USING HADOOP TO ACCELERATE THE ANALYSIS OF SEMICONDUCTOR-MANUFACTURING MONITORING DATA

By

WENJIE ZHANG

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2014
To my advisor,
Dr. Jose A. B. Fortes

To my parents,
Peisong Zhang and Jinzhi Qu
ACKNOWLEDGMENTS

First, I would like to express a special appreciation and thanks to my advisor, Dr. Jose A.B. Fortes for his support to my study and research towards the master’s degree, for his patience, encouragement and immense knowledge. His endless support and guidance enlighten my way of study, research and career. Without his help, this thesis cannot be finished. Second, I would like to thank the rest of my committee members Dr. Renato J Figueiredo and Dr. Tao Li for their patience, support, insightful comments and suggestions. Third, I wish to acknowledge the help provided by Mr. Youngsun Yu from Samsung for the insight discussion of the problem. Fourth, my sincere thanks to Dr. Andrea Matsunaga. Her enthusiasm, patience and encouragement of research and study guide me to the door of research.

In addition, a special thanks to my family and my friends for the support, understanding and encouragement. Words cannot express how grateful I am to my parents for all of the sacrifices that you’ve made on my behalf. At the end I would like express appreciation to Zhuoyuan Song for his support and help in finishing the thesis.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>12</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>13</td>
</tr>
<tr>
<td>1.1 MapReduce Programming Model and Hadoop Workflow</td>
<td>16</td>
</tr>
<tr>
<td>1.1.1 MapReduce Programming Model</td>
<td>17</td>
</tr>
<tr>
<td>1.1.2 Hadoop Platform</td>
<td>18</td>
</tr>
<tr>
<td>1.2 Motivation and Problem Statement</td>
<td>22</td>
</tr>
<tr>
<td>1.2.1 Data Mapping and Computation Distribution</td>
<td>23</td>
</tr>
<tr>
<td>1.2.2 System Optimization</td>
<td>24</td>
</tr>
<tr>
<td>1.3 System Architecture</td>
<td>24</td>
</tr>
<tr>
<td>1.4 Structure of Thesis</td>
<td>28</td>
</tr>
<tr>
<td>2 DESIGN AND IMPLEMENTATION</td>
<td>29</td>
</tr>
<tr>
<td>2.1 Data Description</td>
<td>29</td>
</tr>
<tr>
<td>2.2 Algorithm Analysis</td>
<td>31</td>
</tr>
<tr>
<td>2.2.1 Percentile and Outlier</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2 Process Capability</td>
<td>34</td>
</tr>
<tr>
<td>2.2.3 Euclidean Distance</td>
<td>35</td>
</tr>
<tr>
<td>2.2.4 Equivalence Test</td>
<td>36</td>
</tr>
<tr>
<td>2.3 Data Mapping</td>
<td>36</td>
</tr>
<tr>
<td>2.3.1 Data Mapping: Database to Files</td>
<td>36</td>
</tr>
<tr>
<td>2.3.2 Data Mapping: Files to Mappers</td>
<td>40</td>
</tr>
<tr>
<td>2.4 System Implementation</td>
<td>42</td>
</tr>
<tr>
<td>2.4.1 M-Heavy Implementation</td>
<td>43</td>
</tr>
<tr>
<td>2.4.1.1 Map implementation</td>
<td>43</td>
</tr>
<tr>
<td>2.4.1.2 Reduce implementation</td>
<td>43</td>
</tr>
<tr>
<td>2.4.2 R-Heavy Implementation</td>
<td>45</td>
</tr>
<tr>
<td>2.4.2.1 Map implementation</td>
<td>45</td>
</tr>
<tr>
<td>2.4.2.2 Reduce implementation</td>
<td>46</td>
</tr>
<tr>
<td>2.4.3 MR-Balanced Implementation</td>
<td>47</td>
</tr>
<tr>
<td>2.4.3.1 Map implementation</td>
<td>47</td>
</tr>
<tr>
<td>2.4.3.2 Reduce implementation</td>
<td>48</td>
</tr>
<tr>
<td>2.4.4 Relationships with Data Mapping</td>
<td>49</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>A vertical view of a sample record</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>How different MapReduce implementations constrain data mappings into the map/reduce phase</td>
<td>43</td>
</tr>
<tr>
<td>2-3</td>
<td>Relationships between different implementations and data mapping strategies</td>
<td>50</td>
</tr>
<tr>
<td>3-1</td>
<td>A subset of configuration parameters tuned for system optimization - part 1</td>
<td>52</td>
</tr>
<tr>
<td>3-2</td>
<td>A subset of configuration parameters tuned for system optimization - part 2</td>
<td>53</td>
</tr>
<tr>
<td>4-1</td>
<td>The default values of configuration parameters in Table 3-1 and 3-2</td>
<td>72</td>
</tr>
<tr>
<td>4-2</td>
<td>Experimental results of compress and speculative execution parameters</td>
<td>80</td>
</tr>
<tr>
<td>4-3</td>
<td>The values of configuration parameters after tuned</td>
<td>82</td>
</tr>
<tr>
<td>4-4</td>
<td>Comparison of averages of records calculated using the two-pass algorithm and the sliding-window approach for different numbers of records</td>
<td>84</td>
</tr>
<tr>
<td>4-5</td>
<td>Comparison of standard deviations of records calculated using the two-pass algorithm and the sliding-window approach for different numbers of records</td>
<td>84</td>
</tr>
<tr>
<td>5-1</td>
<td>The advantages and disadvantages of different implementations</td>
<td>93</td>
</tr>
<tr>
<td>Figure</td>
<td>page</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>1-1</td>
<td>Legacy system architecture for semiconductor manufacturing monitoring analysis system. The figure shows the architecture of a legacy analysis system. Data collected from monitoring sensors are stored in a Microsoft database. The analysis system retrieves data from the database and analyzes the data using multiple machines. Results of analysis are sent to an Oracle database for field engineers.</td>
<td>15</td>
</tr>
<tr>
<td>1-2</td>
<td>The MapReduce programming model</td>
<td>18</td>
</tr>
<tr>
<td>1-3</td>
<td>Hadoop MapReduce workflow</td>
<td>21</td>
</tr>
<tr>
<td>1-4</td>
<td>An overview of system architecture for the analysis system on Hadoop</td>
<td>23</td>
</tr>
<tr>
<td>1-5</td>
<td>A high level view of the statistics-based monitoring system and the statical optimization strategy</td>
<td>26</td>
</tr>
<tr>
<td>2-1</td>
<td>Data sets collected during several consecutive days are used for nightly statistical analysis.</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>Database-to-files mapping strategies - arbitrarily mapping</td>
<td>37</td>
</tr>
<tr>
<td>2-3</td>
<td>Database-to-files mapping strategies - per-iCU mapping</td>
<td>38</td>
</tr>
<tr>
<td>2-4</td>
<td>Database-to-files mapping strategies - per-dCU mapping</td>
<td>39</td>
</tr>
<tr>
<td>2-5</td>
<td>Files-to-Mappers mapping strategies</td>
<td>41</td>
</tr>
<tr>
<td>2-6</td>
<td>M-Heavy implementation: Map phase</td>
<td>44</td>
</tr>
<tr>
<td>2-7</td>
<td>R-Heavy implementation: Map phase</td>
<td>45</td>
</tr>
<tr>
<td>2-8</td>
<td>R-Heavy implementation: Reduce phase</td>
<td>46</td>
</tr>
<tr>
<td>2-9</td>
<td>MR-Balanced implementation: Map phase</td>
<td>48</td>
</tr>
<tr>
<td>2-10</td>
<td>MR-Balanced implementation: Reduce phase</td>
<td>49</td>
</tr>
<tr>
<td>3-1</td>
<td>A shuffling phase of a MapReduce job. Each map task contains a map function. The buffer associated with the map function is the place that stores the output from the map function. When the size of the data in the buffer exceeds some threshold, the data are split into disks. The blocks following the buffer in each map task represent the blocks written on the disk. Each reduce task contains a reduce function. Before the reduce function starts, the reduce task retrieves the output from map tasks and merges the outputs as the input to the reduce function, represented by the arrow and the merge streams in each reduce task.</td>
<td>55</td>
</tr>
<tr>
<td>3-2</td>
<td>The sliding window of statistical analysis. The window size is 14 days.</td>
<td>59</td>
</tr>
</tbody>
</table>
3-3 Data mapping strategy with regards to time stamp of each record

4-1 Different data mapping strategies for each implementations

4-2 Different implementations for each data mapping strategy.

