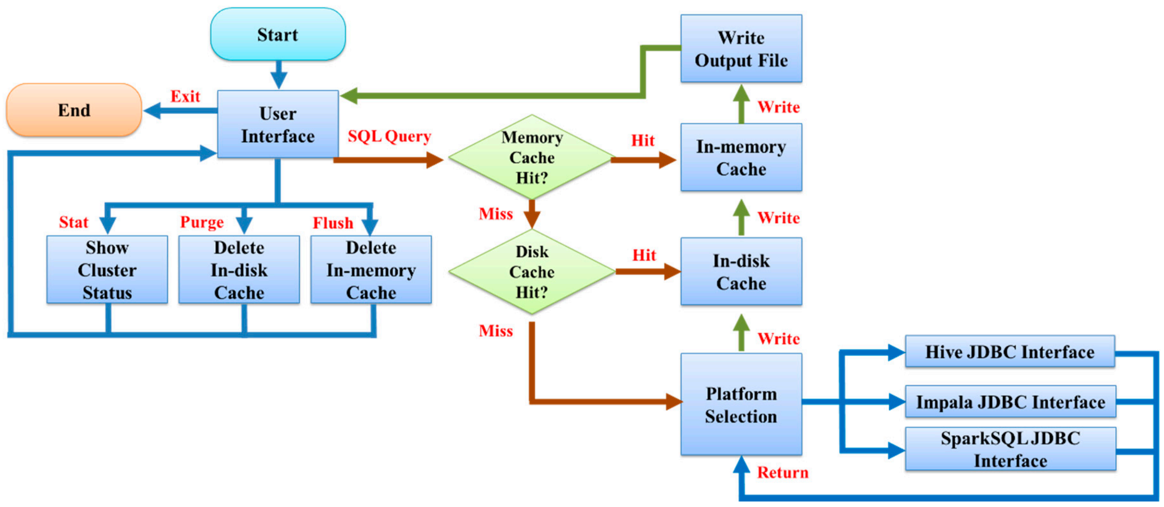


Figure 12. Flowchart of input queries proceeded in multiple big data tools.

3.2. Platform Selection

The integration of big data tools aims to achieve the best performance and efficiency of data retrieval. In order to fulfill this goal, the so-called tool selection implements the appropriate tool to finish a job as soon as possible. Therefore, the incoming query command will be dispatched to the appropriate tool to speed up data retrieval. Hive is a tool supported for SQL command that is executed by MapReduce at Hadoop tool. It must write to disk too many times, resulting in low performance of the query task. To address this problem, Impala uses C++ language to re-write its own high-performance Message Passing Interface (MPI) search engine, dramatically reducing hard disk writing and significantly improving performance. Spark implements in-memory MapReduce technology, ensuring that SparkSQL has a very fast processing speed. Although these three tools are functionally similar, their environment requirements and performances differ. This paper has tested a variety of environments that have different memory capacities for the experimental results. When the remaining memory capacity is 2 GB or less on each server, Impala and SparkSQL had a lot of page swapping, causing extremely low performance or crash that will be shown in Section 4 Experimental results and discussion. When the data scale was larger, it caused the JVM I/O exception and made the program crash. However, when the remaining memory capacity was sufficient, SparkSQL was faster than Hive and Impala. Impala's consumption of memory resources was between those of SparkSQL and Hive. This amount of remaining memory was sufficient for Impala's maximum performance. In the experiment as discovered in Section 2.4, each server was allocated 20 GB of memory allocated to every computing node and set the remaining amount of memory to 3 GB as critical point 1 (denoted as L1) and 15 GB as critical point 2 (denoted as L2). The program automatically selected Hive when the remaining amount of memory was less than L1, Impala between L1 to L2, and SparkSQL when larger than L3. In fact, the critical point values vary with the different conditions of the experimental environment.

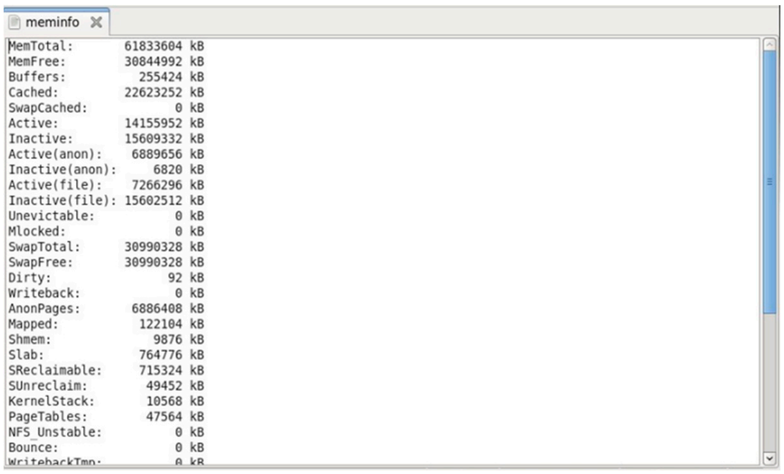
The platform selection algorithm is referred to as PSA, and in PSA, an SQL command is viewed as a user's job forming so-called a job pattern (u, V_C, E_C, f_v, f_e). This paper uses this pattern to develop the automatic platform selection algorithm. Before illustrating the algorithm, we first present the notation it uses. Let u denote the new user's job and u' the proceeding user's job in U, where U stands for the set of user jobs. $f_v(u)$ denotes the amount of the remaining memory unused in each node (virtual machine) when the job u is presented. $f_e(u')$ denotes how long the rest of execution time to finish the job u' will take, where the predicted execution time for a certain amount of data initially has been made according to the experience and then $f_e(u')$ can be estimated by deducting the time taken currently from the predicted execution time. The value of $f_e(u')$ could be an integer k or ∞. V_C denotes nodes $v_1, v_2, \ldots$, and v_i in the cluster, and E_C denotes platforms $e_1, e_2, \ldots$, and e_j in the cluster. A job pattern is defined as (u, V_C, E_C, f_v, f_e) and an incoming job has been stored in the command list where L_C denotes the command list. Once a job has been assigned to the appropriate platform for the execution, this directly puts it into the job queue. In contrast, if the job has not been assigned to any platform, it is put into the job buffer to wait a while. The system is capable of predicting the rest of the execution time to terminate the running specific platform e_j, ($f_e(u')$, e_j, v_i) and calculating the low bound of the execution time T_l to compare with ($f_e(u')$, e_j, v_i). This study uses to obtain a target server's "/proc/meminfo" file. Reading the content of the retrieved meminfo file is to recognize the remaining amount of memory left and can be compared with the least memory amount of e_j, M_j. The platform selection algorithm is shown as Algorithm 1 below.

