SCSI: Real-Time Data Analysis with Cassandra and Spark

Chapter · June 2019

DOI: 10.1007/978-981-13-0550-4_11

2 authors:

Dr-Archana Chaudhari
Symbiosis International (Deemed University) Pune
15 PUBLICATIONS **122** CITATIONS
SEE PROFILE

Preeti Mulay
79 PUBLICATIONS **550** CITATIONS
SEE PROFILE

SCSI: Real-Time Data Analysis with Cassandra and Spark

Archana A. Chaudhari and Preeti Mulay

Highlights

- The open-source framework for stream processing and enormous information
- In-memory handling model executed with the machine learning algorithms
- The data used in subset of non-distributed mode is better than using all data in distributed mode
- The Apache Spark platform handles big data sets with immaculate parallel speedup.

Abstract The dynamic progress in the nature of pervasive computing datasets has been main motivation for development of the NoSQL model. The devices having capability of executing "Internet of Things" (IoT) concepts are producing massive amount of data in various forms (structured and unstructured). To handle this IoT data with traditional database schemes is impracticable and expensive. The large-scale unstructured data required as the prerequisites for a preparing pipeline, which flawlessly consolidating the NoSQL storage model such as Apache Cassandra and a Big Data processing platform such as Apache Spark. The Apache Spark is the data-intensive computing paradigm, which allows users to write the applications in various high-level programming languages including Java, Scala, R, Python, etc. The Spark Streaming module receives live input data streams and divides that data into batches by using the Map and Reduce operations. This research presents a novel and scalable approaches called "***Smart Cassandra Spark Integration (SCSI)***" for solving the challenge of integrating NoSQL data stores like Apache Cassandra with Apache Spark to manage distributed systems based on varied platter of amalgamation of current technologies, IT enabled devices, etc., while eliminating complexity and risk. In this chapter, for performance evaluations, SCSI Streaming framework is compared

A. A. Chaudhari
Symbiosis International (Deemed University), Pune, India
e-mail: chaudhari.archana12@gmail.com

P. Mulay (✉)
Department of CS, Symbiosis Institute of Technology, Symbiosis International (Deemed University), Pune, India
e-mail: preeti.mulay@sitpune.edu.in

© Springer Nature Singapore Pte Ltd. 2019
M. Mittal et al. (eds.), *Big Data Processing Using Spark in Cloud*,
Studies in Big Data 43, https://doi.org/10.1007/978-981-13-0550-4_11

with the *file system-based data stores* such as Hadoop Streaming framework. SCSI framework proved scalable, efficient, and accurate while computing big streams of IoT data.

Keywords Apache Spark · Apache Cassandra · Big data · IoT · MapReduce

1 Introduction

Best way to predict the future is to create it!

—Peter F. Drucker

As mention in above proverb, the writer advises that with the help of technologies it is best to create future in addition to predictions based on Data Mining, Data Warehouse and other related techniques. This is also true with Incremental Learning, *Internet of Everything* (*IoE*), and the new innovation and methodologies that have arrived at our doorstep.

Internet of Things (*IoT*) [1] is very widely used concept and computations around the world. The administration, the scholarly community, and industry are engaged with various parts of research, execution, and business with IoT. The domains in which IoT-based systems are widely used include agriculture, space, health care, manufacturing, construction, water, and mining, etc., to name a few. IoT-based applications such as digitizing agriculture, electric power and human interaction, smart shopping system, electric foundation management in both urban and provincial regions, remote health care monitoring and notification systems, and smart transportation systems are widely used and becoming popular.

The development of data created by means of IoT has assumed a noteworthy part in the big data landscape. The prevalence of IoT makes big data analytics challenging because of handling and accumulation of information from various sensors in the IoT environment. In IoT, different sensors and multivariate objects are involved in data generation, which includes variety, noise, abnormality, and quick development. The data generated using IoT-based devices have different formats as compared to the big data. The list of new dimensions is added to the big data recently and they are popularly known as "7 Vs of Big Data". In this chapter, the Electricity Smart Meter data is considered for research study. This Electricity Smart Meter is based on IoT, GPS, and Cloud Computing concepts. The more elaborate discussion is given in following sections related to Electricity Smart Meter. Based on above discussion, and research it is observed that the same "7 Vs of Big Data" can be mapped directly to multidimensional IoT Data, in which seven estimations relate to the basic parts of IoT/IoE data [2] as shown in Fig. 1.

The dimensions of IoT data can be defined as follows:

1. Volume: The IoT Data comes in an expansive size that is Petabytes, Gigabytes, Zettabytes (ZB) and Yottabytes (YB). The IoT makes exponential development in data.

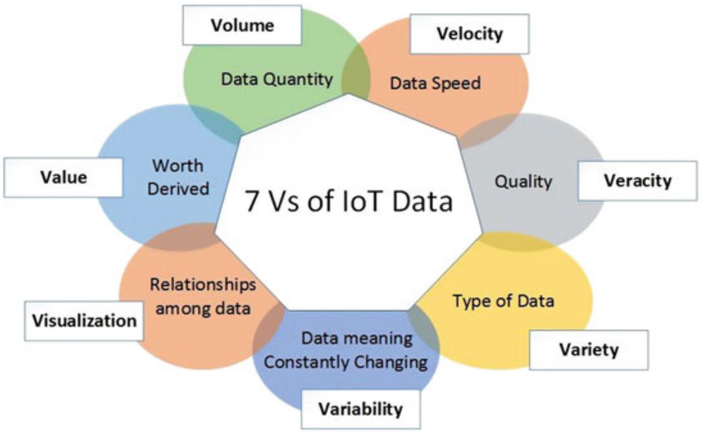


Fig. 1 The proposed dimensions of Internet of Thing (IoT) Data

2. Velocity: This is the most important thing related to dynamic data creation and its speed, along with data which is created, delivered, made, or invigorated.
3. Veracity: The huge amount of IoT data collected for data analysis purposes can lead to factual mistakes and distortion of the collected information. Veracity alludes more to the provenance or unwavering quality of the information source, its context, and how meaningful it is to the analysis based on it.
4. Variety: Big Data reaches out past organized information to incorporate unstructured information of all assortments, for example, as text, audio, video, log files, and more.
5. Variability: Variability is different from variety. Variability means if the significance of the information is continually transforming it can affect your information homogenization.
6. Visualization: Visualization refers to the relationship among data. Utilizing outlines and charts to imagine a lot of complex information is substantially much more effective in conveying meaning than spreadsheets and reports crammed with numbers and equations.
7. Value: Value is the capacity to change over IoT data into a concerning reward.

These 7Vs of IoT data layout is the direction to analytics, with each having inborn value in the process of discovering researchful details. The value of these IoT data is considered only when some useful information is extracted from it which is further used in decision making and pattern recognition, etc., by experts and analysts.

The dynamic and huge progress related to different categories of sensors related to IoT, scientific and industrial datasets are the main impetus behind the development of NoSQL model. As an example, in the IoT environment processing and collecting the data through different sensors including self-broadcasting sensors too.

A huge datasets are created by exchanging thoughts on social media sites including Facebook, Twitter and other platforms. The structure of datasets may differ drastically based on how the data is written, created, and stored using varied attributes. The structure of datasets also varies due to use of different social networking platforms, sensors attached to various mobile devices, servers, etc. A similar concept is discussed by authors in NERSC [3] where data generating from single experiment may include different sensors. Due to variety of sensors in single experiment the format of data varies. Before conceptualizing the NoSQL model, similar challenges existed and the major query was how to use methodologies for handling different structured data in separate datasets and analyzing each dataset separately. This way of handling large scale data is difficult to achieve the "big picture" without establishing basic connection between datasets. For predictions and decision-making based on collective data is the major concept for implementation based on NoSQL.

The IoT-/IoE-based systems are truly distributed in nature. These systems create large unstructured and semi-structured data which is growing exponentially and dynamically. The disturbed system approach is necessary for analyzing non-uniform data. The basic aim of an Apache Spark [4] as a model of decision is to effectually handle design and system issues related to Big Data. The author quotes in this [4] paper that Apache Spark offers functionalities to handle various processing and storage of unstructured, semi-structured, and any non-uniform data. However this support is not a straightforward based on query related to data. To extract required knowledge, or to achieve knowledge augmentation, the developing datasets not only need to be queried to permit timely information collection and sharing but also need to experience complex batch information analysis operations.

The NoSQL systems offer not only the capability of storing huge information but also queries to be applied on stored data. This data later analyze and mined for knowledge augmentation, NoSQL Data store is beneficial, as NoSQL data stores significantly handles in the micro-batches, there is a requirement for a product pipeline allowing Big Data processing models such as Apache Spark to tap IoT/IoE data as sources of input.

Summary of Introduction Section
The integrated research discussed in this chapter comprises of different domains viz., Big Data Analytics, Internet of Things (IoT), Database Management System, Distributed System, etc. This research presents a processing pipeline permitting Spark programs written in any language (such as Java, Scala, R, Python) to make utilization of the NoSQL storage frameworks. Such a pipeline is valuable in applications that require real-time processing. Also, presents the dimension of IoT data. This chapter develops "Smart Cassandra Spark Integration (SCSI)". SCSI utilize Apache Cassandra as a data storage, which is open source distributed framework, and consolidate it with Apache Spark as a real-time data processing, which is a speedy and distributed cluster computing system.