4-3 Time measurement for current data simulation

4-4 Influence of the number of map tasks launched. The dots in the figure show the experimental results. Each dot contains three dimensions, the number of map tasks, the HDFS block size and the execution time. The execution time is shown using different shades of gray. The darker the area is, the less time the job takes to run.

4-5 Influence of the number of map tasks (evenly data distribution)

4-6 Influence of the number of reduce tasks on execution time

4-7 Influence of the buffer size of the output from the map function

4-8 Influence of the parameter io.sort.factor on the execution time

4-9 Influence of the parameter mapred.reduce.parallel.copies on the execution time

4-10 Influence of the reduce phase buffer size (percent). The stars in the figure indicate that the experiments cannot be finished due to the lack of memory space for the reduce task computation.

4-11 Influence of the reduce phase buffer size (records)

4-12 Influence of the parameter mapred.job.reuse.jvm.num.tasks on execution time

4-13 Influence of the parameter mapred.reduce.slowstart.completed.maps on execution time

4-14 Comparison of the execution time before and after tuning of configuration parameters to the R-Heavy implementation

4-15 The execution times of the two-pass algorithm and the sliding window approach to compute averages and standard deviations

4-16 The global-order strategy and the time-stamp-order strategy

4-17 Sorting and the standard deviation computation time

4-18 Computation time and I/O time measurement

4-19 The total execution time for four cases

4-20 Execution time for different cases

5-1 Different system architectures
A-1  The welcome page from CentOS image ................................. 99
A-2  Successfully set up the network connection .......................... 100
A-3  The content of the file 70-persistent-net.rules before updating .... 102
A-4  The content of the file 70-persistent-net.rules after updating ..... 102
A-5  The login web interface of the Cloudera Manager .................. 103
A-6  The web interface for user to specify the nodes on which the Hadoop will be installed .................................................. 104
A-7  The web interface for user to specify the installation package and other options . 104
A-8  The web interface for user to specify the appropriate credential method to access to the cluster ........................................... 105
A-9  The web interface when the installation fails .......................... 106
A-10 The web interface when the installation process goes well ........... 106
A-11 The web interface when installation of the package is successful .... 106
A-12 The web interface when installation of the package is successful .... 107
A-13 The web interface when all services has been installed and the cluster is ready to work .................................................... 107
A-14 The web interface showing the status of services available ........... 108
Large-scale data sets are generated every day by the sensors used to monitor semiconductor manufacturing processes. To effectively process these large data sets, a scalable, efficient and fault-tolerant system must be designed. This thesis investigates the suitability of Hadoop as the core processing engine of such a system. Hadoop has been used in many fields for large-scale data processing, including web-indexing, bioinformatics, data mining, etc, where it is proved to be an effective data processing platform because it provides parallelization and distribution for applications built on it. It also offers scalability and fault-tolerance using only commodity machines.

In this thesis, we propose a system for statistics-based analysis of manufacturing monitoring data using Hadoop. We first present a data-oriented approach through an in-depth analysis of the data to be processed. Then, different Hadoop configuration parameters are tuned in order to improve the performance of the system. In addition, the statistics-based computation algorithm is modified so that it can reuse previous results. By establishing a mapping from data to Hadoop MapReduce programming paradigm, the implemented system outperforms the legacy analysis system by up to 82.7%. Experiments have been done on different mapping strategies to study the effect of configuration parameters on performance. The algorithmic optimization improves the optimized computation time by 50%. However, the non-computation time, i.e. IO operation time, dominates execution time in our experiments.
Massive amounts of data are collected daily in manufacturing plants where sensors are used to track many physical parameters that characterize the operation of a large number of equipment units. The data need to be processed, analyzed, stored and organized in a way that it is easy for engineers to use them. Processing large-scale data sets usually takes a long time. Technologies, such as parallelization and distribution, are used for solving the “big-data analytics” problem. One such technology, Hadoop [2], is designed for large-scale data processing on clusters of computers. In this research, we present a statistics-based analysis system for semiconductor manufacturing monitoring data built on Hadoop. The choice of Hadoop is due to its scalability, efficiency, and transparent support of distributed, parallel and fault-tolerant computational tasks.

A semiconductor manufacturing plant consists of many Equipment Units (EUs) which can be used to produce a variety of semiconductor products. Different products are produced in different production lines, each production line consisting of multiple steps, each using one or more EUs. Sensors embedded in each EU are used to monitor multiple environment parameters of relevance to the quality of the final product. Monitoring every production line generates a large set of data each day. As the scale of semiconductor manufacturing plants grows, so does the scale of the monitoring data. As a consequence, a scalable system for analysis of monitoring data is essential to enable the optimization of semiconductor manufacturing processes.

As a representative example, this thesis consider a statistics-based analysis system that categorizes data collected from monitoring sensors according to different production lines and different manufacturing steps. In each category, collected data are further divided into different data sets based on EUs. The Student’s T-test [3] is used to determine if two data sets are significantly different from each other. T-test, or one of other related statistical hypothesis tests, has been widely used for statistical process
control in manufacturing procedures \cite{4, 5}. Results of the analysis offer field engineers information about malfunctioning EUs such that repairing those EUs takes place in a timely manner.

A common Semiconductor Manufacturing Monitoring Data Analysis System (SMMDAS) is presented in Figure 1-1 \cite{6}. Sensors are embedded in each EU for monitoring environment parameters. Data collected from sensors are preprocessed and stored in a database. The legacy statistics-based analysis system consists of multiple computers, and each computer contains multiple cores. Every night, the system retrieves the data to be analyzed from the database and assigns jobs to multiple computers. Each computer may receive a single job, or multiple jobs. Each core of the computer is responsible for processing an independent job at a time. No correlated processing is required between multiple cores, or multiple jobs. After processing jobs, computers send results to a different database.

Several problems exist in the current parallel system. First, there is no management block responsible for workload balancing, which results in resource underutilization. The parallel system consists of multiple homogeneous computers. However, different computers receive different workloads. As an example shown in Figure 1-1, the job list for Fab-line 2 is empty making the Data-Analytics-Server 2 idle while other servers are busy with a long list of jobs from other Fab-lines. Second, the analysis results are required by every morning for field engineers. However, the current system cannot always meet the deadline, resulting in delay in repairing problematic EUs. In addition, other considerations are also required for an efficient and effective analysis system. As the scale of manufacturing and monitoring procedure grows, the size of data to be analyzed increases as well. Due to problems of the current system and the growing scale of data, scalability is a requirement for a new system design. In addition, the scale of both data and the system is increasing such that failures appear more and more frequently. It is
necessary to design a system with fault-tolerance so that failures don’t influence the correctness of results and performance of the system.

In this chapter, an introduction of MapReduce programming model and its open-source implementation Hadoop are presented along with their corresponding advantages. Second, aiming to address limitations exposed by the legacy SMMDAS, a system built on Hadoop is introduced to achieve the scalability, performance and other desirable features. Third, optimization strategies, including tuning Hadoop configuration parameters and re-using of previous computation results, are discussed.
1.1 MapReduce Programming Model and Hadoop Workflow

A data processing system can be viewed as having three tiers: application, platform and storage. The storage tier handles data storage, replication, access, etc. The application tier supports multiple jobs, users and applications. The platform tier enables the use of multiple distributed resources and provides a programming and unified environment that hides from programmers how programs and data are parallelized, allocated and coordinated. Virtualization technologies enable the use of multiple homogeneous, or even heterogeneous machines, under a unified computing environment. The platform and storage tiers are tightly coupled, hiding the complexity of the underlying management and distribution. The Google MapReduce platform [7], with its corresponding storage system Google File System (GFS) [8], are representative examples of the platform and storage tiers. MapReduce is a programming model inspired by functional programming. Google adopted the programming model and its associated implementation, inheriting the name of MapReduce, for large-scale web index service [7]. With the support of its underlying storage system GFS, MapReduce is used for massive scale data processing in a distributed and parallel manner.

Hadoop [2] is an open-source platform consisting of two services - the MapReduce service and the Hadoop Distributed File System (HDFS) [9]. Hadoop was originally developed by Yahoo! and once it became open-source, it has been applied, analyzed and improved by many engineers and researchers. Many applications have been developed following the MapReduce programming model [1, 10] on Hadoop, in fields such as bio-medical informatics [11, 12], data mining [13, 14] and manufacturing data management [15]. Hadoop has also been widely used in scientific analytics for large-scale data processing [11, 12, 16]. Taylor [17] summarized current bioinformatics applications that use Hadoop. Papadimitriou and Sun [13] developed a distributed co-clustering framework based on Hadoop, and showed the scalability and efficiency provided by its underlying Hadoop platform. Cardona et al. [18] applied the MapReduce programming model to
a Grid-based data mining system, in which two improved schedulers are proposed and simulated.