Algorithm 1. Platform selection algorithm (PSA)

```
     INPUT: Job $u$ from command list $L_C$.
     OUTPUT: Appropriate platform assigned set $assign(u, e_j)$.
1.   Calculate $f_v(u')$ the remaining amount of memory for each node $v_i \in V_C$ (virtual machines) in
     the cluster;
2.   for each platform $e_j \in E_C$, each node $v_i \in V_C$ do
3.   Calculate each platform ($f_e(u')$, $e_j$, $v_i$) the rest of execution time to terminate at each node $v_i \in V_C$;
4.   for each $e_j \in E_C$ do
5.   $assign(u, e_j)$: = {$v_i$ | $v_i \in V$, ($f_e(u')$, $e_j$, $v_i$) $\leq$ the low bound of execution time $T_l$, and ($f_v$ ($u'$), $e_j$, $v_i$) $\geq$ the
     least memory amount of $e_j$, $M_j$};
6.   otherwise, $deassign(u, e_j)$: = {$v_i$ | $v_i \in V$, ($f_e(u')$, $e_j$, $v_i$) > the low bound of execution time $T_l$, or ($f_v$ ($u'$), $e_j$, $v_i$)
     < the least memory amount of $e_j$, $M_j$};
7.   while ($u \in U$ with $assign(u, e_j) \neq \emptyset \vee$ buffer $B_C \neq \emptyset$) do
8.   for each $e_j \in E_C$, $v_i \in V_C$, allocate the necessary memory amount to $assign(u, e_j)$ do;
9.   if queue is not full, dequeue $B_C$: $B_C \backslash assign(u', e_j) \wedge$ enqueue $S_i$: = $S_i \cup assign(u', e_j)$;
10.  elseif queue is not full, enqueue $S_i$: = $S_i \cup$assign($u$, $e_j$);
11.  elseif queue is full, enqueuer $B_C$: = $B_C \cup assign(u, e_j)$;
12.  while ($u \in U$ with $deassign(u, e_j) \neq \emptyset$) do
13.  for each $e_j \in E_C$, $v_i \in V_C$, $u$: = $deassign(u, e_j)$;
14.  enqueue $L_C$: = $L_C \cup u$;
15.  return $assign(u, e_j)$
```

3.3. In-Memory Cache and In-Disk Cache

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids. Nagios provides enterprise-class Open Source information technology monitoring, network monitoring, and server and applications monitoring. Ganglia or Nagios is open-source software and is utilized to monitor server farm resources and obtain related information, but extra hardware resources are required to run the program. This study used SSH to obtain a target server's "/proc/meminfo" file. The meminfo is a virtual file in Linux, so it must be read by a cat command. The content of the retrieved meminfo file is shown in Figure 13. By combining MemTotal and MemFree, it could generate the necessary information required by the platform selection. This function was executed once before starting a selection. It was also executed while the user input a stat command in the program interface. A caching mechanism was more necessary in some environments that have many duplicate SQL commands. Every search query will require time and resources. These can be reduced while the cache hits. Therefore, this study designed a high-speed in-memory cache and a large-capacity in-disk cache and the flowchart is shown in Figure 12. This study used Memcached to develop the proposed in-memory cache to yield very highly-efficient data retrieval but it encountered the capacity constraints. In contrast, the idea was to employ an HDFS distributed file system for the in-disk cache because it saved the search results as a text file and uploaded it to the specified folder in HDFS. According to least recently used (LRU) cache-replacement policy, out-of-date data in the memory cache was deleted to maintain the limit size of memory for a cache. The advantage was that you can reduce the amount of memory used, keep only certain "hot stuff" stored in a cache, put other less-popular, infrequently accessed data in the database, and did not write a copy to cache until the next same request to the database occurred. Clearly, some new data could be added in the database, but data did not instantly become stored in a cache.


meminfo

```
MemTotal:        61833604 kB
MemFree:         30844992 kB
Buffers:           255424 kB
Cached:          22623252 kB
SwapCached:             0 kB
Active:          14155952 kB
Inactive:        15609332 kB
Active(anon):     6889656 kB
Inactive(anon):      6820 kB
Active(file):     7266296 kB
Inactive(file):  15602512 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:       30990328 kB
SwapFree:        30990328 kB
Dirty:                 92 kB
Writeback:              0 kB
AnonPages:        6886408 kB
Mapped:            122104 kB
Shmem:               9876 kB
Slab:              764776 kB
SReclaimable:      715324 kB
SUnreclaim:         49452 kB
KernelStack:        10568 kB
PageTables:         47564 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
```

Figure 13. The meminfo file in Linux.

The mechanism of the in-memory cache together with the in-disk cache was composed of six algorithms to deal with (1) deleting out-of-date data in the memory cache as shown in Algorithm 2, (2) deleting out-of-date file in the disk cache as shown in Algorithm 3, (3) add up-to-date data to the memory cache as shown in Algorithm 4, (4) add up-to-date data to the disk cache as shown in Algorithm 5, (5) visit current data (hit or miss) in the memory cache as shown in Algorithm 6, and (6) visit current data (hit or miss) in the disk cache as shown in Algorithm 7. First, in order to illustrate the algorithm fluently, we defined the notation it used. Let H_m and H_d denote the memory cache and disk cache, respectively. Moreover, H_{mr} and H_{dr} denote the memory cache record and disk cache record, respectively. F_{Hm} denotes a flag where a zero value will conduct the clearing operation periodically to remove the stale data in the memory cache; otherwise a nonzero value will delete the longest one stored in the memory cache. For disk cache F_{Hd} denotes a flag that does the similar operation like the preceding F_{Hm}. The function $\mathrm{T_s}$ is used to show the time stamp and the flag $\mathrm{T_c}$ presents the current time. H_{mfs} and H_{dfs} denotes the total of free space in the memory cache and disk cache, respectively. D_i represents the data in the memory cache, and D_i^* stands for the specific data. Likewise, f_j represents a data file in the disk cache, and f_j^* stands for the specific data file. Et denotes the longest time period for data stored in the cache.

Because each block of in-memory cache has timeliness, Memcached checking the information is timely will automatically replace the cache by an LRU cache-replacement policy when the system is full or the cache has been stored for more than one month. With the implementation of the LRU policy for the in-disk cache, the history list has executed this policy in this program. This list records the location and the last access time to this cache. When a cache hits, the program updates the list once. Checking this list, the program can find some cached data that had not been accessed for a long time and will delete that instantly. The user, through the interface of this program, can enter the purge x command to delete cached data which had not been accessed during a certain x days.

Algorithm 2. Delete out-of-date data in the memory cache

```
Input: $F_{Hm}$, $H_m$, $H_{mr}$.
Output: Updated memory cache record $H_{mr}$.
1 **if** $F_{Hm}$ is zero **then**
2 **while** $H_{mfs} \neq \emptyset$ **do**
3 **if** $T_s(D_i^*) - T_c \geq Et$, $\forall D_i$ **then delete** $D_i^*$ from $H_m$, **update** $H_{mr}$;
  **set** $F_{Hd}$ to nonzero and **call** Algorithm 5 by passing $e_i^*$;
4 **if** $F_{Hm}$ is nonzero **then**
5 **while** $H_{mfs} = \emptyset$ **do**
6 **if** $T_s(D_i^*) - T_c \geq Max(T_s(D_i) - T_c)$, $\forall e_i$ **then delete** $D_i^*$ from $H_m$, **update** $H_{mr}$;
  **set** $F_{Hd}$ to nonzero and **call** Algorithm 5 by passing $D_i^*$;
7 **compute** the rest of cache space for free $H_{mfs}$; **write** $H_{mfs}$ into $H_{mr}$
8 **reset** $F_{Hm}$ to zero;
9 **return** $H_{mr}$.
```

Algorithm 3. Delete out-of-date file in the disk cache

```
Input: $F_{Hd}$, $H_d$, $H_{dr}$.
Output: Updated disk cache record $H_{dr}$.
1 **if** $F_{Hd}$ is zero **then**
2 **while** $H_{dfs} \neq \emptyset$ **do**
3 **if** $T_s(f_j^*) - T_c \geq Et$, $\forall f_j$ **then delete** $f_j^*$ from $H_d$, **update** $H_{dr}$;
4 **if** $F_{Hd}$ is nonzero **then**
5 **while** $H_{dfs} = \emptyset$ **do**
6 **if** $T_s(f_j^*) - T_c \geq Max(T_s(f_j) - T_c)$, $\forall f_j$ **then delete** $f_j^*$ from $H_d$, **update** $H_{dr}$;
7 **compute** the rest of cache space for free $H_{dfs}$; **write** $H_{dfs}$ into $H_{dr}$
8 **reset** $F_{Hd}$ to zero;
9 **return** $H_{dr}$.
```