The research contribution for this chapter is as follows:

- The proposed SCSI handles real-time-dynamic data/time-series data for running real-time application with the integrated Spark and Cassandra frameworks. The

performance and evaluation of SCSI dynamic-data handling is based on MapReduce and Hadoop frameworks.
- The Apache Spark reads the information directly from the different data sources such as empirical multivariate datasets, sensors, time-series datasets and the formatted streaming data is stored into Cassandra for handling high fault-tolerance functionalities.
- The comparative analysis of the execution ramifications of processing data directly to the Apache Cassandra servers versus the use of real-time-dynamic data for processing its takes the snapshot of the database.

2 Background and Shortfalls of Available Approach

2.1 The Cassandra Approach

There are two meaning associated with the term "NoSQL". The first one is "Not only SQL" and other one is "not SQL". NoSQL databases support a subset of SQL commands. It has sprinkling advantages around SQL. The first advantage is there is no need of a rigidly describe schema for the insertion of data directly into the NoSQL database. This data can be changed at anytime, anywhere and these databases [5] adapts itself accordingly. The second advantage is the programmed division and smart detection of the usage of dynamic data. The multiple servers do not cause any problem in NoSQL system. As NoSQL is intelligent and high-level system, but in SQL, the programmer should implement the database schema according to the multiple-storing space arrangements considered in his or her design. The third advantage of a NoSQL data store over a SQL system is that the NoSQL achieves high-speed recovery of data by using caching the data in memory; means a NoSQL system stores valuable data in the cache, just like in computer a processor maintain the cache of recently used resources. Therefore, NoSQL reached a goal of high speed for design and execution of databases, NoSQL is essentially useful for today and future processing of data.

The distributed NoSQL database Cassandra [6] is thoughtfully implemented as non-relational and column-oriented. The initial phase of implementation of Cassandra is by Facebook [7] and now is an Apache project based on open source technologies.

Every Customer-Relationship-Management (CRM) system or Vendible-system generates large amount of dynamic/continuous data. The Cassandra is used to store such huge data effectively. Cassandra uses distributed cluster computing approach to store and handle such data. The cluster computing approach provides flexibility and scalability of the system. The data set consist of columns and rows. The combination of each column and row forms a specific cell having data or unique key related to the dataset. Each column can be sorted using some important key-value pair in all related files, which forms a concept called "keyspace". As mentioned by author in

[8] “the keyspace is alike schema of rational database having families of one or more column in the application”. The authors of [8] has also given the fundamental traits related to Cassandra including following:

- Effective MapReduce and Hadoop concept handling,
- Application systems based on HDFS, shared file systems,
- Scalability and redundancy support with fault tolerance,
- Adjustable data partitioning, etc.

2.2 *Hadoop and Hadoop Streaming*

The Apache Hadoop as discussed by authors in [9] supports an open-source MapReduce framework [10]. The Hadoop Distributed File Systems (HDFS) and MapReduce Frameworks are the two fundamental components of Hadoop Ecosystem. HDFS is used for data storage and management and MapReduce are used for execution of the application. The Hadoop’s JobTracker, execute on the Name node or master node. The aim of job tracker is resolving job details that mean a number of Mappers and Reducers required for the execution, listen to the progress of job and status of the worker(s). The DataNode and TaskTracker are located at the worker node. The distributed cluster computing environment supports collection and storage of data into the HDFS and the group of data chunks available for all clusters equally. Further these data-splits/chunks are processed by worker nodes. The local data nodes are responsible for local clustered data chunks.

The most popular platforms Hadoop and MapReduce are implemented using Java programming concepts, Mappers and Reducers utilizes Java APIs. The challenges in front of developers include rewriting legacy application systems using Java programming concepts, to be suitable for Hadoop platform detailing. Hadoop streaming as discussed in [11] provides one of the solutions which allows developers to code MapReduce jobs/Mapper Reducer’s operations in any programming or scripting languages, even-though Hadoop streaming has a confined model. The Hadoop streaming is used successfully to implement numerous scientific application systems from numerous domains. These applications system includes space-research related system, complex scientific processing, protein succession correlation, tropical storm detection, air waterway recognition, etc. to name a few.

2.3 *Spark and Spark Streaming*

The Apache Spark was incubated with the unique formulation of various clusters/splits from input data sets related to Vendible-systems/machines which are important parts of cluster computing, called workers. The Map Reduce concepts also motivate the use of or formulation of workers. The splitting of data and using it in parallel mode is feasible due to user-defined MapReduce functions.

The UC Berkeley AMP Lab is the place where the Apache Spark, one of the famous open-source distributed frameworks is cultured [12]. The Apache Spark is like Hadoop but it is based on the in-memory system to improve overall performance of system, processing, functions, and applications. It is a recognized analytics platform that ensures a fast, easy to use, and flexible computing. The Apache Spark handles complex analysis on large data sets for example, Spark process the programs almost hundred times faster than Apache Hadoop via MapReduce in-memory system.

The Apache Spark is based on the Apache Hive codebase. In order to improve system performance, Spark swap out the physical execution engine of Hive. In addition, Spark offers APIs to support a fast application development in various languages including Java, R, Python, and Scala [13]. Spark is able to work with all files storage systems that are supported by Hadoop. Apache Spark data model [14] is based on the Resilient Distributed Dataset (RDD) [15] abstraction. RDDs constitute a read-only collection of objects stored in system memory across multiple machines. Such objects are available without requiring a disk access. Furthermore, if a partition is lost it can be rebuilt. The Spark project consists of multiple components for recovery of fault, interacting with storage systems, scheduling of task, management of memory, etc.

The Spark components are listed as follows:

- Spark SQL [16]: One important feature of Spark SQL is that it unifies the two abstractions: relational tables and RDD. So programmers can easily mix SQL commands to query external data sets with complex analytics. Concretely, users can run queries over both imported data from external sources (like Parquet files Hive Tables) and data stored in existing RDDs. In addition, Spark SQL allows writing RDDs out to Hive tables or Parquet files. It facilitates fast parallel processing of data queries over large distributed data sets for this purpose. It uses query languages called "HiveQL". For a fast application development, Spark develop the Catalyst framework. This one enables users via Spark SQL to rapidly add new optimizations.
- Spark streaming [17]: Spark Streaming is another component that provides automatic parallelization, as well as scalable and fault-tolerant streaming processing. It enables users to stream tasks by writing batches like processes in Java and Scala [18]. It is possible to integrate batch jobs and interactive queries. Spark streaming execute series of short batch jobs in-memory data stored on in RDDs is called as Streaming computation.
- MLlib [19]: Machine Learning Library (MLlib) is a Spark implementation of common Machine Learning functionalities related to binary classification, clustering, regression and collaborative filtering etc. to name a few.
- GraphX [20]: GraphX constitutes a library for manipulating graphs and executing the graph-parallel calculation. Similar to Spark SQL and Spark Streaming, GraphX enhance the significance of Spark RDD API. Hence clients can make a coordinated

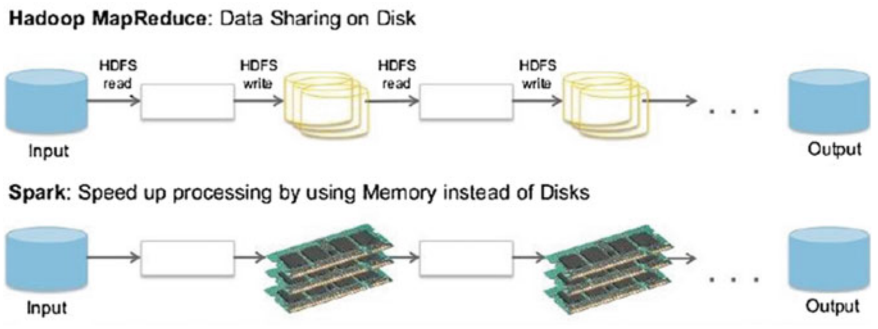

Hadoop MapReduce: Data Sharing on Disk
HDFS read
HDFS write
HDFS read
HDFS write
Input
Output
Spark: Speed up processing by using Memory instead of Disks
Input
Output

Fig. 2 Disk-based MapReduce versus Spark in-memory data processing

diagram with subjective properties connected to every vertex and edge. GraphX offers different operators that support graphs manipulation, for example, subgraph and Map Vertices.

The concept on which the Spark streaming operates is called as micro-batches. The Spark streaming accepts input dataset and divides that data into micro-batches [21], then the Spark engine processes those micro-batches to produce the final stream of results in sets/batches. Figure 2 shows the data processing details carried out by MapReduce through disk and the Spark processes the data in-memory, therefore the Spark is faster than MapReduce.

To Summarize

The Apache Spark is the most suitable platform for dynamic data/stream-data handling, and for real-time data analytics. In comparison with Hadoop, a Resilient Distributed Datasets (RDD) [15] is created and a Directed Acyclic Graph (DAG) [15] is prepared, as the related memory handling structures are maintained for Spark. It empowers the customers to store the data which is saved in the memory and to perform computation related to graph-cycle for comparable data, particularly from in-memory. The Spark framework saves enormous time of disk I/O operation. Hence Apache Spark is 100 times faster than Hadoop. The Spark can be set up to handle incoming real-time data and process it in micro-batches, and then save the result to resilient storage, such as HDFS, Cassandra, etc.

The Cassandra database is Distributed Database Management System, developed entirely in JAVA. The Cassandra has its own native query language called Cassandra Query Language, but it is a small subset of full SQL and does not support sufficient functionalities to handle aggregation and ad hoc queries.