In this section, we present background on the MapReduce programming model and the workflow of a Hadoop job in order to explain how Hadoop achieves parallelization and distribution.

1.1.1 MapReduce Programming Model

MapReduce programming model consists of two functions, *map* and *reduce*. The input to a MapReduce program is a set of *<key, value>* pairs, and the output of the program usually is another set of *<key, value>* pairs.

Figure 1-2 shows a diagram of the MapReduce programming model. The map phase consists of multiple Mappers - the same map function is carried out on each Mapper. Mappers are executed independently, meaning that there are no interactions among them. The *map* function takes a *<key, value>* pair as an input, and generates a set of intermediate *<key, value>* pairs as the output. There is no relationship between the input and output types. As shown in Figure 1-2, the input to the map phase is abstracted as *<K_i, V_j>* , and the output of the map phase is abstracted as *<k_x, v_y>* . The MapReduce programming model usually has a shuffling phase, in which the intermediate *<key, value>* pairs with the same key are sent to the same *reduce* function. Similar to the map phase, the reduce phase consists of multiple Reducers, and the same reduce function is carried out on each Reducer. The *reduce* function takes a list of values associated with the same key as the input, generates a set of *<key, value>* pairs as the output, and writes them into the file system. Similar to the map phase, the type of inputs can be different from the type of outputs.

Since both map and reduce functions are executed independently, an application following the MapReduce programming paradigm can be naturally distributed and

---

1 subscripts are used to identify pairs where keys and/or values are identical
The MapReduce programming model parallelized across a cluster of computers. Hadoop is a software system that provides a programming platform and run-time environment for MapReduce jobs. HDFS is the storage system associated with Hadoop - it provides distributed file system capabilities for large-scale data sets. MapReduce and HDFS services are tightly coupled such that users only need to program map/reduce functions as discussed above and Hadoop takes care of parallelization and distribution of computations and input data. A typical Hadoop MapReduce job workflow is discussed in the next section.

1.1.2 Hadoop Platform

In a typical Hadoop cluster, the MapReduce service relies on one master node and multiple slave nodes. The master node is responsible for scheduling, assigning map/reduce tasks to slave nodes, management of the cluster, etc. Slave nodes are machines that execute map/reduce tasks. Correspondingly, HDFS relies on one name node and multiple data nodes. The name node stores metadata, and application data are distributed among data nodes [9]. Both the name node of HDFS and the master node of the MapReduce
service use dedicated machines. A Hadoop MapReduce job workflow is described below (HDFS is considered as a transparent storage system that provides metadata information to the MapReduce service).

A JobTracker is a software component running on the master node and responsible for coordinating a job run. Correspondingly, a TaskTracker is a software component running on each slave node and responsible for running tasks of the job. The client program is written in Java in this example (otherwise, the Hadoop streaming utility is used for programs written in other programming languages) [1]. The client, or user program, submits a MapReduce job to a Hadoop cluster. The JobTracker receives the job information, assigns a unique job id to the submitted job, retrieves input file information from the underlying distributed storage system (for example, HDFS), and assigns map/reduce tasks to slave nodes according to the current status of slave nodes. Each TaskTracker communicates with the JobTracker and executes map/reduce tasks assigned to it. Each TaskTracker sends a “heartbeat” periodically to the JobTracker, as an indication of the health of the slave node. The heartbeat channel is also used for sending messages to the JobTracker. When a map/reduce slot is available in a slave node, the JobTracker assigns a map/reduce task to the slave node through the TaskTracker. The TaskTracker retrieves the input from distributed file system, and launches a child Java program to execute the task. After the task has finished, the TaskTracker collects the task information and sends the task information back to the JobTracker. The JobTracker notifies the client when the job has finished.

Figure 1-3 shows the workflow of a MapReduce job in Hadoop. The numbers 1 to 6 indicate the steps of a job execution. First, the client program submits the job to a Hadoop cluster. Second, the JobTracker retrieves file splits information, e.g. the location of files and the number of splits, from the distributed storage. For the third step, the heartbeat information is sent by the TaskTracker to the JobTracker to notify the JobTracker that there are available slots in a slave node. Fourth, the JobTracker
assigns a task to the TaskTracker based on the node information that the TaskTracker sent. Usually, the JobTracker assigns a map task to the slave node taking into account the location of input files. Fifth, the TaskTracker retrieves input information from the distributed storage. If it is assigned a map task, the TaskTracker retrieves the input file split information. If it is assigned a reduce task, the TaskTracker retrieves intermediate data generated by the map phase according to information provided by the JobTracker. Finally, in step six, the TaskTracker launches a new Java program to execute the assigned task. After the task has finished, the information is sent to the JobTracker through the heartbeat message channel.

An important property of Hadoop is fault-tolerance [1]. There are three kinds of failures in the workflow shown in Figure 1-3. One kind of failure is a task failure, i.e. the task executed in a child JVM of a slave node fails. If the TaskTracker detects the failure, it will free the slot and execute another task. The failure information is reported to the JobTracker through the heartbeat message channel. When the JobTracker receives the failure information, it will try to reschedule the task on a different node. The second kind of failure is a TaskTracker failure, i.e. when the JobTracker detects the failure of a TaskTracker (by not receiving heartbeat information or receiving the information too slow). The JobTracker stops scheduling tasks to that node and reschedule tasks on that node to other healthy nodes. The third kind of failure is a JobTracker failure. There is no mechanism to tolerate the JobTracker failure. YARN (or Hadoop 2.0) [19] introduces a new data center management structure that provides a mechanism to tolerate such a failure in a data center.

As discussed above, the Hadoop MapReduce service and its associated storage system HDFS enable parallelization and distribution among available resources. Multiple advantages are provided by the software platform. First, as the scale of the input increases, the Hadoop platform can easily scale up such that additional resources can
Figure 1-3. Hadoop MapReduce workflow

1. Run job
2. Retrieve input information from distributed storage
3. Heartbeat and execution results
4. Assign job to slave node
5. Retrieve input information for assigned jobs
6. Launch a map/reduce job

be added for larger scale data processing. As the resources in the cluster are underutilized, the platform can also easily scale down. Second, with regards to fault-tolerance, HDFS enables data durability through replications in different nodes, and the MapReduce service tolerates failures by speculative execution. Third, without needing to consider the complexity of underlying distributed system, Hadoop provides to users a simple programming interface. Fourth, different configuration parameters can be tuned for
different application requirements. Fifth, users can inherit, such as low cost and flexibility of using open-source software.

In summary, the Hadoop platform provides properties including scalability, fault-tolerance, easy-to-program interface, tunable configurations, benefits from open-source software, etc. It has been applied to run parallel algorithms in many fields for massive data processing \[1, 10\]. We next introduce the proposed SMMIDAS and discuss how the system benefits from the Hadoop platform.

### 1.2 Motivation and Problem Statement

Motivated by the previously discussed problems and requirements, we propose to build the analysis system on Hadoop. As discussed in Section 1.1, the Hadoop MapReduce service and HDFS provide management of underlying resources and hide the complexity of the distributed system from applications built on it. The properties of the Hadoop platform can be inherited by the analysis system so that it is capable of processing large-scale data sets. With respect to scalability, the system can easily scale up by adding more resources when the size of input data increases or the performance of applications decreases. With regards to fault-tolerance, through MapReduce speculative execution and HDFS replication, the system can obtain correct results even when there are failures in the cluster. The system will also rely on the open-source nature of Hadoop thus benefiting from low cost in both acquisition and upgrades. Thus, the system is proposed to be built using the Hadoop platform. The goal of this research is to design, implement, optimize and verify an effective and efficient distributed system for analysis of monitoring data. To this end, the system architecture will replace the current analysis system by a Hadoop system, as shown in Figure 1-4. Data collected from monitoring sensors are still stored in a database. The analysis system retrieves the data from the database to HDFS which is responsible for distributing data among multiple computers. The statistics-based analysis system is built on the Hadoop MapReduce service. After the analysis, the data are sent
back to a database. We propose a new system architecture without involving a database in Chapter 5.

Figure 1-4. An overview of system architecture for the analysis system on Hadoop

For purposes of building the statistics-based analysis system, the following problems must be considered.