Algorithm 4. Add up-to-date data to the memory cache

```
Input: $F_{Hm}$, $H_m$, $H_{mr}$.
Output: Updated memory cache record $H_{mr}$.
1 **while** $H_{mfs} = \emptyset$ **do**
2 **set** $F_{Hm}$ to zero and **call** Algorithm 2;
3 **while** $H_{mfs} \neq \emptyset$ **do**
4 **if** $N(D_i^+) < H_{mfs}$, **then add** $D_i^+$ to $H_m$, $\forall D_i^+$, **update** $H_{mr}$;
5 **if** $N(D_i^+) \geq H_{mfs}$, **then set** $F_{Hm}$ to nonzero and **call** Algorithm 2;
6 **add** $D_i^+$ to $H_m$, $\forall D_i^+$, **update** $H_{mr}$;
7 **compute** the rest of cache space for free $H_{mfs}$; **write** $H_{mfs}$ into $H_{mr}$
8 **reset** $F_{Hm}$ to zero;
9 **return** $H_{mr}$.
```

Algorithm 5. Add up-to-date data to the disk cache

```
Input: $F_{Hd}$, $H_d$, $H_{dr}$.
Output: Updated memory cache record $H_{dr}$.
1  while $H_{dfs} = \emptyset$ do
2  set $F_{Hd}$ to zero and call Algorithm 3;
3  while $H_{dfs} \neq \emptyset$ do
4  if $N(D_i^+) < H_{dfs}$, then add $D_i^+$ to $H_d$, $\forall D_i^+$, update $H_{dr}$;
5  if $N(D_i^+) \geq H_{dfs}$, then set $F_{Hd}$ to nonzero and call Algorithm 3;
6  add $D_i^+$ to $H_d$, $\forall D_i^+$, update $H_{dr}$;
7  compute the rest of cache space for free $H_{dfs}$; write $H_{dfs}$ into $H_{dr}$
8  reset $F_{Hd}$ to zero;
9  return $H_{mr}$.
```

The search results are simultaneously stored in-memory and in-disk, but the in-memory cache has the highest access priority. When the in-memory cache hits, the result will not be retrieved by the in-disk cache and big data tool. When the in-memory cache misses, the result will be retrieved through the in-disk cache and will be automatically written back into the in-memory cache. When both in-memory and in-disk caches have missed, the result will be retrieved by big data tools and written to the in-memory and in-disk caches.

Algorithm 6. Visit current data (hit or miss) in the memory cache

```
Input: $D_i^+$ $H_m$, $H_{mr}$.
Output: result.
1  while $H_{mfs} = H_m$ do
2  call Algorithm 7;
3  while $H_{mfs} < H_m$ do
4  if (find $D_i^+$ in $H_m$, $\forall D_i^+$) is true, then update memory_hit_ratio and write information to result;
5  if (find $D_i^+$ in $H_m$, $\forall D_i^+$) is false, then update memory_miss_ratio and call Algorithm 7;
6  return result.
```

Algorithm 7. Visit current data (hit or miss) in the disk cache

```
Input: $D_i^+$ $H_d$, $H_{dr}$.
Output: result.
1  while $H_{dfs} = H_d$ do
2  call Algorithm 1;
3  while $H_{dfs} < H_d$ do
4  if (find $D_i^+$ in $H_d$, $\forall D_i^+$) is true, then update disk_hit_ratio and write information to result;
5  if (find $D_i^+$ in $H_d$, $\forall D_i^+$) is false, then update disk_miss_ratio, call Algorithm 1, and result: = execute
   ($assign(u, e_j)$);
6  return result.
```

3.4. Execution Procedure

This study designed a user interface, a command line interface (CLI), and has designated six kinds of system instructions for the purpose of controlling and monitoring the system at any time. All instructions of the system are listed in Table 4. Users can access all functions of the system through this interface, such as an SQL command that is input via this interface. There are over 45 Java programming routines have been coded to accomplish the optimization of the integration of three proposed big data tools and a UML diagram of the code architecture is shown in Figure 14.

Figure 12 shows the flow chart of a SQL query execution. In Figure 12, the administrator uses the user interface to monitor the status of each node in the server farm. Platform selection started to check the remaining amount of memory available instantly on every computing node, and then it chose the appropriate tool to normally proceed a dispatched job. The proposed approach platform selection was denoted as PS in the experiments. Once a job was finished, the results were stored back in HDFS and the whole process was terminated. This work presents a kind of first-come-first-serve (FCFS) manner to proceed with SQL queries. FCFS was applied to dispatch a single job to a single tool—Hive, Impala, or SparkSQL—denoted as FCFS-SP-Hive, FCFS-SP-Impala, or FCFS-SP-SparkSQL, respectively, in the experiments. Similarly, platform selection with FCFS, denoted as FCFS-PS, was also applied in the experiments where the actions in platform selection checked the remaining amount of memory in every single computing node.

Table 4. A list of instructions for a command line interface in system.

Command	Function
sql [sql_cmd]	Enter a SQL query command
stat	Displays the current cluster status
flush	Clear all in-memory cache
purge [days]	Clear part of cache that are not accessed in a number of days
display [on | off]	Turn on/off to display search results
enforced [name | auto]	Forced to use a specific tool (Hive, Impala, or SparkSQL)

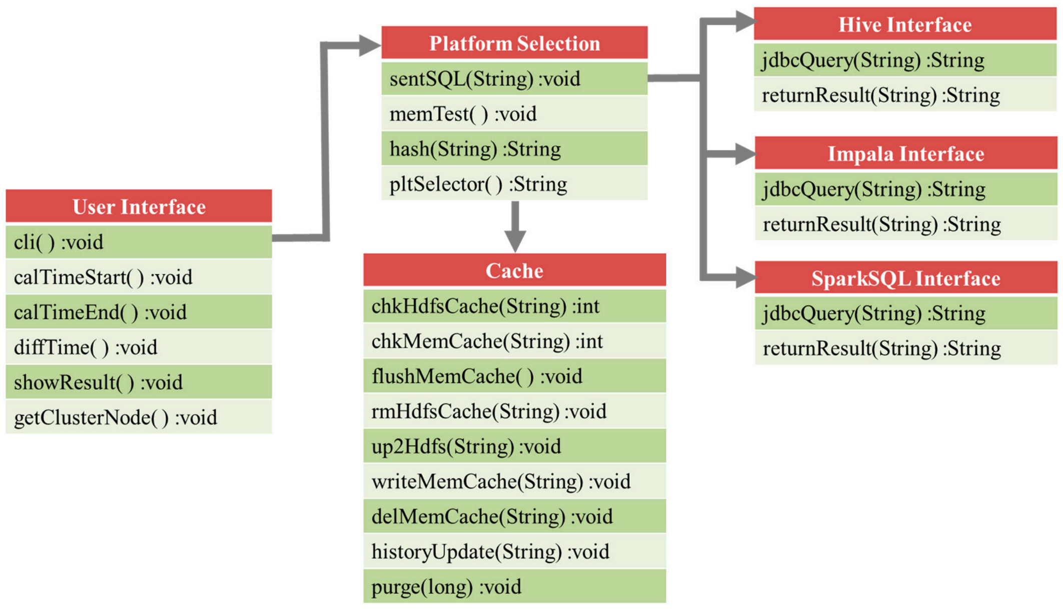


Figure 14. UML diagram of the code architecture.