3 Smart Cassandra Spark Integration (SCSI)

According to Mike Olson, Cloudera Chief Strategy Officer Sums up the breathtaking growth of the Spark and a paradigm shift in customer preferences witnessed by his Hadoop distributor company [22]—

> (Belissent, J.et al. 2010): *"Before very long, we expect that Spark will be the dominant general–purpose processing framework for Hadoop. If you want a good, general-purpose engine these days, you're choosing Apache Spark, not Apache MapReduce".*
>
> —Mike Olson

Not since peanut butter and jelly is an epic combo, the Spark is the world's foremost distributed analytics platform, delivering in-memory analytics with a speed and ease of use *unheard-of* in Hadoop. The Cassandra is the lighting fast distributed database powering IT giants such as Outbrain and Netflix.

Problem Statement
The Apache Spark is in-memory computing framework because it is preferably uses memory to cache partitions and having its own query language called as Spark SQL. The Apache Cassandra is Distributed NoSQL database. The Apache Cassandra is incompetent to handle aggregation and ad hoc queries. So when the Spark is paired with Cassandra, it offers a more *feature-rich* query language and allows to do data analytics that does not provided by Cassandra alone as well as the Spark alone.

The proposed framework ***Smart Cassandra Spark Integration (SCSI)*** is the collaboration of the Apache Cassandra and the Apache Spark. The proposed SCSI system architecture is as shown in Fig. 3 consist of a five key entities such as driver program, a cluster manager, workers, executors, and tasks. The master-slave model of distributed computing uses the SCSI. The node contains the cluster manager (or master process) is called master node of the cluster, while slave node contains the worker process. The functionalities or roles & responsibilities of a master node includes following:

- Resolve job details,
- Decide requirement of number of mappers and reducers for execution of jobs,
- Monitor the progress of every job,
- Keep log of every worker, etc.

The roles assigned to worker node or the worker nodes are conceptualized to gain following tasks:

- Receive, accept, and work on the commands from master node,
- Inform the status of every assigned work to the master,
- Execution of tasks and communication with local Cassandra database is the done by the executor,
- Entire application execution is done by the driver.

The proposed system SCSI uses the RDD data structures, which is called Resilient Distributed Datasets. RDD is dispersed datasets partitioned over processing nodes.

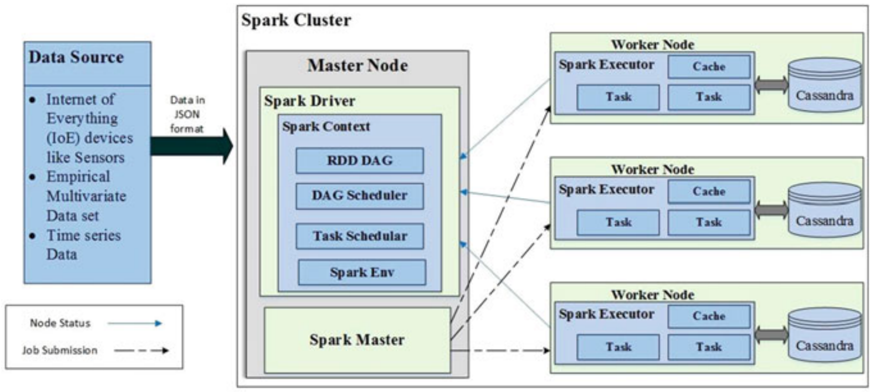


Fig. 3 High-level proposed SCSI system architecture

RDD records not only the number of partitions but also metadata of partitions. There are two ways to create the new RDD. The first, new RDD is generated whenever a transformation applies to an old RDD [23]. The recently created RDD object records, the dependency and transformation as a function object. The second new RDD is form by joining the various RDD together, the dependencies set of the new RDD is inherited from the parent RDD. The RDD transformation is same as declare a pointer to the particular object.

During the execution of a client application in SCSI system, the Driver and Worker plays an important role. The starting point for the execution of the application is the driver program, which performs the tasks scheduling, the tasks distribution, etc., over the multiple workers. Then workers manage parallel-processing tasks by using computing nodes and creates executor. The Executor communicates with the local data storage that means the Cassandra for task executions. The dataset for the system can be in the form of text, CSV, JSON, etc. For the proposed system JSON input file format is used. The sensors node generates the massive amount data in different formats, converts them into JSON formatto submit to the SCSI cluster. SCSI executes the task with the help of driver and worker, the details of execution are discussed in Sect. 3.2.

3.1 SCSI Node Deployment

This section of the chapter explains the role of SCSI nodes deployments on a standalone cluster. The new SCSI cluster is formed by an integration of the Apache Spark and the Apache Cassandra nodes in a big data cluster. Once the SCSI cluster is deployed, master nodes starts the master process and the slave node starts the worker process for controlling the entire cluster. Figure 4, demonstrates the integrated role of the Apache Spark and the Apache Cassandra nodes deployments on a standalone

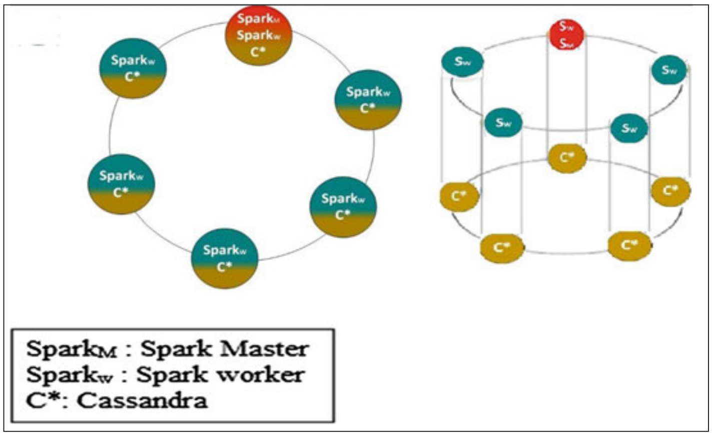


Fig. 4 The integrated role of the Apache Spark and Apache Cassandra nodes deployments on a standalone cluster

cluster. The standalone mode of cluster uses a basic cluster manager that is supplied with the Apache Spark. The spark master URL will be as follows:

SCSI://<hostname>:7077

Where, *<hostname>* is the name of the host to execute the master node details and *7077* is the port number, which is the default value, but it is configurable. This SCSI cluster manager supports First- In-First-Out (FIFO) job scheduling algorithm. It also allows concurrent application scheduling by setting the resource configuration options for each application. Each SCSI node in the cluster runs the Spark and the Cassandra combination. The logical flow of the execution of the application in the SCSI cluster is discussed in Sect. 3.2.

3.2 The Logical Flow of SCSI Cluster for Job Execution

Figure 5 describes the SCSI processes for execution of applications in parallel across the cluster nodes.

The Logical Flow (Algorithm) of Job Execution is as follows:
Step 1: Whenever sensor data is collected or client submits the application code, the Driver Program instantiates the Spark Context. The Spark Context converts the transformations and actions into logical DAG for execution. This logical DAG is then converted into a physical execution plan, which is then broken down into smaller physical execution units.
Step 2: The driver then interacts with the cluster manager or the Spark Master to negotiate the resources required to perform the tasks of the application code.

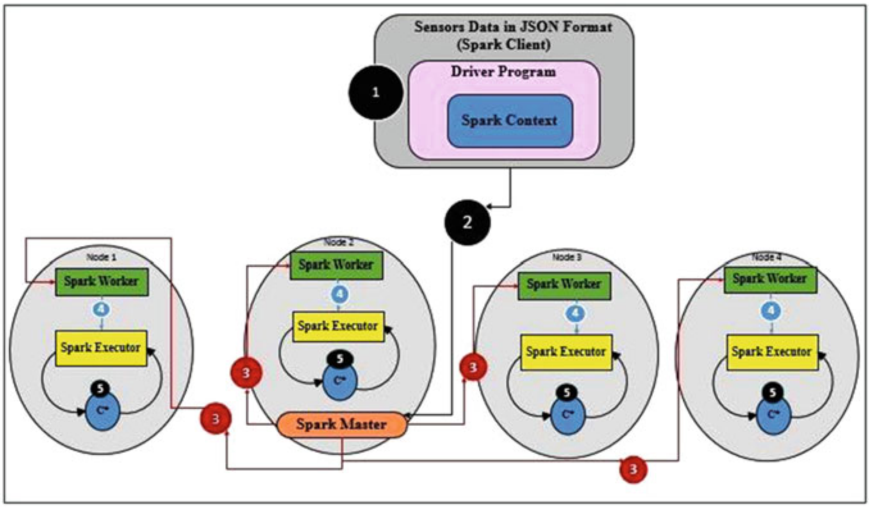


Fig. 5 The proposed sequential execution of client application

Step 3: The Spark Master then interacts with each of the worker nodes to understand the number of executors running in each of them to perform the tasks. The Spark worker consists of processes that can run in parallel to perform the tasks scheduled by the driver program, these processes are called the executors.
Step 4: The executors performs the data processing for the application code. In static allocation, the number of executors are fixed and run throughout the lifetime of the Spark application. In dynamic allocation, the user can also decide how many numbers of the executors are required to run the tasks, depending on the workload. Before the execution of tasks, the executors are registered with the driver program through the Master, so that the driver knows how many numbers of executors are running to perform the scheduled tasks. The executors then start executing the tasks scheduled by the worker nodes through the Spark Master
Step 5: Also, executor performs the following things:

i. Read from and write the data to the Cassandra database as describe in Sect. 3.3.
ii. The prediction of sensor data stored in Cassandra (i.e., memory, or disk).