1.2.1 Data Mapping and Computation Distribution

The purpose of the SMMDAS is to analyze the “data”. Thus, the input data are required to be studied in-depth so that they can be appropriately distributed among multiple machines. A thoughtless data mapping strategy can lead to unbalanced workload in a Hadoop cluster, which results in performance degradation and under-utilization of resources. To this end, the question of how to map the data into HDFS must be carefully answered.

Correspondingly, the computation must be mapped to the Hadoop MapReduce service as well. The computation can be finished in either one-step MapReduce workflow or multi-step MapReduce workflow (or iterative MapReduce [20]). Different mapping strategies have different advantages and disadvantages. For example, iterative applications
contain multiple steps of computation and each step is parallelized such that the data processing time required by each step is reduced. Iterative applications on Hadoop, however, generate new map/reduce tasks in each iteration, resulting in overhead for creating new tasks for each iteration. Ekanayake et al. [20] improve the solution by introducing Twister, an extension of the Hadoop platform, for iterative scientific applications. Thus, it is important to design an appropriate computation mapping strategy for the SMMDAS. Other extensions of the Hadoop software platform may also be adopted for performance purposes.

1.2.2 System Optimization

After the system is deployed, optimization of the system will benefit both the performance of the statistics-based analysis and the utilization of the whole computation cluster. Both external and internal improvements are required. From the external perspective, optimization can be achieved by tuning different configuration parameters. Applications are significantly influenced by different configuration settings [21]. Changing parameters to appropriate values is shown to significantly improve performance of applications. From the internal perspective, optimization can be achieved by changing the algorithm of the analysis. Through algorithmic optimization, previous computation results can be used for future data analysis.

According to the problems discussed in this section, we present this research in a high-level view in the following section.

1.3 System Architecture

Figure 1-5 shows a high-level view of the statistics-based analysis system using Hadoop (bottom) and strategies to statically optimize the system (top).

The bottom of Figure 1-5 solves the problem of data mapping and computation distribution. Since data are a critical design consideration, we propose a data-oriented mapping from the database to the Hadoop MapReduce paradigm. This mapping consists of two steps, mapping the database to files and mapping files to Hadoop. A one-step
MapReduce workflow is proposed for mapping the computation to the MapReduce service. Based on different mapping strategies, we deploy the statistics-based analysis system in three different ways: Map-Heavy, MapReduce-Balanced and Reduce-Heavy.

The top of Figure 1-5 solves the problem of system optimization. MAPE-K (Monitoring, Analysis, Planning, Execution and Knowledge) can be used as a self-management architecture in autonomic computing [22]. However, in this research, we do not consider dynamic optimization but follow the basic idea the MAPE-K approach to statically optimize the analysis system. As shown in Figure 1-5, the Monitoring block collects metrics of the system, such as the CPU/memory utilization, input data and the execution time; the Analysis block provides adjustment suggestions by analyzing different performance metrics with regards to the knowledge of the system execution history or new policies; the Planning block produces a series of changes based on the suggestions from the Analysis process; the Execution block carries out adjustments to the system. Two methodologies are used in this research for system optimization. From an external point of view, different Hadoop configuration parameters are tuned for decreasing the execution time. The Monitoring block collects the current execution time of the analysis. The Analysis block compares the current execution time with historic execution time and some well-known tuning directions. It suggests the tuning directions of the configuration parameters. The Planning block generates target values of the configuration parameters to be tuned. The Execution block carries out the tuning actions. From an internal point of view, the algorithm described in Chapter 2 is optimized for decreasing the execution time. A dynamic procedure that follows MAPE-K would be similar to tuning the configuration parameters as discussed above.

Works have been done for applying Hadoop in manufacturing procedures. Bao et al. [15] applied Hadoop on the sensor data management in manufacturing. They focused on the data management perspective and proposed to replace the RDBMS with HBase [23]. Most computations used in their system are simple queries or statistics operations. Our
work focuses on a complex statistics-based computation, which cannot be finished within a database. The use of HBase or Hive may be an interesting design for the proposed architecture without involving the database in Chapter 5. To serve the increasing scale of the data, Goss and Veeramuthu [24] surveyed current technologies towards the “Big Data” problem for a new semiconductor manufacturing company, including technologies such as Appliance RDBMS, Hadoop and in-memory DB, as well as new architectures, for example, General Engineering and Manufacturing Data Warehouse (GEM-D) solution. They compared advantages versus disadvantages of possible technologies without implementing them. We have taken a further step in this research by designing and implementing
the system using Hadoop. Experimental evaluation results, as well as two optimization methodologies, are also presented in this thesis.

Most Hadoop applications are I/O bounded [1]. Hadoop tries to achieve “data locality”, i.e. to assign tasks to where the data are located. Multiple studies have researched this data-locality property and suggested methods to manage task assignments. Liu et al. [25] studied the performance of HDFS, especially, for processing small files. For improving the I/O performance of small files, they proposed a grouping methodology to combine small files. Our experimental results verify the observation and take advantage of combining small files for performance improvement. Xie et al. [26] discussed the effect of data placement in a heterogeneous Hadoop cluster. An imbalanced data placement in heterogeneous environment causes severe performance degradation in data-intensive Hadoop applications. By implementing a data placement mechanism in HDFS, input data can be distributed based on the computing capacity of each node. We use the balancer provided by Hadoop. Since our cluster is built on a homogeneous environment as a private cloud, the balancer works well in the current setup. Also, we found out that increasing the number of replications for each file in HDFS results in increasing the possibility of execution on local data. For purposes of saving energy and I/O efficiency, the intermediate data generated by the map phase can be compressed. Chen et al. [27] proposed an algorithm to measure data characteristics so that users of Hadoop can make the decision of whether to compress the intermediate data or not. It is better to compress the map output in our system. Data locality is used in Hadoop to avoid slow tasks by reducing network transmission [28]. However, scheduling tasks with regards to the data location cannot necessarily improve the performance of a Hadoop job. If map tasks are scheduled on slow nodes on the basis of data locality, the performance of a job is possibly degraded. Again, the current system is built on a homogeneous environment. Such works may be useful when using a heterogeneous environment.
1.4 Structure of Thesis

This thesis is structured as follows: Chapter 2 describes the input data and a data-oriented design by introducing different data mapping strategies and different implementations. Chapter 3 proposes different strategies for system optimization including tuning Hadoop configuration parameters and algorithmic optimization. Chapter 4 discusses experimental results. Chapter 5 concludes the thesis and recommends future research directions.
CHAPTER 2
DESIGN AND IMPLEMENTATION

The system studied in this thesis for statistical analysis of manufacturing sensor measurements is implemented by using Hadoop to do all the data processing. The design of this system requires a good understanding of what kind of data is to be analyzed, how they are to be analyzed and where they should be located. This chapter answers these issues and describes three implementations considering different data mapping scenarios. A data-oriented design approach is used to map data from the database to the Hadoop storage. This chapter characterizes the data that are to be analyzed, the algorithm is used to analyze them, mapping strategies from the database to HDFS as well as three implementations of the mapping of computations to the Hadoop map/reduce phase.

2.1 Data Description

The input data can be considered as a large table stored in a database, where each row is a measurement record that includes when and where the measurement took place as well as the measured “item”, product and equipment identifications. Table 2-1 is a vertical view of an example of a record from a sample of a typical data set.

Useful information for statistical analysis includes ITEM (environment parameter, such as temperature, pressure and gas flow rate), RECIPE (product and manufacturing step information), EQP_ID (EU information), TIME (start and end time of the measurement), SPEC (upper and lower specifications of the measured record) and AVG (the average of the measured value from start to end time, i.e. the information to be analyzed).