3.5. Performance Index

In order to evaluate the performance of the several approaches proposed in this paper, the performance index (PI) [27] was derived from first measuring an average access time in any specified environment with a variety of commands for a certain approach using Equation (1), next calculating weighted average access time in any specified environment with various data size in the files using Equation (2), then inducing a normalized performance index according to all of environments among the approaches using Equation (3), and finally resulting in the performance index according to different tests using Equation (4). In these equations, the subscript s indicates the index of the testing SQL command, the subscript i denotes the index of the testing file, j denotes the index of the testing environment, and k denotes the index of the testing approach. Equation (1) calculates the average access time AAT_{ijk} for each testing file with various commands. In Equation (1), AAT_{sijk} represents the average access time for a certain SQL command and a certain data file, and $N_{command}$ represents the number of SQL commands we applied. Equation (2) calculates the average access times $\overline{AAT_{jk}}$ over the different data files, in which AAT_{ijk} represents average access time for a certain data file, as given in Equation (1). Equation (3) calculates the normalized performance index for a specific testing environment. Equation (4) calculates the performance index (PI) for a specific approach, SF_1 is a scaling factor that aims to quantify the value for observation.

$$AAT_{ijk} = \frac{\sum_{s=1}^{r} AAT_{sijk}}{N_{command}},\ where\ s = 1,2,\ldots,r,\ i = 1,2,\ldots,l,\ j = 1,2,\ldots,m,\ k = 1,2,\ldots,n \tag{1}$$

$$\overline{AAT_{jk}} = \sum_{i=1}^{l} !_i \cdot AAT_{ijk},\ where\ j = 1,2,\ldots,m,\ k = 1,2,\ldots,n,\ \sum_{i=1}^{l} !_i = 1 \tag{2}$$

$$\overline{PI_{jk}} = \frac{\frac{1}{\overline{AAT_{jk}}}}{\underset{h=1,2,\ldots,m}{MAX}\left(\frac{1}{\overline{AAT_{hk}}}\right)},\ where\ j = 1,2,\ldots,m,\ k = 1,2,\ldots,n \tag{3}$$

$$PI_k = \left(\sum_{k=1}^{n} W_j \cdot \overline{PI_{jk}}\right) \cdot SF_1,\ where\ j = 1,2,\ldots,m,\ k = 1,2,\ldots,n,\ SF_1 = 10^2,\ \sum_{k=1}^{n} W_k = 1 \tag{4}$$

4. Experimental Results and Discussion

The experiments have considered several facts, such as SQL command complexity, data size, and computing resources, which can influence the approach performance significantly. Regarding the system efficiency, the performance index of each approach could be obtained when the weighted average execution time had been evaluated. This section introduces two cases, namely Case 1 and Case 2, involved in the experiments. In short, Case 1 employed the computer simulation to make the proposed one proof-of-concept a success of examples for engineering computing, and Case 2 adopted the real-world use cases to show proof-of-use in the area of scientific research.

4.1. Experimental Environment in Case 1

The test used the virtual machine's dynamic resource adjustments to create experimental environments that have different amounts of remaining memory. Experimental environments were arranged as listed in Table 5. Each approach was compared by executing different sizes of test data and different complexity of SQL commands. We have downloaded several plain text books from a website [28] and have piled up them to be different data sizes for the test. Test data, which have four fields, were randomly generated by the program. The first column was a unique key string and the other 26 columns had saved words according to their acronym from A/a to Z/z, respectively. The scale of test data is listed in Table 6. Test SQL commands are designated as listed in Table 7. The test method is listed in Table 8.

Table 5. Test environment.

Test Environment	**Memory Size**
Test environment I	Reserve 3 GB remaining memory space
Test environment II	Reserve 10 GB remaining memory space
Test environment III	Allocate all memory space 20 GB

Table 6. Test data set.

No.	Data Size	Codename
1	850 GB	A
2	30 GB	B
3	400 GB	C
4	10 GB	D
5	500 GB	E
6	630 GB	F
7	1 TB	G
8	20 GB	H
9	100 GB	I
10	700 GB	J

Table 7. Test SQL command.

Category	Function
SQL command I	Only search specific fields e.g., SELECT key FROM test10g ORDER BY val1;
SQL command II	Add the comparison condition e.g., SELECT key FROM test10g WHERE val1 + val2 = 250;
SQL command III	Execute the commands containing Table JOIN e.g., SELECT test100g.val1, test100g.val2, test10g.val3 FROM test100g, test10g WHERE test100g.val2 = test10g.val2;
SQL command IV	Execute the commands containing GROUP BY e.g., SELECT val1, count(val1) FROM test100g GROUP BY val1;
SQL command V	Execute the commands containing INSERT INTO e.g., INSERT INTO test10g (key, val1, val2, val3) values ('key69', '85', '163', '69');

Table 8. Test method.

Method	Command
Hive	Use "enforced hive" command to run the tool
Impala	Use "enforced impala" command to run the tool
SparkSQL	Use "enforced sparksql" command to run the tool
In-memory Cache	Reaction time when cache hit
In-disk Cache	Reaction time when cache hit

4.2. Experimental Results in Case 1

These experimental results are the various experimental environments that executed different sizes of data and different SQL commands. For all of the experiments in Case 1, the response with a query for In-memory Cache was always within 5 s, and for In-disk Cache, it was between 5 and 10 s. Therefore, the following experimental results were given by the other methods.

4.2.1. Test Environment I

The experimental results data are shown in Figures 15–19. The experimental results show that Hive could still complete the job in the status that lacked memory, but Impala and SparkSQL crashed in reaction to some scales of data size.

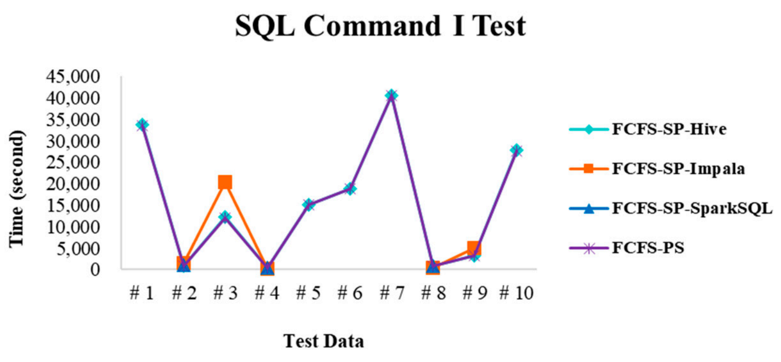


Figure 15. Execution time of SQL command I in test environment I.

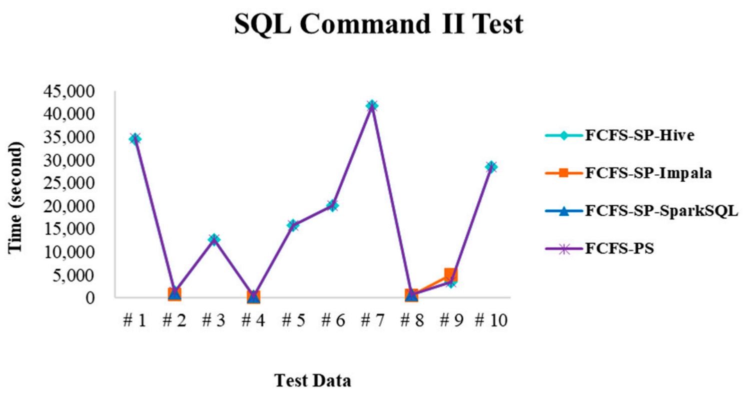


Figure 16. Execution time of SQL command II in test environment I.