3.3 The Internals Details of SCSI Job Execution

Instead of using any cluster manager [4], SCSI comes with the facility of a single script that can be used to submit a program, called as ***SCSI spark-submit***. It launches the application on the cluster. There are various options through which the *SCSI spark-submit* can connect to different cluster managers and controls many resources

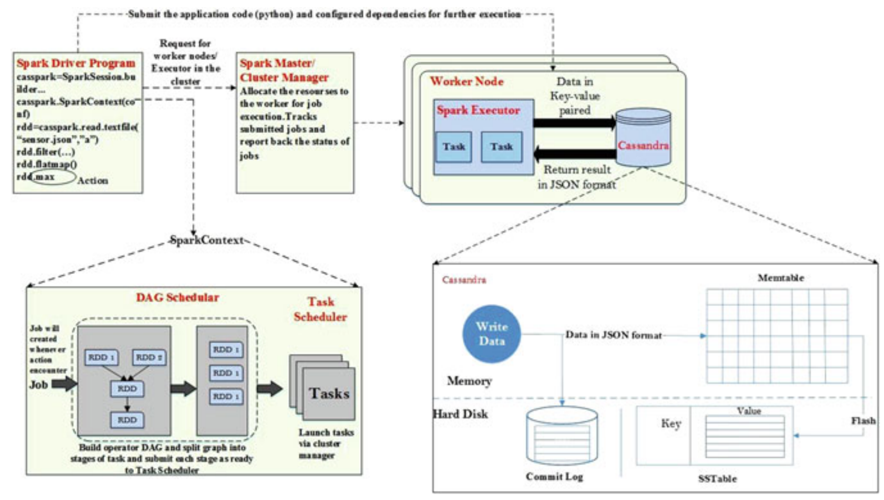

Submit the application code (python) and configured dependencies for further execution
Request for worker nodes/ Executor in the cluster
Spark Driver Program
casspark=SparkSession.builder...
casspark.SparkContext(conf)
rdd=casspark.read.textfile("sensor.json","a")
rdd.filter(...)
rdd.flatmap()
rdd.max
Action
Spark Master/ Cluster Manager
Allocate the resourses to the worker for job execution. Tracks submitted jobs and report back the status of jobs
Worker Node
Spark Executor
Task
Task
Data in Key-value paired
Cassandra
Return result in JSON format
SparkContext
DAG Schedular
Task Scheduler
Job will created whenever action encounter
Job
RDD 1
RDD 2
RDD
Tasks
Build operator DAG and split graph into stages of task and submit each stage as ready to Task Scheduler
Launch tasks via cluster manager
Cassandra
Memtable
Write Data
Data in JSON format
Memory
Hard Disk
Commit Log
Key
Value
SSTable
Flash

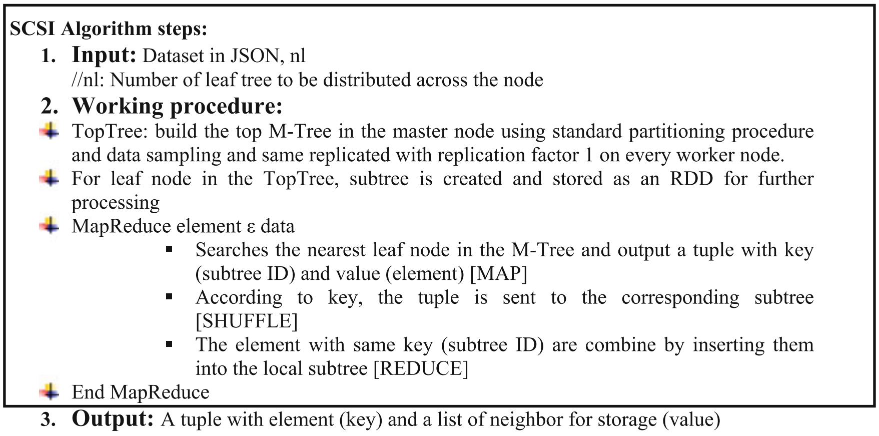
Fig. 6 The Internal details of the SCSI system during job execution

that application gets. For some cluster managers, *SCSI spark-submit* runs the driver within the cluster while for others, it runs only on the local machine. With this in mind, when *SCSI client submits* an application to the cluster with *SCSI spark-submit* this is what happens internally as shown in Fig. 6.

A standalone application starts and instantiates a SparkContext instance (and it is only then when the Spark client can call the application a driver). The driver program request to cluster manager or the Spark master for worker node/executor in the cluster, also asks for resources to launch executors. On behalf of the driver program, the cluster manager launches the executors. The driver process runs through the user application depending on the actions and transformations over RDDs task are sent to executors. The executors run the tasks and saves the result into the Cassandra. Whenever data is written into the Cassandra, it is first written in the Memtable (an actual memory) and at the same time, log files are committed to disk which ensures full data durability and safety. These log files are a kind of backup for each write to the Cassandra, which helps ensure data consistency even during a power failure because upon reboot, the data will be recovered in-memory from these log files. Adding more and more information to the Cassandra will finally result in reaching the memory limit. Then, the data stored by primary key is flushed into actual files on a disk called SSTables (sorted-strings table). Whenever any of the worker nodes does not respond to any query means that particular worker is not active, the tasks that are required to be performed will be automatically allocated to other worker nodes. SCSI automatically deals with this type of situation. For example, if Map () operation is executing on node1, and after some time node1, gets fail the SCSI rerun it on another node. Then execute the SparkContext. Stop () from the driver program or from the main method, all the executors terminated and the cluster resources released by the cluster manager.

3.4 The Algorithm for SCSI System

This section represents the lazy learning based solution for in-memory primitives of Spark using Distributed Matric Tree [24]. This algorithm forms a TopTree in the master node.



```
SCSI Algorithm steps:
  1. Input: Dataset in JSON, nl
     //nl: Number of leaf tree to be distributed across the node
  2. Working procedure:
  ✚ TopTree: build the top M-Tree in the master node using standard partitioning procedure
     and data sampling and same replicated with replication factor 1 on every worker node.
  ✚ For leaf node in the TopTree, subtree is created and stored as an RDD for further
     processing
  ✚ MapReduce element ε data
          ▪ Searches the nearest leaf node in the M-Tree and output a tuple with key
            (subtree ID) and value (element) [MAP]
          ▪ According to key, the tuple is sent to the corresponding subtree
            [SHUFFLE]
          ▪ The element with same key (subtree ID) are combine by inserting them
            into the local subtree [REDUCE]
  ✚ End MapReduce
  3. Output: A tuple with element (key) and a list of neighbor for storage (value)
```

3.5 The List of Benefits of SCSI System

The SCSI system which is basically an innovative integration of the Apache Spark and the Cassandra proved various real-time system characteristics and they are as follows:

1. **Speedup**: It is observed that the speed of code execution by SCSI is hundred time faster than Hadoop MapReduce. This became feasible due to SCSI's supports to acyclic data flows and in-memory computations by using advanced Directed Acyclic Graph execution engine.
2. **Usability**: User is given freedom to code in varied high-level programming languages using Java, Scala, Python, and R, etc. SCSI has built user-friendly and compatible parallel apps from high-level operators to use it interactively with the Scala, Python, and R shells.
3. **Universal, broad, and comprehensive**: SCSI supports broad combination of SQL, real-time-dynamic data handling, data frames, MLlib, GraphX and the Spark functionalities together in single platform.
4. **Availability**: The varied range of platforms supported and integrated by SCSI includes standalone-SCSI with cluster-node in EC2 of Cloud Computing platform services, Hadoop, Mesos, etc. SCSI has the ability to passionately access

Table 1 Comparison of Cassandra queries with SCSI queries

Basis	Cassandra queries	SCSI queries
Join and union	No	Yes
Transformation	Limited	Yes
Outside data integration	No	Yes
Aggregations	Limited	Yes

data sources from HDFS, Cassandra, HBase, and S3. The SCSI can be executed using its standalone cluster mode, on EC2, on Hadoop YARN, and on Apache Mesos.

5. **Data Locality**: SCSI tasks are executed on the node that stores the data also. Hence its provide high data locality
6. **Execute the Aggregation Query**: SCSI system execute the SUM [25], MIN, MAX, AVG and other aggregations query. Also, execute the ad hoc queries. The Table 1 shows the comparison of Cassandra queries with SCSI queries.

3.6 Ancillary Learning About Integrated File Systems and Analytical Engine

i. **MapReduce and HDFS**
Few frameworks are very efficient for high-speed computation system being computation-intensive where some frameworks provide scalable data-intensive as well as computation-intensive architecture. The MapReduce has its traditional advantages with HDFS but with the growing necessity of real-time big data analytics, it needs to improve cluster resource management.

ii. **Spark and HDFS**
The Spark on HDFS is comparatively more suitable for big data analytics applications. The Spark supports in-memory computation features and the HDFS can deal with a huge volume of data. Together they provide high-speed processing with fault tolerance and data replication. The Spark's process the data by keeping intermediate results in main-memory (cache), because of this Spark is more suitable for those systems where iterative processing is required. Though the Spark on HDFS performs well for all analytical problems. They identify a correlation between different environmental indicators of the sensor datasets by using Hadoop and Spark.

iii. **MapReduce and Cassandra**
A new approach can be considered where Apache MapReduce will work on Cassandra data store, which reduces the read/write overhead. As the Cassandra is compatible with both the Apache MapReduce and the Apache Spark, its integration with the Hadoop MapReduce results in high fault tolerance.