The monitoring system collects data during the manufacturing process. Every night, the analysis system retrieves data from 14 days (the current day and the previous 13 days), based on the time stamp associated with each record as shown in Figure 2-1. Assume the current date is 03/27/14, the statistics-based analysis is performed on the data from 03/14/14 to 03/27/14. On the second day, 03/28/14, the statistics-based analysis is performed on the data from 03/15/14 to 03/28/14. We discuss possibilities to
Table 2-1. A vertical view of a sample record

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Content</th>
<th>Detail Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOT_ID</td>
<td>LOT0759.1</td>
<td>Record identifier</td>
</tr>
<tr>
<td>WAFER_ID</td>
<td>01</td>
<td>Wafer ID</td>
</tr>
<tr>
<td>PRODUCT_ID</td>
<td>A5</td>
<td>Product ID</td>
</tr>
<tr>
<td>STEP_SEQ</td>
<td>A5_001</td>
<td>Sequence step</td>
</tr>
<tr>
<td>EQP_MODEL</td>
<td>DIFFUSION</td>
<td></td>
</tr>
<tr>
<td>EQP_ID</td>
<td>DIFF01</td>
<td>Equipment unit ID</td>
</tr>
<tr>
<td>CHAMBER_ID</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>RECIPE</td>
<td>A5_001_R01</td>
<td>Information of wafer, product and an ID</td>
</tr>
<tr>
<td>START_TIME</td>
<td>22:26.5</td>
<td></td>
</tr>
<tr>
<td>END_TIME</td>
<td>47:26.5</td>
<td></td>
</tr>
<tr>
<td>ITEM</td>
<td>Gas_Flow_Rate</td>
<td>The environment variable to be measured</td>
</tr>
<tr>
<td>MIN</td>
<td>100.4286</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>105.7051</td>
<td></td>
</tr>
<tr>
<td>AVG</td>
<td>106.7414</td>
<td>The average value to be analyzed</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.56929</td>
<td></td>
</tr>
<tr>
<td>UPPER_SPEC</td>
<td>109</td>
<td>The upper specification</td>
</tr>
<tr>
<td>LOWER_SPEC</td>
<td>101</td>
<td>The lower specification</td>
</tr>
<tr>
<td>TARGET</td>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>

use time stamp information and re-use of results for optimizing the analysis procedure in Chapter 3.

Figure 2-1. Data sets collected during several consecutive days are used for nightly statistical analysis.

For purposes of statistical analysis, the input data table can be partitioned into independent subtables, one for each recipe-and-item combination. These subtables are independent in the sense that the statistical analysis is done individually and independently for each subtable. Each subtable is referred to as an independent
Computation Unit (iCU). Each subtable is further partitioned into dependent subtables, one for each EU. Since subtables belonging to the same iCU have to be grouped together to finish the computation, each dependent subtable is referred to as a dependent Computation Unit (dCU). Let $K_r$ denote a combination of recipe and item, $K_e$ an equipment id and $K_t$ a combination of recipe, item and equipment id.

\[
K_r \leftarrow <\text{recipe, item}>
\]

\[
K_e \leftarrow <\text{equipment id}>
\]

\[
K_t \leftarrow <\text{recipe, item, equipment id}>
\]

Thus, the whole statistical analysis is independently performed on each iCU and it consists of partial related analysis done on each of the dCUs that make up an entire iCU.

### 2.2 Algorithm Analysis

The statistical analysis process is referred to as the “equivalence test”. The purpose of this analysis is to characterize the difference between two sets of data, a reference data set and a test data set, using Student’s T-test \[3\].

Algorithm 1 describes the analysis procedure. The input of the procedure is a linked list ($inputList$). Since the analysis is performed independently on each iCU, the data in the same $inputList$ share the same $K_r$. The $i$th element in the list is a 1D double precision array, $A_i$, representing a dCU. Each value in $A_i$ corresponds to an AVG value extracted from the records that share the same $K_t$.

For purposes of mapping the analysis algorithm to the MapReduce programming model later, the algorithm can be considered consisting of two stages. The first stage of the procedure sorts an array $A_i$ and then computes several percentile \[29\] values from $A_i$. The lower outlier limit (LOL) and upper outlier limit (UOL) are computed using the percentile values computed in the previous step. Outliers \[30\], either extremely small or extremely large values, are eliminated from $A_i$. Then, the procedure uses the average $\mu_i$.
Algorithm 1 Procedure for equivalence test

**Input:** LinkedList<Double[][]> inputList

for each DoubleArray $A_i$ in inputList do
    Sort $A_i$
    Compute the percentile values of $A_i$
    Compute the lower and upper outlier limits LOL, UOL using percentile values
    Eliminate all $A_i[k]$ that are not in the range [LOL, UOL]
    Compute average $\mu_i$ and standard deviation $\sigma_i$ of entries of $A_i$
    Compute the process capability index $C_{pk}[i]$
end for

Stage 1

Pick $A_i$ with the largest $C_{pk}$ as the reference set $R$
Compute Euclidean distance $W_R[k] = \sqrt{(R[k] - \mu_R)^2}$
Compute Euclidean distance $B_R[k] = \sqrt{(R[k] - \mu)^2}$
Compute $\mu_{W_R}, \sigma_{W_R}$ and $\mu_{B_R}, \sigma_{B_R}$

for each DoubleArray $A_i$ in inputList do
    Compute Euclidean distance $W_i[k] = \sqrt{(A_i[k] - \mu_i)^2}$
    Compute Euclidean distance $B_i[k] = \sqrt{(A_i[k] - \mu)^2}$
    Compute $\mu_{W_i}, \sigma_{W_i}$ and $\mu_{B_i}, \sigma_{B_i}$
    Do a pairwise Student’s t-test between $W_R$ (or $B_R$) and $W_i$ (or $B_i$)
    Final decision based on the result of t-test
end for

Stage 2

and the standard deviation $\sigma_i$ of $A_i$ to compute the process capability index [31]. This procedure is done for every $A_i$ in the input list independently.

The second stage of the procedure picks the $A_i$ whose $C_{pk}$ is the largest from the inputList as the reference data set, denoted as $R$. Other data sets are referred to as testing data sets. Each testing data set in the inputList is preprocessed by computing the Euclidean distance between the reference data set and itself. Then, a pair-wise Student’s T-test is carried out between the reference data set and each of the testing data sets. Final decisions (on whether the testing data set is equivalent to the reference data set) are determined by the results of T-test [4]. In summary, the first stage computation is performed on each dCU independently, while the second stage computation is performed on each iCU independently.
The followings sections describe the functions in Algorithm 1 in detail¹, providing information useful for algorithmic optimization of the statistics analysis in Chapter 3.

2.2.1 Percentile and Outlier

In statistics, an outlier is referred to as a data element that is extremely distinct from other data in the sample [30]. Outliers usually indicate problems affecting data reliability, for example, an error during measurement. To identify an outlier requires an upper limit and a lower limit, i.e. boundaries that determine excluded data. To compute the outlier limits, a percentile function is computed on the list of sample data. A percentile is a value below which proportions of values in a group fall into [29]. For example, the pth percentile is a value, \( V_p \). There are at most \( (p)\% \) data less than \( V_p \), and at most \( (100 - p)\% \) data greater than \( V_p \). The 50th percentile is the median value of a data set. The percentile value can be estimated using function \text{Percentile} \text{ as follows [32]. Note that } L \text{ is a 1D sorted list (or an array).}

\textbf{function} \text{Percentile(data} \ L, \text{ double} \ p) \text{)

\begin{align*}
L : & \text{sorted 1D list (or an array)} \\
N : & \text{the size of the list} \ L \\
p : & \text{the percentile} \\
r = & \frac{p}{100} \times (N + 1) \\
k \leftarrow & \text{integer part of} \ r \\
d \leftarrow & \text{fractional part of} \ r \\
V = & L_k + d \times (L_{k+1} - L_k)
\end{align*} \\
\textbf{end function}

¹ The description doesn’t reflect the specific implementation used in our experimental system. Instead, it presents general approaches described in common engineering statistics books and materials of which the implementation is a representative example.
The Percentile function first computes the rank, \( r \), of the input percentile \( p \). Then, it uses the entries with ranks (i.e. indexes) \( k \) and \( k + 1 \), where \( k \) is the integer part of the computed rank, \( r \), to compute the percentile value, \( V \). The result, \( V \), is computed by multiplying the fractional part of the computed rank and the difference between the \((k + 1)\)th and \(k\)th entries. The 25th percentile is referred to as first quartile, denoted as \( Q_1 \) and the 75th percentile is referred to as third quartile, denoted as \( Q_3 \). The interquartile range, \( IQR \), is the difference between \( Q_3 \) and \( Q_1 \),

\[
IQR = Q_3 - Q_1.
\]  

The outlier limits are estimated as

\[
UOL(\text{Upper Outlier Limit}) = Q_3 + c \times IQR, \tag{2-2}
\]

\[
LOL(\text{Lower Outlier Limit}) = Q_1 - c \times IQR, \tag{2-3}
\]

where \( c \) is a constant. A typical value of \( c \) is 1.5. Any data outside this range are considered as outliers. Outliers are eliminated in the first stage of the Algorithm 1.

2.2.2 Process Capability

“A process is a combination of tools, materials, methods and people engaged in producing a measurable output; for example a manufacturing line for machine parts.”[31] Process capability is a measurable property of a process, which serves for measuring the variability of the output and comparing the variability with certain specifications. “The process capability index is a statistical measurement of the process capability: the ability of a process to produce output within specification limits.”[31] The process performance index is one of the commonly accepted process capability indices, which is defined and computed as \( C_{pk} \) in Algorithm 1 and Equation (2–4) below.