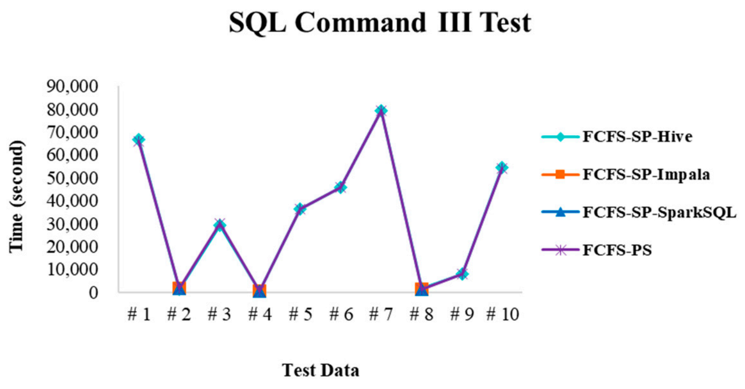


Figure 17. Execution time of SQL command III in test environment I.

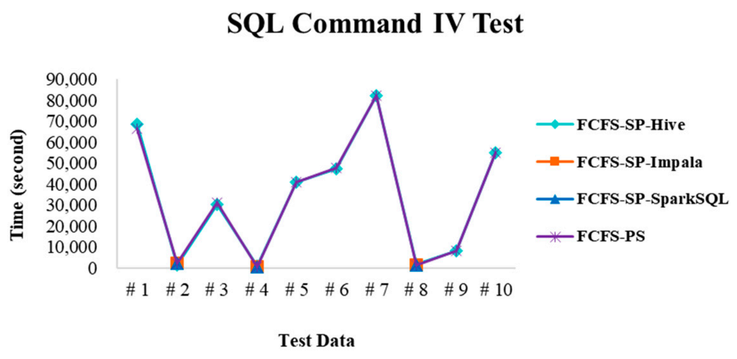


Figure 18. Execution time of SQL command IV in test environment I.

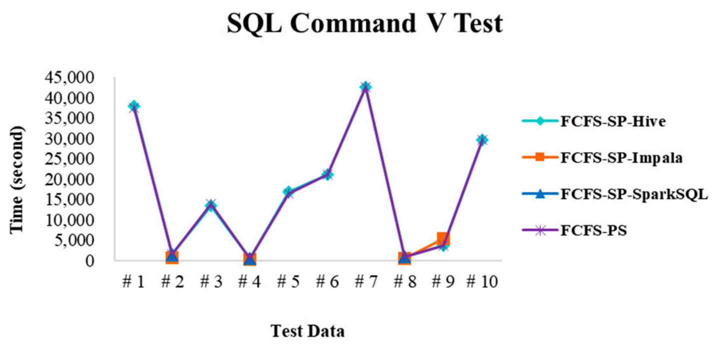


Figure 19. Execution time of SQL command V in test environment I.

4.2.2. Test Environment II

The experimental results data are shown in Figures 20–24. The experimental results show that Impala's performance was more prominent with medium amounts of memory.

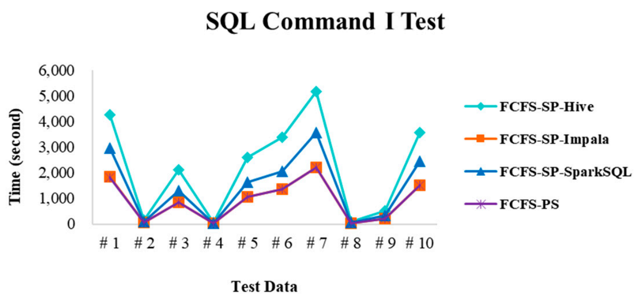


Figure 20. Execution time of SQL command I in test environment II.

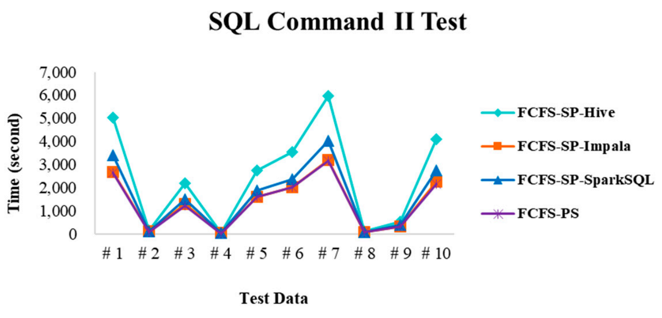


Figure 21. Execution time of SQL command II in test environment II.

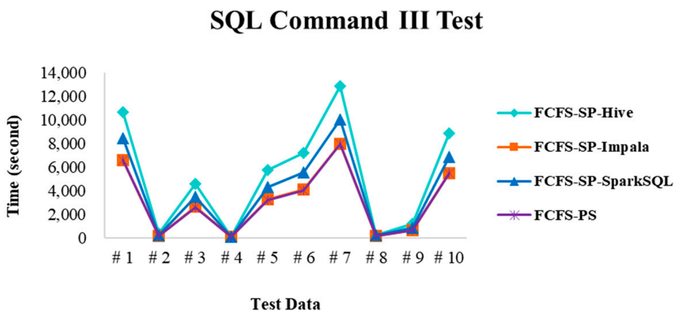


Figure 22. Execution time of SQL command III in test environment II.

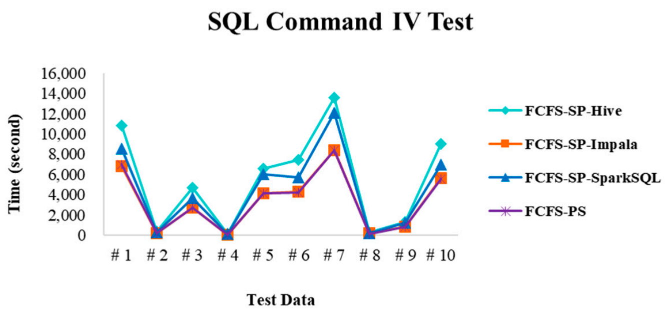


Figure 23. Execution time of SQL command IV in test environment II.

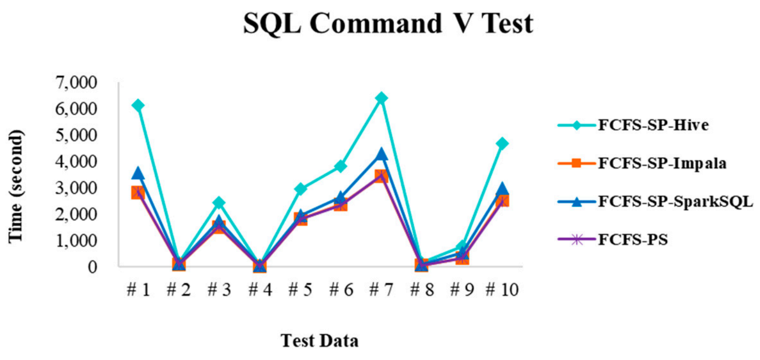


Figure 24. Execution time of SQL command V in test environment II.

4.2.3. Test Environment III

The experimental results data are shown in Figures 25–29. The experimental results show that SparkSQL's performance was excellent with more adequate amounts of memory.