Table 2 The existing technologies and their integration to support big data analytics

Storage	Data processing	Advantages	Disadvantages
HDFS	MapReduce/Yarn	• Scale-out architecture • Fault tolerance • Optimized scheduling • High availability	• Problem with resource utilization • Not suitable for real-time analytics
HDFS	Spark	• High computation speed • In-memory features • Data locality	• suitable for interactive processing
Cassandra	MapReduce/Yarn	• Scale-out architecture • Fault tolerance	• Not suitable for iterative processing
Cassandra	Spark	• Best suitable for iterative and interactive processing • High speed for parallel processing • Data locality • High scalability • Fault tolerance	• Complex structure

iv. **Spark and Cassandra**
For the real-time, online web and mobile applications dataset, the Apache Cassandra database is a perfect choice, whereas the Spark is the fastest processing of colder data in the data lakes, warehouses, etc. Its integration effectively supports the different analytic "tempos" needed to satisfy customer requirements and run the organization [26]. Their integration may result in high I/O throughput, the data availability, and the high-speed computation, the high-level data-intensive and the computation-intensive infrastructure as shown in Table 2.

4 The Mapreduce Streaming Above the Cassandra Dataset

4.1 *The SCSI Streaming Pipeline over the Cassandra Datasets*

Figure. 7 shows the details related to the SCSI proposal, introduced in this chapter. This section gives the details related to the SCSI pipelines for the Cassandra datasets. This SCSI streaming pipeline is designed using steps including Data preparation, transformation and processing. The details related to these three stages, performance

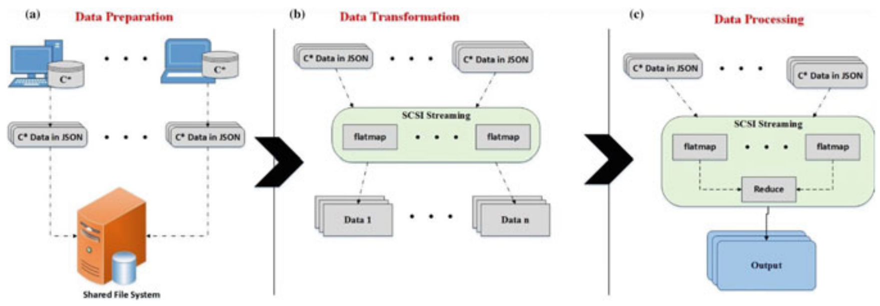


Fig. 7 The three stages of the SCSI streaming pipeline structure architecture

results, etc., are described in following Sect. 5. As shown in Fig. 7a shows, Data Preparation stage contains various worker nodes. These worker nodes are responsible for fetching the dataset from the local the Cassandra servers. Further, these worker nodes also carefully store the dataset details into the shared file system. Figure 7b shows, the Data Transformation stage which exports dataset in JSON format. The flat-map function is used to convert datasets into required specific formats. Figure 7c shows, the Data Processing stage details, which also uses the flat-map function. The task of this stage involves reducing the required procedures involved in reformatted data, produced in (b).

4.1.1 The First Stage of SCSI: Data Preparation

In the first stage of SCSI, for data preparation, the required input datasets are made available using the Cassandra servers. These datasets for ease of further processing are stored into the HDFS like distributed the file system having shared modes. As discussed by authors in [7], the Cassandra permits exporting data details from datasets fetched by servers into equivalent and required JSON formats. Each SCSI node is able to download data from linked-server powered by Cassandra to shared file formats, utilizing this built-in characteristic of the Cassandra. For every write request for the Cassandra server, data is first written in the Memtable (an actual memory) and at the same time, the log files are committed to the disk which ensures full data durability and safety. These log files are a kind of backup for each write to the Cassandra, which helps ensure the data consistency even during a power failure because upon reboot, the data will be recovered in-memory from these log files. Adding more and more information to the Cassandra will finally results in reaching the memory limit. Then, the data stored by primary key is flushed into actual files on a disk called Sorted-Strings Table, SSTables, (for details, refer the Sect. 3.3).

For the experimental setup of the SCSI framework, each worker is connected to its linked-server powered by the Cassandra and is capable to export actual Memory table called Memtable into the Stored String Table. Once the flushing of the data is completed, a worker starts the exporting operation. By using the "put" command, the associated worker nodes emphases on congregation of records in individual files, in shared mode. Also “put” command splits the input dataset into micro-batches and those chunks are place in the SCSI cluster. For more detailed comparison of SCSI with Hadoop is given in Sect. 5.1.

4.1.2 The Second Stage of SCSI: Data Transformation [MR1]

The first stage of SCSI is ready with the downloaded input datasets from the Cassandra servers, and followed by placing them to the files in JSON and sharable format. The SCSI architecture is proposed to handle issues related legacy application executables, which are difficult to rewrite or modify using Java for target results. The second stage of the SCSI architecture is Data Transformation (MR1) as shown in Fig. 7b.

The transformation phase involves in processing of each input records, and converts them to the required formats. The intermediate output files accommodate the results, as a part of transformation. This stage is used to start the flat-map operation of SCSI, however, no reduce operation is implemented yet. The responsibility of reducer function is to convert JSON files to appropriate format. The dependencies between nodes, data and or processing dependencies is not handled by this stage and hence best fitted for the SCSI streaming framework.

The Data Transformation stage of SCSI streaming is possible to implement in any programming language. For this research work the Python Scripts are used. The SCSI operations are based on the iterative data series, whose output becomes the input to the remaining stages of the SCSI streaming operations. This system allows users to specify the impactful attributes from given datasets and also convert the dataset in recommended file formats. This stage usually reduces data size, which ultimately improves the performance of the next stage. The SCSI and Hadoop streaming comparative analysis is further discussed in Sect. 5.2.

4.1.3 The Third Stage of SCSI: Data Processing [MR2]

The data processing stage execute the Python scripting programs, which was the initial target applications of data transformation, over the sensors data. The data is now available in a format that can be processed. In this stage of SCSI Streaming, flat-map and reduce operations is recommended and used, to run executables which is generated from the second stage of SCSI pipeline by using flat-map and reduce operation. The Hadoop and proposed SCSI streaming comparative analysis is further discussed in Sect. 5.3.

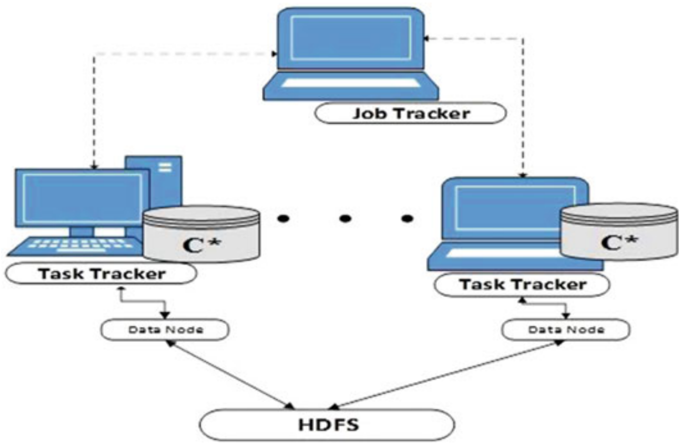


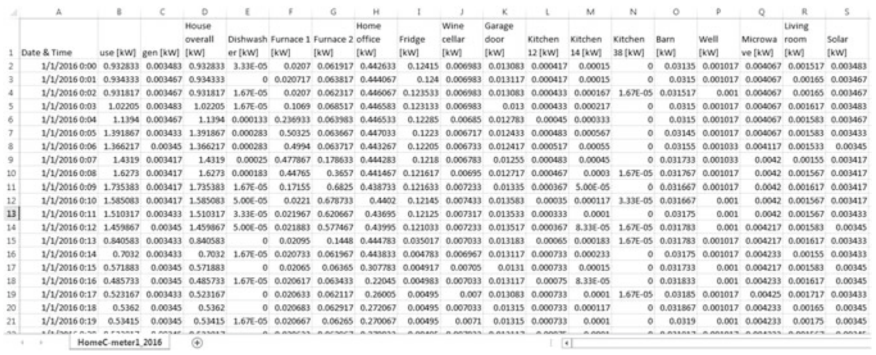
Fig. 8 The layout of Hadoop streaming over Cassandra dataset

4.2 Hadoop Streaming Pipeline Over Cassandra Datasets

The Hadoop streaming pipeline also works in three stages: data preparation, transformation, and processing. In the first stage, each data node of the Hadoop is connected to the linked-server powered by the Cassandra. By using the "put" command placed JSON formatted file into the Hadoop Distributed File System (HDFS) [11]. Also the "put" command splits the input dataset to the distribute data among the Data Nodes of the HDFS. These data-splits are input to the next stage (Data Transformation) of Hadoop Streaming pipeline. The HDFS requires the use of various APIs to interact with the files because it is a non-POSIX compliant file system [11]. The Hadoop streaming is implemented in non-Java programming language. The assumption is that the executables generated by the second stage of Hadoop streaming is not used by HDFS API due to this not having immediate access of the input splits. To address this issue, in the data processing, TaskTracker reads the input from the HDFS, processes them by data node and writes the results back to the HDFS. The layout of the Hadoop streaming over the Cassandra dataset is shown in Fig. 8.