Given a data set – the output from the system that monitors the process – and its corresponding upper specification (USL), lower specification (LSL), the process
performance index is defined and computed as

\[ C_{pk} = \min\left\{ \frac{USL - \mu}{3 \cdot \sigma}, \frac{\mu - LSL}{3 \cdot \sigma} \right\}, \tag{2-4} \]

where \( \mu \) is the average of the data set and \( \sigma \) is the standard deviation of the data set.

Another metric that is used to estimate the process capability is defined as

\[ \hat{C} = \frac{USL - LSL}{6 \cdot \sigma}. \tag{2-5} \]

This metric is not used in our prototypes, but they are easily modifiable to use such a metric if necessary.

2.2.3 Euclidean Distance

After the elimination of outliers from the sample data, data sets that come from different EUs but share the same \( K_r \) are collected together. The data set with the maximum process performance index is picked as the reference data set, \( R \). Each of the data sets of other equipment units, \( A_i \), is paired with the reference data set. For each element in each data set, Within-Distance, \( W_i[k] \), is defined as the Euclidean distance between a single record of the data set and the average of the data set. For each element in the reference data set, Between-Distance, \( B_R[k] \), is defined as the Euclidean distance between a single record of the reference data set and the average of all records in all data sets except the reference one. For each element in each data set except the reference data set, Between-Distance, \( B_i[k] \), is defined as the Euclidean distance between a single record of the data set and the average of all records in the reference data set. The equations to compute the Euclidean distances are listed below.

For the reference data set,

\[ W_R[k] = \sqrt{(R[k] - \mu_R)^2}, \tag{2-6} \]

\[ B_R[k] = \sqrt{(R[k] - \mu)^2}, \tag{2-7} \]
where $R[k]$ refers to the $k$th element in the reference data set, $\mu_R$ refers to the average of the reference data set and $\mu$ refers to the average of all records except those from the reference data set.

For other testing data sets,

\[
W_i[k] = \sqrt{(A_i[k] - \mu_i)^2},
\]

\[
B_i[k] = \sqrt{(A_i[k] - \mu_R)^2},
\]

where $A_i[k]$ refers to the $k$th element in the $i$th data set and $\mu_i$ refers to the average of the $i$th data set.

**2.2.4 Equivalence Test**

The equivalence test is performed on the previously computed Euclidean distances of each of the data sets and the reference data set.

The core of the equivalence test is Student’s T-test \([4]\), which is used to determine whether two sets of data are significantly different from each other. A numerical algorithm to compute the p-value \([4]\), a measurable result of T-test, can be found in \([33]\).

**2.3 Data Mapping**

To minimize the overhead of data communication among nodes during the map phase, Hadoop is designed so that it assigns map tasks to nodes where the input data are located. Thus, appropriately mapping data from the database to Hadoop has significant influence on the system performance.

Hadoop applications require the support from underlying HDFS. Data are stored as files in HDFS. Thus, mapping data to Hadoop consists of two stages: the database to files and files to Mappers.

**2.3.1 Data Mapping: Database to Files**

According to the data and the algorithm description, the whole data set can be partitioned into iCUs or dCUs. Based on how the data are partitioned (i.e. assigned to
HDFS blocks), we propose three strategies: arbitrarily mapping, per-iCU mapping and per-dCU mapping.

Arbitrarily mapping maps the data to files arbitrarily (i.e. in the order in which they are read from the database), shown in Figure 2-2. Therefore, records belonging to different iCUs and dCUs may be stored in the same file. The advantage of this strategy is that data can be distributed evenly across the cluster.

![Diagram](image-url)

Figure 2-2. Database-to-files mapping strategies - arbitrarily mapping
Figure 2-3. Database-to-files mapping strategies - per-iCU mapping.
Figure 2-4. Database-to-files mapping strategies - per-dCU mapping
Per-iCU mapping maps the data in the same iCU to the same file, shown in Figure 2-3. Records share the same $K_r$ are mapped to the same file. Since all records in the same iCU are stored in a file, a single file can be analyzed independently without interference with other files.

Per-dCU mapping maps the data in the same dCU to the same file, shown in Figure 2-4. Records share the same $K_t$ ($K_r$ and $K_e$) are mapped to the same file. All records in the same dCU are stored in a file, and part of the analysis (Stage 1 of Algorithm 1) can be performed independently on a file without interfering with other files.

2.3.2 Data Mapping: Files to Mappers

The data to be analyzed are stored as files in HDFS. As introduced in Chapter 1, the Hadoop MapReduce service provides parallelization and distribution to applications running on it. Files are distributed among nodes in a cluster. The Hadoop platform provides users flexibility to split files so that users can decide whether to split a file and how to split a file. Based on how many files a Mapper processes, we also propose three strategies to map files to Mappers: one-to-one mapping, many-to-one mapping and one-to-many mapping.

One-to-one mapping maps one file to one Mapper, as shown in Figure 2-5-a. Each Mapper handles a file independently. This can be controlled by overwriting `isSplitable()` method to let it return false. In a situation that a file contains indivisible data, in other words, the data have to be processed by a single Mapper independently, this mapping strategy works properly. On the other hand, the disadvantage of the strategy is that when a file is stored in different nodes, processing the file as a whole brings data transmission overhead in the map phase execution.

Many-to-one mapping maps multiple files to one Mapper, as shown in Figure 2-5-b. Each Mapper handles multiple files. Hadoop provides a class `CombineInputFormat` that combines several small files to be processed by a map task. We consider an alternative approach — a preprocessing procedure to combine multiple files to one file and each
A One-to-one mapping  

B Many-to-one mapping  

C One-to-many mapping  

Figure 2-5. Files-to-Mappers mapping strategies
Mapper takes the combined file as the input. The preprocessing approach alleviates HDFS meta-data storage and, also, reduces file processing time during a MapReduce job execution. In the case that the size of each file is relatively small, assigning each file to one Mapper creates much overhead. Thus, assigning multiple files to a Mapper is a proper strategy.

One-to-many mapping maps one file to multiple Mappers, as shown in Figure 2-5-c. Each Mapper handles part of a file. This is the most common situation in a Hadoop application. When a file is too large, HDFS splits the file into relatively smaller chunks and Hadoop assigns each chunk to a Mapper during the job execution. The split size can also be set larger than the block size of HDFS [1]. However, this is not recommended since it decreases the possibility of processing data locally. The one-to-many strategy doesn’t work in the scenario when there are dependencies between computations performed on different chunks.

2.4 System Implementation

As illustrated in Chapter 1, the structure of a Hadoop application follows the MapReduce paradigm. To distribute the computation, the statistics-based analysis is mapped to the MapReduce programming model. In the current SMMDAS, the size of data belonging to the same iCU is relatively small so that each iCU can be processed using one computing node in a Hadoop cluster. Thus, we propose to map Algorithm 1 to a single step MapReduce job. We discuss the scenario that the size of each dCU and each iCU increases in Chapter 4. Algorithm 1 consists of two main stages. The first stage can be computed for each dCU independently. The second stage processes data belonging to the same iCU. Based on how much computation carried out in the map/reduce phase, we propose three implementations: Map-Heavy (M-Heavy) implementation, Reduce-Heavy (R-Heavy) implementation and MapReduce-Balanced (MR-Balanced) implementation. Table 2-2 summarizes how different MapReduce implementations constrain data mappings into the map/reduce phase (the blank one means no specific constraints).
Table 2-2. How different MapReduce implementations constrain data mappings into the map/reduce phase

<table>
<thead>
<tr>
<th>phase</th>
<th>M-Heavy</th>
<th>R-Heavy</th>
<th>MR-Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>map phase</td>
<td>iCU</td>
<td>iCU</td>
<td>dCU</td>
</tr>
<tr>
<td>reduce phase</td>
<td>iCU</td>
<td>iCU</td>
<td>iCU</td>
</tr>
</tbody>
</table>

2.4.1 M-Heavy Implementation

The M-Heavy implementation maps the whole analysis procedure to the map phase. Thus, data belonging to the same iCU have to be collected and sent to the same Mapper. This benefits the case when the size of each iCU is relatively small.

2.4.1.1 Map implementation

In the M-Heavy implementation, the map phase is responsible for the whole analysis procedure. Data belonging to the same iCU have to be read by the same Mapper. The file split flag is set to false so that a file cannot be split between multiple Mappers. Figure 2-6 is the computation procedure and the data format/definition for each step in the map phase implementation.