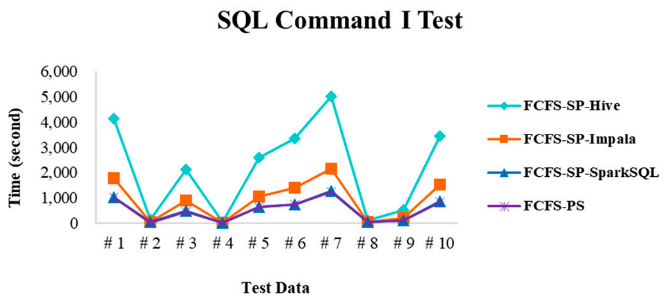


Figure 25. Execution time of SQL command I in test environment III.

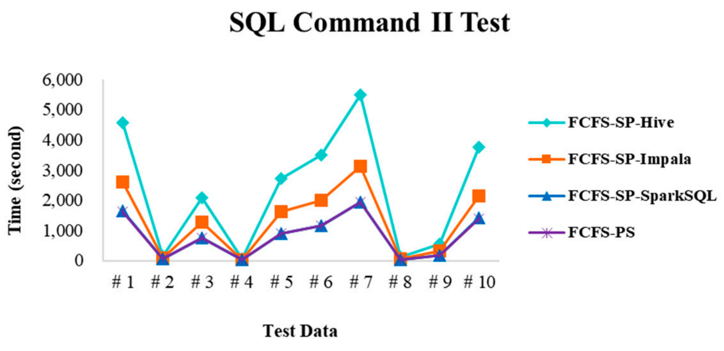


Figure 26. Execution time of SQL command II in test environment III.

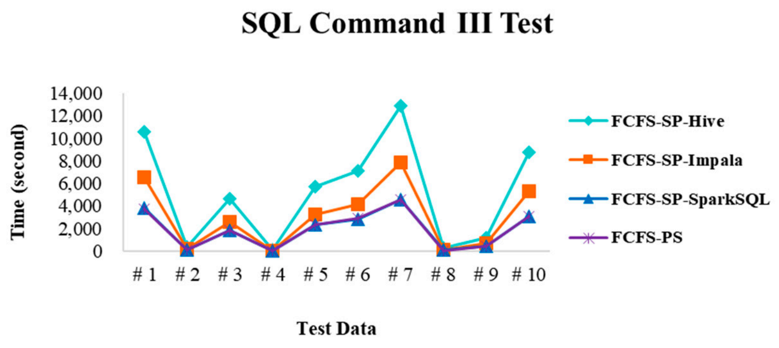


Figure 27. Execution time of SQL command III in test environment III.

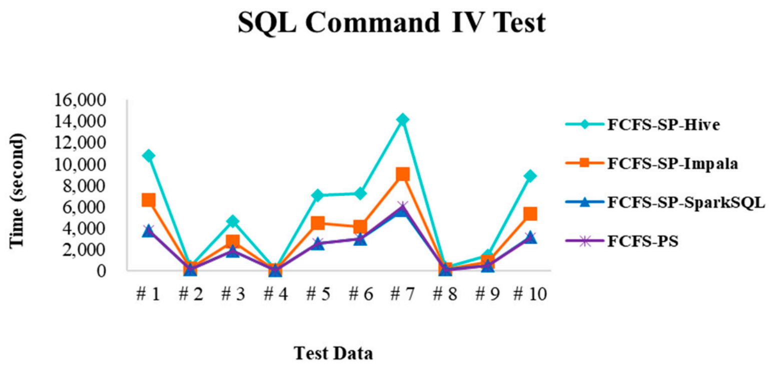


Figure 28. Execution time of SQL command IV in test environment III.

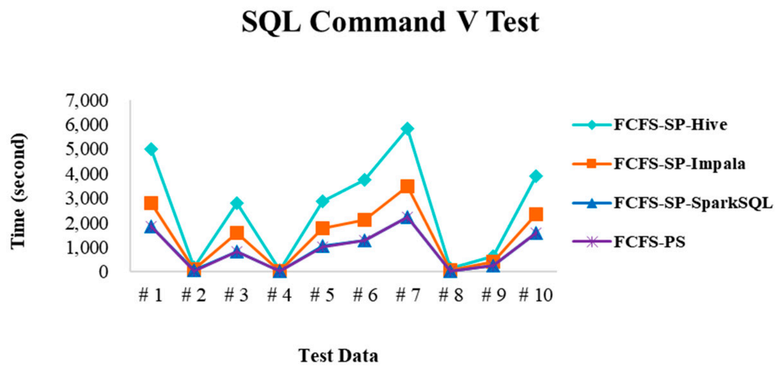


Figure 29. Execution time of SQL command V in test environment III.

4.2.4. Performance Evaluation

Fewer misses occurred in the experiments because the cache size was bigger enough in our design, where the cache size of the in-memory cache allocated 5 GB as well as an in-disk cache of 20 GB. The probability of the occurrence of a cache miss was about 0.0034 (34/1000) for the case of the in-memory cache and 0.0016 (16/1000) for the in-disk cache. According to the computed data from Figures 14–25, the weighted average execution time for launching a SQL query command to each approach was evaluated and then a summary of the performance comparison among Hive, Impala, SparkSQL, and the proposed methods are listed in Table 9 where the value in each cell means the average of the execution time for four commands running four data sets in a specific test environment and eventually the average of execution time for a specific approach for three test environments. As a result, the proposed approach outperformed the other benchmarks to show its best performance for a SQL query command execution in the big data environment.

Table 9. Performance comparison among the methods for Case 1 (unit: s).

Method	Environment I	Environment II	Environment III	Weighted Average Execution Time
FCFS-SP-Hive	22,834	3604	3530	3567
FCFS-SP-Impala	-	2047	2033	2040
FCFS-SP-SparkSQL	-	2681	1257	1969
FCFS-PS	24,325	2169	1340	1755

Note: Symbol "-" indicates the datum was not available.

The following is to evaluate the performance index for the above-mentioned approaches. We first substituted the average execution times from Table 9 into Equation (3) to find the normalized performance index among different approaches in the test, as listed in Table 10. Next, we substituted those results into Equation (4) to find the average normalized performance index, as listed in Table 11. As a result, in Table 11 it is noted that the proposed approach in this paper obtained the best performance index when compared with the others.

Table 10. Normalized performance index for Case 1.

Operation	FCFS-SP-Hive	FCFS-SP-Impala	FCFS-SP-SparkSQL	FCFS-PS
Environment I	-	-	-	-
Environment II	0.5436	0.9940	0.7573	1.0000
Environment III	0.3414	0.6082	0.9934	1.0000

Note: Symbol "-" indicates the datum was not available.

Table 11. Average normalized performance index for Case 1.

Method	Average Normalized PI	Performance Index
FCFS-SP-Hive	0.443	44
FCFS-SP-Impala	0.801	80
FCFS-SP-SparkSQL	0.875	88
FCFS-PS	1.000	100

4.3. Experimental Environment in Case 2

Case 2 shows test data collected from the real-world use case, as listed in Table 12. Each use case had its own a specific SQL command making sense of statistics used to measure the average execution time as per request. The test environment is as shown in Table 5. The test tool is shown in Table 8. The real-world data with various use cases are listed as follows: (1) world-famous masterpiece, (2) load of production machine: underload, (3) load of production machine: overload, (4) qualified rate of semiconductor products, (5) correlation between temperature and electricity use per capita, (6) correlation between rainfall and electricity use per capita, (7) flight information in the airport, (8) traffic violation/accidents, and their related information are as follows:

Table 12. Test use case.