5 Performance Results

This section explains the performance of the SCSI streaming system with the Hadoop streaming system with respect to computation speed. The experiments are carried out on a five nodes Spark cluster, one node is master, and the remaining four are workers. The Spark version 2.2.0 installed on each node, also installed the Apache Cassandra version 3.11.1 on each Spark slave nodes. The master node has Intel Core i5 6600 K 3.50 GHz Quad Core processor, 6 cores, a 64-bit version of Linux 2.6.15



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
1	Date & Time	use [kW]	gen [kW]	House overall [kW]	Dishwasher [kW]	Furnace 1 [kW]	Furnace 2 [kW]	Home office [kW]	Fridge [kW]	Wine cellar [kW]	Garage door [kW]	Kitchen 12 [kW]	Kitchen 14 [kW]	Kitchen 38 [kW]	Barn [kW]	Well [kW]	Microwave [kW]	Living room [kW]	Solar [kW]
2	1/1/2016 0:00	0.932833	0.003483	0.932833	3.33E-05	0.0207	0.061917	0.442633	0.12415	0.006983	0.013083	0.000417	0.00015	0	0.03135	0.001017	0.004067	0.001517	0.003483
3	1/1/2016 0:01	0.934333	0.003467	0.934333	0	0.020717	0.063817	0.444067	0.124	0.006983	0.013117	0.000417	0.00015	0	0.0315	0.001017	0.004067	0.00165	0.003467
4	1/1/2016 0:02	0.931817	0.003467	0.931817	1.67E-05	0.0207	0.062317	0.446067	0.123533	0.006983	0.013083	0.000433	0.000167	1.67E-05	0.031517	0.001	0.004067	0.00165	0.003467
5	1/1/2016 0:03	1.02205	0.003483	1.02205	1.67E-05	0.1069	0.068517	0.446583	0.123133	0.006983	0.013	0.000433	0.000217	0	0.0315	0.001017	0.004067	0.001617	0.003483
6	1/1/2016 0:04	1.1394	0.003467	1.1394	0.000133	0.236933	0.063983	0.446533	0.12285	0.00685	0.012783	0.00045	0.000333	0	0.0315	0.001017	0.004067	0.001583	0.003467
7	1/1/2016 0:05	1.391867	0.003433	1.391867	0.000283	0.50325	0.063667	0.447033	0.1223	0.006717	0.012433	0.000483	0.000567	0	0.03145	0.001017	0.004067	0.001583	0.003433
8	1/1/2016 0:06	1.366217	0.00345	1.366217	0.000283	0.4994	0.063717	0.443267	0.12205	0.006733	0.012417	0.000517	0.00055	0	0.03155	0.001033	0.004117	0.001533	0.00345
9	1/1/2016 0:07	1.4319	0.003417	1.4319	0.00025	0.477867	0.178633	0.444283	0.1218	0.006783	0.01255	0.000483	0.00045	0	0.031733	0.001033	0.0042	0.00155	0.003417
10	1/1/2016 0:08	1.6273	0.003417	1.6273	0.000183	0.44765	0.3657	0.441467	0.121617	0.00695	0.012717	0.000467	0.0003	1.67E-05	0.031767	0.001017	0.0042	0.001567	0.003417
11	1/1/2016 0:09	1.735383	0.003417	1.735383	1.67E-05	0.17155	0.6825	0.438733	0.121633	0.007233	0.01335	0.000367	5.00E-05	0	0.031667	0.001017	0.0042	0.001617	0.003417
12	1/1/2016 0:10	1.585083	0.003417	1.585083	5.00E-05	0.0221	0.678733	0.4402	0.12145	0.007433	0.013583	0.00035	0.000117	3.33E-05	0.031667	0.001	0.0042	0.001567	0.003417
13	1/1/2016 0:11	1.510317	0.003433	1.510317	3.33E-05	0.021967	0.620667	0.43695	0.12125	0.007317	0.013533	0.000333	0.0001	0	0.03175	0.001	0.0042	0.001567	0.003433
14	1/1/2016 0:12	1.459867	0.00345	1.459867	5.00E-05	0.021883	0.577467	0.43995	0.121033	0.007233	0.013517	0.000367	8.33E-05	1.67E-05	0.031783	0.001	0.004217	0.001583	0.00345
15	1/1/2016 0:13	0.840583	0.003433	0.840583	0	0.02095	0.1448	0.444783	0.035017	0.007033	0.013183	0.00065	0.000183	1.67E-05	0.031783	0.001017	0.004217	0.001617	0.003433
16	1/1/2016 0:14	0.7032	0.003433	0.7032	1.67E-05	0.020733	0.061967	0.443833	0.004783	0.006967	0.013117	0.000733	0.000233	0	0.03175	0.001017	0.004233	0.00155	0.003433
17	1/1/2016 0:15	0.571883	0.00345	0.571883	0	0.02065	0.06365	0.307783	0.004917	0.00705	0.0131	0.000733	0.00015	0	0.031733	0.001	0.004217	0.001583	0.00345
18	1/1/2016 0:16	0.485733	0.00345	0.485733	1.67E-05	0.020617	0.063433	0.22045	0.004983	0.007033	0.013117	0.00075	8.33E-05	0	0.031833	0.001	0.004233	0.001617	0.00345
19	1/1/2016 0:17	0.523167	0.003433	0.523167	0	0.020633	0.062117	0.26005	0.00495	0.007	0.013083	0.000733	0.0001	1.67E-05	0.03185	0.001017	0.00425	0.001717	0.003433
20	1/1/2016 0:18	0.5362	0.00345	0.5362	0	0.020683	0.062917	0.272067	0.00495	0.007033	0.01315	0.000733	0.000117	0	0.031867	0.001017	0.004233	0.00165	0.00345
21	1/1/2016 0:19	0.53415	0.00345	0.53415	1.67E-05	0.020667	0.06265	0.270067	0.00495	0.0071	0.01315	0.000733	0.0001	0	0.0319	0.001	0.004233	0.00175	0.00345

Fig. 9 Smart-electric meter dataset (*Source* http://traces.cs.umass.edu/index.php/Smart/Smart)

Table 3 Cassandra configuration

Parameter	Value
Cassandra heap size	2 GB
Partitioner	Random partitioner
Replication factor	1
Row caching	off

of 64-bit version and 32 GB of RAM. The slave node connected via 8 GB Gigabit Ethernet, which has Intel Xeon CPUs, 6 cores, 64 GB of RAM.

Dataset: The dataset to be used for research purpose is Electricity Smart Meter data connected at individual homes as shown in Fig. 9, total 50 K records in the dataset file. The case study also executed on different sensor data that means temperature, light and humidity datasets obtained from the UCI and other repositories [27].

For this research the Electricity Smart Meter dataset is chosen because next generation Smart-Energy-Meters is the necessity of this world today and it is the future. To effectually implement Smart-Energy-Meter systems on large scale, a true distributed system should be developed and maintained. Such Smart-Energy-Meters are the devices which are primarily based on IoT, GPS and Cloud computing concepts. The devices which are IoT enabled, generates massive amount of data, especially Smart-Energy-Meters will, for sure. To handle such multivariate massive data, SCSI is the best suitable model.

The Apache Cassandra and Apache spark configuration are specified in Tables 3 and 4 respectively. In addition to these, specified distributions of the default parameters shipped used by both Cassandra and spark.

Table 4 Spark configuration

Parameter	Value
dfs.block.size	128 MB
dfs.replication	1
Mapred.tasktracker.flapmap.tasks.maximum	6
Io.sort.mb	100 MB

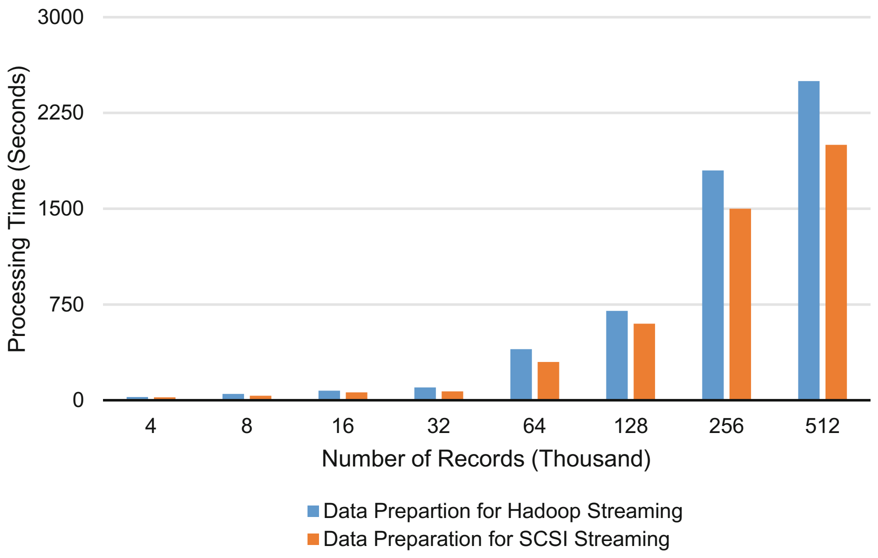


Fig. 10 The shifting of data from Cassandra server into the file system for SCSI and Hadoop Streaming. As the data sizes increases, cost is also increases

5.1 Data Preparation

For experimental setup, a worker node of the SCSI framework runs a Cassandra server. The aim is to store attributes like temperature, humidity, and light sensors data in the Cassandra. For this setup, replication factor is set to one for the Cassandra server's data distribution.