The input to a Mapper is a <key, value> pair, where the key is the filename and the value is the whole content of the file. The Mapper extracts useful information from each record and stores them in a nested hash map, \( m \). In the case that data belonging to multiple iCUs are contained in the same file (many-to-one data mapping), each entry in the out layer hash map is an iCU, and each entry in the inner layer hash map is a dCU. Thus, the Stage 1 of Algorithm 1 is performed on each entry of the inner layer hash map independently, and the Stage 2 of Algorithm 1 is performed on each entry of the out layer hash map independently.

2.4.1.2 Reduce implementation

Since the whole analysis computation is done in the map phase, the reduce phase is unnecessary and deprecated.
Figure 2-6. M-Heavy implementation: Map phase
2.4.2 R-Heavy Implementation

The R-Heavy implementation maps the whole analysis procedure to the reduce phase, and Mappers filter the data by different $K_r$s. In this implementation, there is no limitation on the input data.

2.4.2.1 Map implementation

Like a typical Hadoop application, the R-Heavy implementation filters (or categorizes) the data during the map phase. Each Mapper reads a block of data, a whole file or part of a file, depending on the block size of HDFS and file splitting configuration settings. The input of the map function is a $<key, value>$ pair, where the key is the offset of a record and the value is the record. Figure 2-7 shows the computation procedure and the data format/definition of the map phase. The Mapper takes a record and extracts useful information from the record, especially ICU information. The output of the Mapper is a $<key, value>$ pair, where the key is the ICU information and the value is the useful record information, including the data to be analyzed.

![Computation Procedure Diagram](image)

Figure 2-7. R-Heavy implementation: Map phase
2.4.2.2 Reduce implementation

After the map phase has been finished, `<key, value>` pairs whose keys are the same are sent to the same Reducer. In this case, records from the same iCU are sent to the same Reducer. The Reducer takes a list of values generated by the map phase that share the same iCU as the input, and performs Algorithm 1 on each values in the same iCU. Figure 2-8 shows the computation procedure and the data format/definition in the reduce phase for the R-Heavy implementation.
2.4.3 MR-Balanced Implementation

The MR-Balanced implementation maps the first-stage computation to the map phase and the second-stage computation to the reduce phase. Since the first-stage computation requires that the data are grouped by dCUs, data belonging to the same dCU have to be collected and sent to the same Mapper before the computation starts. Mappers send results belonging to the same iCU to the same Reducer. This implementation benefits for the case that the size of each dCU is relatively large.

2.4.3.1 Map implementation

Since the first-stage computation of Algorithm 1 is performed in the map phase, records belonging to the same dCU have to be sent to the same Mapper. Thus, each Mapper has to read the whole file in case that the file is split. The input to the Mapper is a <key, value> pair where the key is a file’s name and the value is the content of the file. Figure 2-9 shows the implementation of the map function in each Mapper. The Mapper pulls contents of a file into memory and organizes single records by different dCUs. Then the Mapper computes percentiles, eliminates outliers and calculates the process capability for each list of values that belong to the same dCU. The result of computations is stored in an object, referred to as InterData, which can be transferred between Mappers and Reducers. An example of an instance of the InterData object is shown as follows.

InterData is an object representing a list of AVG values belonging to the same dCU. Following is an example of an instance of InterData. It implements Serializable interface so that it can be transmitted between map and reduce phases, which is useful for the MR-Balanced implementation.

```
23976  # table size
97.5395 97.5483 97.5485 ...  # the data table
0.3850  # process performance index, $P$
0.4121  # process capability index $\hat{P}$
A5_001_R01Gas_Flow_Rate, DIFF06  # RECIPE + ITEM, EQP_ID
```
The output of each Mapper is a set of <key, value> pairs where the key is \( K_r \) of the iCU and the value is an InterData object. The data belonging to the same iCU are grouped together and sent to Reducers. The default Partitioner, HashPartitioner, is used to distribute outputs of Mappers to the Reducers based on the hash value of the keys.

Figure 2-9. MR-Balanced implementation: Map phase

### 2.4.3.2 Reduce implementation

After the map phase has finished, the Reducer receives outputs from Mappers. Each Reducer receives a list of InterData objects with the same \( K_r \) (data belong to the same iCU).

The input to the Reducer is a value list associated with the same key, where each value corresponds to an InterData object. Figure 2-10 shows the implementation of the

---

**Computation Procedure**

1. **Input file**
   - Read a whole file through a record reader

2. **<key, value> pairs**
   - Split the file into lines. Extract <key, value> pairs.
   - Store <key, value> pairs into a HashMap<String, ArrayList>
   - Loop the HashMap and do analysis to each elements of the HashMap by transferring ArrayList to double array and eliminate outliers
   - Analysis result is constructed as a <key, value> pair and send to Reducer

**Data Format/Definition**

- The whole content as a String
- <key, value> pairs
  - Key: ITEM+RECIPE+EQP_ID
  - Value: AVG
- map<String, ArrayList>
  - String: ITEM+RECIPE+EQP_ID
  - ArrayList: A list of AVGs
- map<String, ArrayList>
  - String: ITEM+RECIPE+EQP_ID
  - ArrayList: A list of AVGs within outlier limits
- Key: ITEM + RECIPE
  - Value: InterData (consisting a list of AVGs)

**Intermediate outputs**
Reducer in the MR-Balanced implementation. The Reducer loops the value list and picks the InterData that contains the maximum process performance index, $C_{pk}$, as the reference data set. For other InterData objects, the Reducer computes Euclidean distances of the data, both Within-Distance and Between-Distance, does an equivalent test using Euclidean distances, and generates the final decision. Outputs of the Reducer are $<\text{key, value}>$ pairs where the key consists of RECIPE, ITEM and EQP_ID ($K_t$) and the value is the final decision, e.g. Equivalent, meaning that the data set $A_i$ is comparable to the reference data set and its associated EU is healthy.

**Figure 2-10.** MR-Balanced implementation: Reduce phase

### 2.4.4 Relationships with Data Mapping

Table 2-3 shows the relationship between different implementations and different strategies of data mapping. The R-Heavy implementation has no limitation on data
mapping strategy while other two implementations both have requirements for data mapping strategies. Thus, the R-Heavy implementation works for all data-mapping strategies. As discussed in Chapter 1, for experiments purposes, the database is kept functioning. If the future architecture removes the database, the R-Heavy implementation is the only solution that is capable of functioning without assistance from the database or preprocessing the data. The map phase of the R-Heavy implementation groups the data according to iCU or dCU information of each record so that no grouping from the database or preprocessing the data is required prior to the computation. The MR-Balanced implementation works in per-iCU one-to-one and many-to-one strategies as well as per-dCU one-to-one and many-to-one strategies. The M-Heavy implementation only works in per-iCU one-to-one and many-to-one strategies. Both M-Heavy and MR-Balanced implementations are constrained by specific data mapping strategies. The input data to these two implementations require either an assistance from the database or a preprocessing block that categorizes the data according to different $K_r$s and $K_t$s.

Table 2-3. Relationships between different implementations and data mapping strategies

<table>
<thead>
<tr>
<th>Implementations</th>
<th>One-to-one</th>
<th>Many-to-one</th>
<th>One-to-many</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrarily</td>
<td>R-Heavy</td>
<td>R-Heavy</td>
<td>R-Heavy</td>
</tr>
<tr>
<td>Per-iCU</td>
<td>R-Heavy,</td>
<td>R-Heavy,</td>
<td>R-Heavy</td>
</tr>
<tr>
<td></td>
<td>M-Heavy,</td>
<td>M-Heavy,</td>
<td>R-Heavy</td>
</tr>
<tr>
<td>MR-Balanced</td>
<td>MR-Balanced</td>
<td>MR-Balanced</td>
<td></td>
</tr>
<tr>
<td>Per-dCU</td>
<td>R-Heavy,</td>
<td>R-Heavy,</td>
<td>R-Heavy</td>
</tr>
<tr>
<td></td>
<td>MR-Balanced</td>
<td>MR-Balanced</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 3
SYSTEM OPTIMIZATION

We have conducted an in-depth study of the data to be analyzed and proposed different data mapping strategies. Based on the data mapping strategies and the computations performed on the data, three implementations are introduced to map the statistics-based analysis to the Hadoop platform. In this chapter, we try to optimize the system from two perspectives: tuning Hadoop configuration parameters and optimizing the analysis algorithm by factoring computations when possible so that results can be re-used and computations avoided.