No.	Data Size	Use Case	Codename
1	10 GB	World-famous masterpiece	WC
2	250 GB	Load of production machine: Overloading	OD
3	250 GB	Load of production machine: Underloading	UD
4	1 TB	Qualified rate of semiconductor products	YR
5	750 GB	Correlation among temperature and people's power utilization	TE
6	750 GB	Correlation among rainfall and people's power utilization	RE
7	100 GB	Flight information in the airport	AP
8	500 GB	Traffic violation/accidents	TA

The data of use case WC is read from the real world-famous masterpiece—*Alice-in-Wonderland Art-of-War*, *Huckleberry-Finn*, *Sherlock-Holmes*, and *Tom-Sawyer*, and we carried out the work of Word Count in this experiment. The second and third use cases, OD and UD, collected the data from a certain large semiconductor factory in Taiwan that included the goods number, responsible employee, name of production machine, and so on, and the file format was a standard xls format of EXCEL to find out the machine with a too high or too low use frequency in the production scheduling. The fourth use case YR received the data including various semiconductor test items and PASS or FAIL results in which the file format was a standard csv format from another big semiconductor factory in Taiwan, and the experiment was to calculate the yield rate of the products to see whether it reached the qualified standard of the company (99.7%). The fifth and sixth use cases, TE and RE, acquired the data set out of the official website at the Taipower company (Taipei, Taiwan), Taiwan's biggest electricity supply firm, and collect the information about Taiwan's area rainfall and temperature data from the Central Weather Bureau website as well. The purpose of this experiment was to find out the correlation between Taiwan's rainfall and electricity use per capita, and the temperature and electricity use per capita based on statistical correlation. In the use case AP, flight information in the airport with the standard csv data file was collected from an airport in New York and the purpose of this experiment was to calculate the frequency in statistics about how many times the departure flights get to the next airport each week. The final use case TA examined the data concerning traffic violation/accidents reported in USA and this experiment aimed to calculate the frequency in statistics regarding how often traffic violations/accidents occur every month.

4.4. Experimental Results in Case 2

The experiments have were applied in various experimental environments (test environment I, II, and III) to carry out every use case with its specific SQL command and collected data set. Similarly, for all of the experiments in Case 2, the response with a query for In-memory Cache was always within 5 s, and for In-disk Cache, the response was between 5 and 10 s. Therefore, the following experimental results were given by the other methods. As a result, the averaged execution time for the experiment of each use case with test environment I, II, and III are shown in Figures 30–32, respectively.

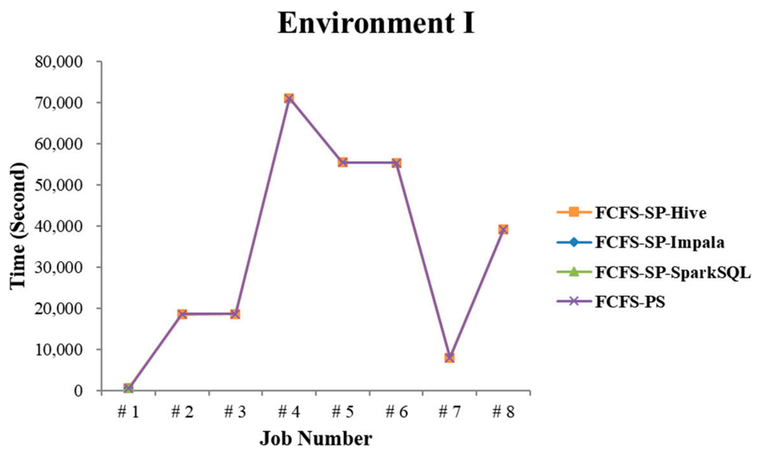


Figure 30. Execution time of each use case in test environment I.

Similarly, the performance comparison and performance index for Case 2 have been done in the same way as Case 1. Table 13 gives the performance comparison among the methods presented in this paper. Furthermore, the normalized performance indexes in two different test environments among the methods were evaluated as indicated in Table 14. Table 15 shows the performance index for Case 2 between the tool selection and the other alternatives.

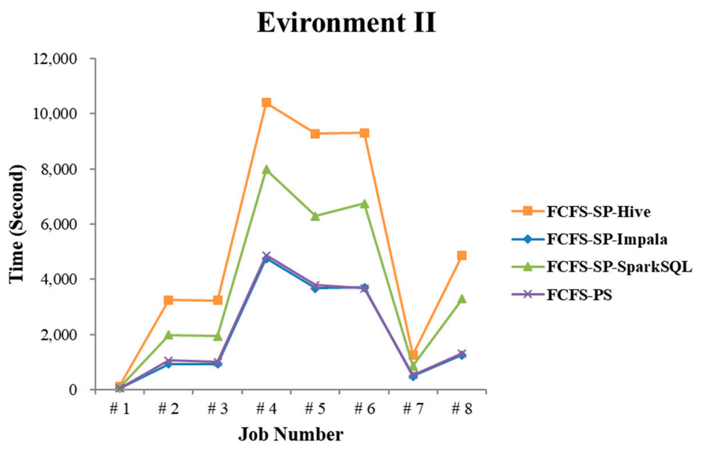


Figure 31. Execution time of each use case in test environment II.

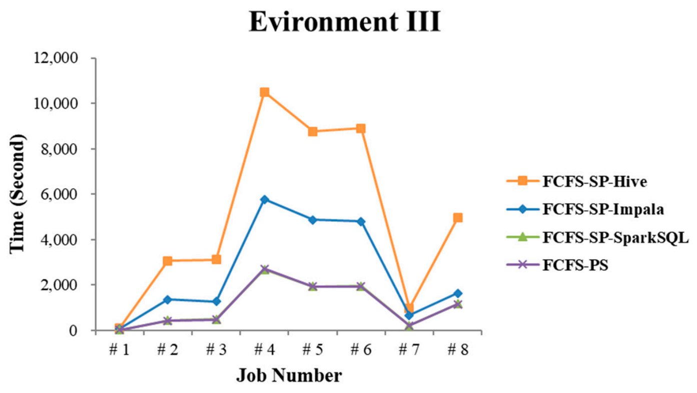


Figure 32. Execution time of each use case in test environment III.

Table 13. Performance comparison among the methods for Case 2 (unit: s).

Method	Environment I	Environment II	Environment III	Weighted Average Execution Time
FCFS-SP-Hive	33,405	5215	5045	5130
FCFS-SP-Impala	-	1972	2553	2262
FCFS-SP-SparkSQL	-	3647	1110	2378
FCFS-PS	33,406	2036	1105	1570

Note: Symbol "-" indicates the datum is not available.

Table 14. Normalized performance index for Case 2.

Operation	FCFS-SP-Hive	FCFS-SP-Impala	FCFS-SP-SparkSQL	FCFS-PS
Environment I	-	-	-	-
Environment II	0.3781	1.0000	0.5582	1.0000
Environment III	0.2190	0.4329	0.9959	1.0000

Note: Symbol "-" indicates the datum is not available.

Table 15. Average normalized performance index for Case 2.