Figure 10 demonstrates the execution of SCSI streaming and Hadoop streaming by taking an image of the input dataset from the Cassandra into the shared file system for processing. As data size increases, the cost of shifting data from the Cassandra servers expectedly increases. There exists a linear relation between the shifting cost and the data size. The expense of shifting 4000 records takes nearly 60 times less than shifting 512 thousand input records. Additionally Fig. 10 demonstrates the difference between data preparation for SCSI Streaming and Hadoop Streaming. It is observed that at 4000 records, the speed of Data Preparation for SCSI Streaming is same as Hadoop streaming and it is more than 1.3 times faster at 64 and 256 thousand records.

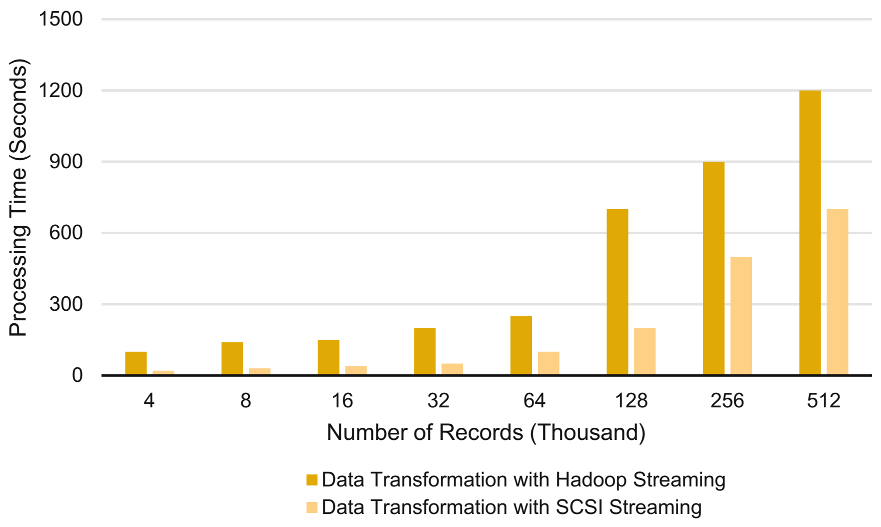


Fig. 11 Pre-processing of SCSI streaming applications for Cassandra data. SCSI streaming performance faster than Hadoop streaming

5.2 Data Transformation [MR1]

Figure 11 demonstrates the performance of Data Transformation phase, which converts the snapshot of the target dataset into the required format using both SCSI Streaming and Hadoop Streaming. At the 4000 input records, the speed of data transformation for SCSI streaming is ten times faster than Hadoop Streaming. At 64 and 256 thousand records SCSI streaming fifty percent faster than Hadoop streaming.

5.3 Data Processing [MR2]

This section explains the performance of executing the various application in Data Processing phase with the Hadoop Streaming and the SCSI Streaming. Also, it shows the overall cost for combining the data transformation and the data processing stages. As explained in Sect. 5.2, the Data Transformation stage not only converts the image of the input dataset into the desired format but also the size of the dataset is reduced. Due to the data size reduction, the data processing input to be much less as compared to the data transformation which ultimately improves the performance.

Figure 12 demonstrates the performance of the Data Processing, excluded in the Data Preparation and Data Transformation phases for the execution of the submitted application by user. At the stage of 4000 input records, the SCSI Streaming is 30% faster than the Spark Streaming and 80% speeder than the Hadoop Streaming. For the increased input data size, the Spark streaming is faster than the Hadoop Streaming

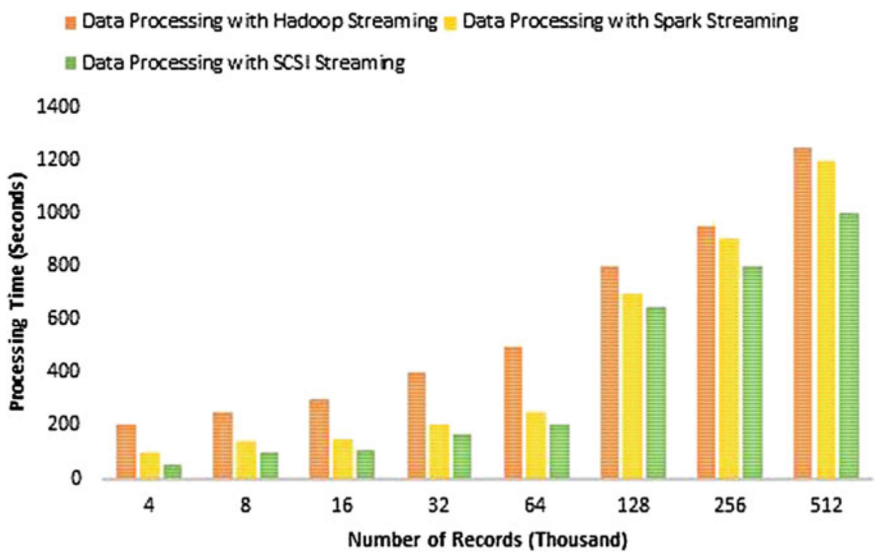


Fig. 12 Excluded the data preparation and data transformation phase during the data processing execution

and slower than the SCSI Streaming. At 64 and 256 thousand records, the speed of Data Processing for SCSI Streaming is fifty percent faster than Hadoop streaming.

Similarly, Fig. 13 demonstrates the performance of the execution of the same application, which includes Preparation and Transformation stages. For shifting the input dataset out of the database not only includes the cost of data movement but also needs split the exported data explicitly for each worker node in the cluster. At initial input data set, the SCSI Streaming is 1.3 times quick than the Hadoop streaming and at 512 thousand input records it is 3.6 times fast.

To Summarize

These two diagrams (Figs. 12 and 13) demonstrates that shifting information from the database accompanies a more cost which can easily overshadows the cost of accessing records straightforward from the database rather than the file system. As the research point of view, if the target application is advisable to keep running in data processing phase, it is best to store the input data in the cluster and use the SCSI system. That means reading the input by the Spark Mappers directly from the Cassandra servers and executes the given applications.

6 Related Work

As per Cluster of European research project on the Internet of Things [28], anything which having processing power is called as "Things", which are dynamic members

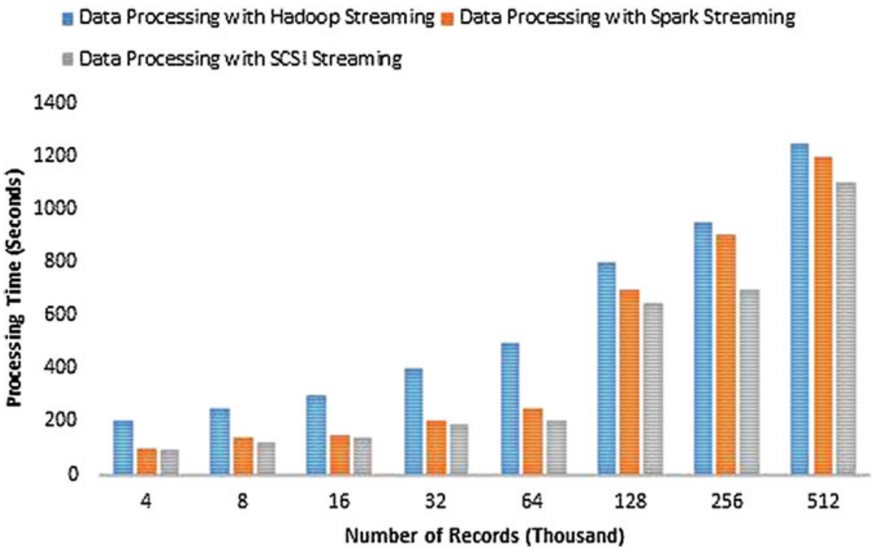


Fig. 13 Included the data preparation and data transformation phase during the data processing execution

in information and social processes, business and industries where they are enabled to interact and communicates with each other and with the environment by trading sensed data and information about the nature. IoT-based devices are capable of responding independently to the physical world's events and influencing it by running processes that trigger activities and make services with or without coordinate human intervention. As indicated by Forrester [22], a smart environment uses Information and Communications Technologies (ICT) to make the basic foundations, parts and administrations of a city organizations, medicinal services, public safety, real estate, education, transportation and utilities more aware, intuitive and effective. In [5, 29] explained the dataset generation technique by using the Horizontal Aggregation Method and by using the WEKA tool analyze this dataset. In [25] Structured Queries like Select, Project, Join, Aggregates, CASE, and PIVOT are explained.

There are plenty of applications which uses the NoSQL technology with MapReduce. In DataStax Enterprise [26] invented a system, in which Big Data [2] framework built on top of Cassandra which supports Hive, Pig, Apache Spark, and the Apache Hadoop. According to Thusoo et al. [29] Hive built on top of the Hadoop to support querying over the dataset which stored in a distributed file system like HDFS. According to the Huang et al. [23], Apache Spark is in-memory processing framework, also introduced the Remote Sensing (RS) algorithm and the remote sensing data incorporating with Resilient Distributed Datasets (RDDs) of Spark. According to Yang [30], Osprey is a middleware to provide MapReduce like adaptation to internal failure support to SQL databases. Osprey splits the structured queries into the number of subqueries. Also distributes the data over the cluster with replication factor three in classic MapReduce style. It provides the fault-tolerance and load balancing support to the Structured Query Language database, without concentrating on processing of data on such frameworks which utilizes the MapReduce style. According to the Kaldewey et al. [31], presents Clydesdale for handling organized information with Hadoop. They give a correlation about Hive, demonstrating the performance advantages of MapReduce model, more specifically Hadoop, but Clydesdale does not use the NoSQL database with non-Java applications.