Method	Average Normalized PI	Performance Index
FCFS-SP-Hive	0.299	30
FCFS-SP-Impala	0.716	72
FCFS-SP-SparkSQL	0.777	78
FCFS-PS	1.000	100

4.5. Discussion

This work presented an important idea to prevent program crashes during the query execution, as shown in Figures 15–19 and 30 where Impala and/or SparkSQL broke down due to an insufficient amount of memory remaining in any computing node. Therefore, a multi-tool system having integrated Hive, Impala, and SparkSQL needed to be built to sustain real-time high-speed parallel computing in a cluster. As a result, the speed of a job execution using the proposed approach of platform selection was 2.63 times faster than Hive in Case 1 experiment, and 4.57 times faster in the Case 2 experiment. In particular, if the target hits the in-memory cache or the in-disk cache, the query could reduce the response time to within seconds, as the SQL commands were repeated. Therefore, the query response hitting the in-memory cache will get the best performance index of 100 when compared with the other methods. Notice that SparkSQL was almost the same performance as the platform selection as allocated memory was sufficient for SparkSQL. Nevertheless, the platform selection, together with the cache mechanism, enabled the integrated system to have the best performance because the former can select the appropriate tool in a timely manner to handle the input query with effectiveness and efficiency in a multi-tool system and the latter gave the fastest query response if a cache hit occurred. The optimization of job scheduling would be another issue to take into account such as shortest job first, priority, and SQL complexity. In addition, an interesting viewpoint is to consider improving the network QoS so as to speed up data traffic over the network significantly. This could enhance the system throughput effectiveness and efficiency.

5. Conclusions

This paper has achieved automatically detecting the status of a cluster through checking the remaining memory size at nodes and then choosing the appropriate tool to deal with an SQL command for a fast query response. In addition to the fast query response shown in Case 1, the most significant contribution of this work is to implement applications, such as OLAP, data mining, and real-time statistics, by using complex SQL commands for real world use cases, as demonstrated in Case 2. In future work, we will devote to reformulating this case study with newer versions of the components (Hive, Impala, and SparkSQL) to use underlying new versions of Apache Spark and Apache Hadoop.

Author Contributions: B.R.C. and Y.-D.L. conceived and designed the experiments; H.-F.T. collected the experimental database and proofread the paper; B.R.C. wrote the paper.

Funding: This research received funding from Ministry of Science and Technology, Taiwan, Republic of China, under grant numbers MOST 105-2221-E-390-013-MY3 and MOST 107-2622-E-390-002-CC3.

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

References

1. Almgren, K.; Kim, M.; Lee, J. Extracting Knowledge from the Geometric Shape of Social Network Data Using Topological Data Analysis. *Entropy* **2017**, *19*, 360. [CrossRef]
2. Fan, S.; Lau, R.Y.; Zhao, J.L. Demystifying Big Data Analytics for Business Intelligence through the Lens of Marketing Mix. *Big Data Res.* **2015**, *2*, 28–32. [CrossRef]
3. Wixom, B.; Ariyachandra, T.; Douglas, D.E.; Goul, M.; Gupta, B.; Iyer, L.S.; Turetken, O. The Current State of Business Intelligence in Academia: The Arrival of Big Data. *Commun. Assoc. Inf. Syst.* **2014**, *34*, 1–13.
4. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Anthony, S.; Murthy, R. Hive: A Warehousing Solution over A Map-Reduce Framework. *Proc. VLDB Endow.* **2009**, *2*, 1626–1629. [CrossRef]
5. Liu, T.; Margaret, M. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. *ACM Sigplan Not.* **2003**, *38*, 107–118. [CrossRef]
6. Yadav, R. *Spark Cookbook*; Packt Publishing Ltd.: Birmingham, UK, 2015.
7. Shvachko, K.V. Apache Hadoop: The Scalability Update. *Login Mag. USENIX* **2011**, *36*, 7–13.
8. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; Mccauley, M.; Stoica, I. Fast and Interactive Analytics over Hadoop Data with Spark. *Login Mag. USENIX* **2012**, *37*, 45–51.

9. Borthakur, D. The Hadoop Distributed File System: Architecture and Design. *Hadoop Proj. Website* **2007**, *11*, 21.
10. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
11. Chang, F.; Dean, J.; Ghemawat, S.; Hsieh, W.C.; Wallach, D.A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the 2006 USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, USA, 6–8 November 2006; pp. 205–218.
12. Ghemawat, S.; Gobioff, H.; Leung, S.T. The Google File System. In Proceedings of the ACM SIGOPS Operating Systems Review (SOSP '03), Bolton Landing, NY, USA, 19–22 October 2003; Volume 37, pp. 29–43.
13. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS Oper. Syst. Rev.* **2007**, *41*, 205–220. [CrossRef]
14. Casado, R.; Younas, M. Emerging Trends and Technologies in Big Data Processing. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 2078–2091. [CrossRef]
15. Abadi, D.; Babu, S.; Özcan, F.; Pandis, I. SQL-on-Hadoop Systems: Tutorial. *Proc. VLDB Endow.* **2015**, *8*, 2050–2051. [CrossRef]
16. Bajaber, F.; Elshawi, R.; Batarfi, O.; Altalhi, A.; Barnawi, A.; Sakr, S. Big Data 2.0 Processing Systems: Taxonomy and Open Challenges. *J. Grid Comput.* **2016**, *14*, 379–405. [CrossRef]
17. Zlobin, D.A. In-Memory Data Grid. 2017. Available online: http://er.nau.edu.ua/bitstream/NAU/27936/1/Zlobin%20D.A.pdf (accessed on 1 August 2018).
18. Chang, B.R.; Tsai, H.-F.; Tsai, Y.-C. High-Performed Virtualization Services for In-Cloud Enterprise Resource Planning System. *J. Inf. Hiding Multimed. Signal Process.* **2014**, *5*, 614–624.
19. Proxmox Virtual Environment. Available online: https://p.ve.proxmox.com/ (accessed on 1 August 2018).
20. Chang, B.R.; Tsai, H.-F.; Chen, C.-M.; Huang, C.-F. Analysis of Virtualized Cloud Server Together with Shared Storage and Estimation of Consolidation Ratio and TCO/ROI. *Eng. Comput.* **2014**, *31*, 1746–1760. [CrossRef]
21. Thusoo, A.; Sarma, J.S.; Jain, N.; Shao, Z.; Chakka, P.; Zhang, N.; Antony, H.S.; Liu, R.; Murthy, R. Hive—A Petabyte Scale Data Warehouse using Hadoop. In Proceedings of the IEEE 26th International Conference on Data Engineering, Long Beach, CA, USA, 1–6 March 2010; pp. 996–1005.
22. LLVM 3.0 Release Notes. Available online: http://releases.llvm.org/3.0/docs/ReleaseNotes.html (accessed on 1 August 2018).
23. Gibilisco, G.P.; Krstic, S. InstaCluster: Building a big data cluster in minutes. *arXiv* **2015**, arXiv:1508.04973.
24. Fitzpatrick, B. Distributed Caching with Memcached. *Linux J.* **2004**, *2004*, 5.
25. Chang, B.R.; Tsai, H.-F.; Chen, C.-Y.; Guo, C.-L. Empirical Analysis of High Efficient Remote Cloud Data Center Backup Using HBase and Cassandra. *Sci. Program.* **2015**, *294614*, 1–10. [CrossRef]
26. Li, P. Centralized and Decentralized Lab Approaches Based on Different Virtualization Models. *J. Comput. Sci. Coll.* **2010**, *26*, 263–269.
27. Chang, B.R.; Tsai, H.-F.; Chen, C.-M. Assessment of In-Cloud Enterprise Resource Planning System Performed in a Virtual Cluster. *Math. Probl. Eng.* **2014**, *2014*, 947234. [CrossRef]
28. Many Books. Available online: http://manybooks.net/titles/shakespeetext94shaks12.html (accessed on 1 August 2018).