7 Conclusion

To effectively combine the distributed data stores, such as Apache Cassandra with scalable distributed programming models, such as Apache Spark, it needs the software pipeline. This software pipeline allows users to write a Spark program in any language (Java, Scala, Python, and R) to make use of the NoSQL data storage framework concept. This research presented a novel scalable approache called ***Smart Cassandra Spark Integration (SCSI)*** for solving the challenges of integration of NoSQL data stores like Apache Cassandra with Apache Spark to manage distributed IoT data. This chapter depicts two diverse methodologies, one fast processing Spark working with the distributed Cassandra cluster directly to perform operations and the other exporting the dataset from the Cassandra database servers to the file system for further processing. Experimental results demonstrated the predominance and eminent qualities of SCSI streaming pipeline over the Hadoop streaming and also, exhibited the relevance of proposed SCSI Streaming under different platforms. The proposed framework is scalable, efficient, and accurate over a big stream of IoT data.

The Direction for future work is to integrate MPI/OpenMP with Cassandra. To improve speed performance of a Data Analytics in the Cloud, this idea is an interesting subject of the research.

Annexure

How to Install Spark with Cassandra

The following steps describe how to set up a server with both a Spark node and a Cassandra node (Spark and Cassandra will both be running on localhost). There are two ways for setting up a Spark and Cassandra server: if you have DataStax Enterprise [3] then you can simply install an Analytics Node and check off the box for Spark or, if you are using the open source version, then you will need to follow these steps.

This assumes you already have Cassandra setup.

1. Download and setup Spark

  i. Go to http://spark.apache.org/downloads.html.
  ii. Choose Spark version 2.2.0 and "Pre-built for Hadoop 2.4" then Direct Download. This will download an archive with the built binaries for Spark.
  iii. Extract this to a directory of your choosing: Ex. ~/apps/spark-1.2
  iv. Test Spark is working by opening the Shell

2. Test that Spark Works

  i. cd into the Spark directory
  ii. Run "./bin/spark-shell". This will open up the Spark interactive shell program
  iii. If everything worked it should display this prompt: "Scala>"
  iv. Run a simple calculation: Ex. sc.parallelize(1 to 100).sum(_+_)
  v. Exit the Spark shell with the command "exit"

The Spark Cassandra Connector
To connect Spark to a Cassandra cluster, the Cassandra Connector will need to be added to the Spark project. DataStax provides their own Cassandra Connector on GitHub and we will use that.

1. Clone the Spark Cassandra Connector repository: https://github.com/datastax/spark-cassandra-connector
2. cd into "spark-Cassandra-connector"
3. Build the Spark Cassandra Connector

  i. Execute the command "./sbt/sbt assembly"
  ii. This should output compiled jar files to the directory named "target". There will be two jar files, one for Scala and one for Java.
  iii. The jar we are interested in is "spark-cassandra-connector-assembly-1.1.1-SNAPSHOT.jar" the one for Scala.

4. Move the jar file into an easy-to-find directory: ~/apps/spark-1.2/jars

To Load the Connector into the Spark Shell

1. start the shell with this command:
   ../bin/spark-shell–jars~/apps/spark-1.2/jars/spark-cassandra-connector-assembly-1.1.1-SNAPSHOT.jar
2. Connect the Spark Context to the Cassandra cluster:

  i. Stop the default context: sc.stop
  ii. Import the necessary jar files:importcom.datastax.spark.connector._, import org.apache.spark.SparkContext, import org.apache.spark.SparkContext._, import org.apache.spark.SparkConf
  iii. Make a new SparkConf with the Cassandra connection details:Valcone=new SparkConf (true).set ("spark.cassandra.connection.host", "localhost")
  iv. Create a new Spark Context:valsc=new SparkContext(conf)

3. You now have a new SparkContext which is connected to your Cassandra cluster.
4. From the Spark Shell run the following commands:

   i. valtest_spark_rdd=sc.cassandraTable("test_spark", "test")
   ii. test_spark_rdd.first
   iii. The predicted output generated

References

1. Ray, P.: A survey of IoT cloud platforms. Future Comput. Inform. J. **1**(1–2), 35–46 (2016)
2. UMassTraceRepository. http://traces.cs.umass.edu/index.php/Smart/Smart
3. National energy research scientific computing center. http://www.nersc.gov
4. Apache Spark. http://spark.apache.org
5. Chaudhari, A.A., Khanuja, H.K.: Extended SQL aggregation for database. Int. J. Comput. Trends Technol. (IJCTT) **18**(6), 272–275 (2014)
6. Lakshman, A., Malik P.: Cassandra: structured storage system on a p2p network. In Proceeding of the 28th ACM Symposium Principles of Distributed Computing, New York, NY, USA, pp. 1–5 (2009)
7. Cassandra wiki, operations. http://wiki.apache.org/cassandra/Operations
8. Dede, E., Sendir, B., Kuzlu, P., Hartog, J., Govindaraju, M.: An evaluation of cassandra for Hadoop. In Proceedings of the IEEE 6th International Conference Cloud Computing, Washington, DC, USA, pp. 494–501 (2013)
9. Apache Hadoop. http://hadoop.apache.org
10. Premchaiswadi, W., Walisa, R., Sarayut, I., Nucharee, P.: Applying Hadoop's MapReduce framework on clustering the GPS signals through cloud computing. In: International Conference on High Performance Computing and Simulation (HPCS), pp. 644–649 (2013)
11. Dede, E., Sendir, B., Kuzlu, P., Weachock, J., Govindaraju, M., Ramakrishnan, L.: Processing Cassandra Datasets with Hadoop-Streaming Based Approaches. IEEE Trans. Server Comput. **9**(1), 46–58 (2016)
12. Acharjya, D., Ahmed, K.P.: A survey on big data analytics: challenges, open research issues and tools. Int. J. Adv. Comput. Sci. Appl. **7**, 511–518 (2016)
13. Karau, H.: Fast Data Processing with Spark. Packt Publishing Ltd. (2013)
14. Sakr, S.: Chapter 3: General-purpose big data processing systems. In: Big Data 2.0 Processing Systems. Springer, pp. 15–39 (2016)
15. Chen, J., Li, K., Tang, Z., Bilal, K.: A parallel random forest algorithm for big data in a Spark Cloud Computing environment. IEEE Trans. Parallel Distrib. Syst. **28**(4), 919–933 (2017)
16. Sakr, S.: Big data 2.0 processing systems: a survey. Springer Briefs in Computer Science (2016)
17. Azarmi, B.: Chapter 4: The big (data) problem. In: Scalable Big Data Architecture, Springer, pp. 1–16 (2016)
18. Scala programming language. http://www.scala-lang.org
19. Landset, S., Khoshgoftaar, T.M., Richter, A.N., Hasanin, T.: A survey of open source tools for machine learning with big data in the Hadoop ecosystem. J. Big Data 2.1 (2015)
20. Wadkar, S., Siddalingaiah, M.: Apache Ambari. In: Pro Apache Hadoop, pp. 399–401. Springer (2014)
21. Kalantari, A., Kamsin, A., Kamaruddin, H., Ebrahim, N., Ebrahimi, A., Shamshirband, S.: A bibliometric approach to tracking big data research trends. J. Big Data, 1–18 (2017)

Web References

22. Belissent, J.: Chapter 5: Getting clever about smart cities: new opportunities require new business models. Forrester Research (2010)
23. Huang, W., Meng, L., Zhang, D., Zhang, W.: In-memory parallel processing of massive remotely sensed data using an Apache Spark on Hadoop YARN model. IEEE J. Sel. Topics Appl. Earth Obs. Remote Sens. **10**(1), 3–19 (2017)
24. Soumaya, O., Mohamed, T., Soufiane, A., Abderrahmane, D., Mohamed, A.: Real-time data stream processing-challenges and perspectives. Int. J. Comput. Sci. Issues **14**(5), 6–12 (2017)
25. Chaudhari, A.A., Khanuja, H.K.: Database transformation to build data-set for data mining analysis—a review. In: 2015 International Conference on Computing Communication Control and Automation (IEEE Digital library), pp. 386–389 (2015)
26. DataStax Enterprise. http://www.datastax.com/what-we-offer/products-services/datastax-enterprise
27. Blake, C.L., Merz, C.J.: UCI repository of machine learning database. Department of Information and Computer Science, University of California, Irvine, CA (1998). http://www.ics.uci.edu/~mlearn/MLRepository.html
28. Sundmaeker, H., Guillemin, P., Friess, P., Woelfflé, S.: Vision and challenges for realizing the Internet of Things. In: CERP-IoT-Cluster of European Research Projects on the Internet of Things (2010)

Additional References

29. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R.: Hive-a petabyte scale data warehouse using Hadoop. In Proceedings of the IEEE 26th International Conference Data Engineering, pp. 996–1005 (2010)
30. Yang, C., Yen, C., Tan, C., Madden S.R.: Osprey: implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In: Proceedings of the IEEE 26th International Conference on Data Engineering, pp. 657–668 (2010)
31. Kaldewey, T., Shekita, E.J., Tata, S.,: Clydesdale: structured data processing on MapReduce. In Proceedings of the 15th International Conference on Extending Database Technology, New York, NY, USA, pp. 15–25 (2012)