3.1 Hadoop Configuration Parameters Tuning

To achieve timely and scalable computation, Hadoop configuration parameters need to be tuned for each specific application. Studies [21, 34] show that the performance of a Hadoop application suffers severely if inappropriate configuration parameters have been set. Many in-depth works [21, 35, 36] have been done for improving the performance by tuning configuration parameters. We adopt a static approach to evaluate the application on a small amount of data to decide on appropriate configuration parameter settings. A subset of parameters are summarized in Table 3-1 and Table 3-2. The target of the tuning is the R-Heavy implementation, since it is the implementation that works for every data mapping strategy. Also, it takes advantage of the data shuffling phase performed by the Hadoop platform, which is done by collecting values generated in the map phase into groups of values, each group sharing the same key. We introduce these configuration parameters in this chapter and evaluate their effect on system performance in Chapter 4.

3.1.1 Parameters of the MapReduce Programming Model

As discussed in Chapter 1, applications following the MapReduce programming model are naturally parallelized and distributed among clusters of computers. The number of map tasks and the number of reduce tasks determine the degree of parallelization.
<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapred.map.tasks</td>
<td>Number of map tasks in a job execution</td>
</tr>
<tr>
<td>dfs.block.size</td>
<td>HDFS block size</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>Number of reduce tasks in a job execution</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>The size (in MB) of the buffer that is used for writing a map task output</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>The threshold of the map output buffer. If the size of output in the buffer exceeds the threshold, the background thread starts to spill content into disks.</td>
</tr>
<tr>
<td>io.sort.factor</td>
<td>The maximum number of streams used for merging map-phase output or reduce-phase input</td>
</tr>
<tr>
<td>mapred.compress.map.output</td>
<td>This flag indicates whether or not to compress the output of map tasks</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>The number of threads used in reduce tasks to copy intermediate results from map tasks to the reduce task.</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>The proportion of the memory of a reduce task that is used to hold output from map tasks.</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>The threshold (the proportion of the buffer used to hold output from map tasks) of the buffer that is used to hold input to a reduce task.</td>
</tr>
<tr>
<td>mapred.inmem.merge.threshold</td>
<td>The threshold (the number of records from map tasks) of the buffer that is used to hold input to reduce task.</td>
</tr>
<tr>
<td>mapred.map.tasks.speculative.execution</td>
<td>The flag enables the map phase speculative execution.</td>
</tr>
<tr>
<td>mapred.reduce.tasks.speculative.execution</td>
<td>The flag enables the reduce phase speculative execution.</td>
</tr>
</tbody>
</table>
Table 3-2. A subset of configuration parameters tuned for system optimization - part 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapred.job.reuse.jvm.num.tasks</td>
<td>The number of tasks to reuse the same JVM. Note that they don’t share the JVM. Tasks that use the same JVM are executed sequentially.</td>
</tr>
<tr>
<td>io.file.buffer.size</td>
<td>The buffer size used by Hadoop IO</td>
</tr>
<tr>
<td>mapred.reduce.slowstart.completed.</td>
<td>The proportion of finished map tasks when the reduce task starts.</td>
</tr>
<tr>
<td>maps</td>
<td></td>
</tr>
<tr>
<td>mapred.tasktracker.map.tasks.</td>
<td>The maximum number of map tasks executed on the same TaskTracker.</td>
</tr>
<tr>
<td>maximum</td>
<td></td>
</tr>
<tr>
<td>mapred.tasktracker.reduce.tasks.</td>
<td>The maximum number of reduce tasks executed on the same TaskTracker.</td>
</tr>
<tr>
<td>maximum</td>
<td></td>
</tr>
<tr>
<td>mapred.child.java.opts</td>
<td>Java heap size for map/reduce tasks</td>
</tr>
</tbody>
</table>

The number of map tasks is determined by the total number of splits of input data to a MapReduce job. (It is assumed that the file split flag is true so that a large file can be split across multiple storage nodes. If this flag is false, a large file can still be split. The map task, however, will read a whole file, no matter how many blocks a file is split into. In this case, the number of map tasks is determined by the number of input files.) Using the one-to-one data mapping strategy, each split refers to a file. Using the one-to-many data mapping strategy, each split refers to a block of a file. The block size of HDFS determines the number of blocks into which a file is split in the storage system. If the file size is less than the block size, the file cannot be split. Otherwise, the number of splits is computed as \[ \left\lceil \frac{F_s}{B_s} \right\rceil \], where \( F_s \) refers to as the size of a file, \( B_s \) refers to as the block size of HDFS and \( \lceil \rceil \) denotes the “ceiling of”. To decrease the number of map tasks, the block size of HDFS should be increased and the number of files should be decreased when given the total amount of data unchanged.
The number of reduce tasks refers to how many reduce tasks are launched during a job execution. The parameter is controlled by users. Usually, the number of reduce tasks is set according to available resources [1].

For the R-Heavy implementation, the map phase is responsible for filtering useless information and categorizing data by iCUs while the reduce phase conducts the analysis described in Algorithm 1. The number of files and the block size of HDFS determine the number of map tasks and the number of reduce tasks is controlled by parameter `mapred.reduce.tasks`. Chapter 4 shows evaluation results of the effect of these two factors.

### 3.1.2 Parameters of Hadoop Workflow

Extending the overview of Hadoop in Chapter 1, this section presents an in-depth view of the Hadoop workflow. We then analyze different configuration parameters involved in the workflow that may influence the performance of applications.

As shown in Figure 1-3, when the master node assigns a map/reduce task to a slave node, the TaskTracker on the slave node runs the map/reduce task. Figure 3-1 shows the map/reduce task execution workflow, including the shuffling phase, in detail.

In a Hadoop MapReduce job, the input data to the reduce task are sorted by the keys. The shuffling phase in Figure 1-3 shows the process of sorting and transforming intermediate data from the map phase to the reduce phase. Figure 3-1 details the shuffling phase execution. Each map task is allocated a memory block used for buffering the output of the task. During the map task execution, outputs of the map task are written to the buffer first. When the size of the data in the buffer exceeds a specific threshold, a background thread starts to spill the data to the local disk. Every time the size of the data in the buffer exceeds the threshold, there is a file generated. At the end of the map task, there may be multiple files, which are merged into three files usually. Before the reduce task starts, several threads retrieve the outputs from map tasks. Since the reduce task may retrieve outputs from different map tasks, there may be multiple pieces of data before the execution of the reduce task. A background thread merges multiple pieces...
Figure 3-1. A shuffling phase of a MapReduce job. Each map task contains a map function. The buffer associated with the map function is the place that stores the output from the map function. When the size of the data in the buffer exceeds some threshold, the data are split into disks. The blocks following the buffer in each map task represent the blocks written on the disk. Each reduce task contains a reduce function. Before the reduce function starts, the reduce task retrieves the output from map tasks and merges the outputs as the input to the reduce function, represented by the arrow and the merge streams in each reduce task.

of data into one piece. The reduce task then starts with the one-piece input from the shuffling phase. After the reduce function execution, outputs are written into HDFS.

Intermediate outputs generated during the map phase are stored in a buffer whose size is specified by the $io.sort.mb$ parameter. If the amount of intermediate data exceeds a threshold, a background thread spills the intermediate data to the disk so that additional outputs from map tasks can be written to the buffer. The threshold is determined by both the size of the buffer and a percent of that size $io.sort.spill.percent$ above which data are spilled. Before the intermediate outputs are written to the disk, the background thread partitions the data according to corresponding reduce tasks. The data to be sent to the same reduce task are in a same partition and sorted according to intermediate keys. If
io.sort.spill.percent percent of the buffer is full, the background thread starts to spill the content to the disk with regards to the partition in the buffer.

Due to the amount of intermediate data that map tasks generate, there may be multiple spilled files. Before the map phase ends, multiple spilled files are merged into a single file. The configuration parameter io.sort.factor determines the number of streams used to merge at once. It is also possible to compress the intermediate data before the data are written into the disk. If the data are compressed at the map phase, they have to be decompressed at the reduce phase. mapred.compress.map.output is the flag deciding whether or not to compress the output of the map phase. mapred.map.output.compression.codec tells the Hadoop platform the specific compression library [1].

For the R-Heavy implementation, the map phase extracts useful information and composes < K_r, V > pairs. Before the map phase ends, the intermediate < K_r, V > pairs are partitioned and sorted with regards to different K_r.s. The configuration parameters involved in this phase may have significant effect on the execution time of the application.

During the reduce phase, the reduce task retrieves the partition of the data that are supposed to be sent to it. Since multiple map tasks may finish at different time, the reduce task starts to retrieve the partition as long as the map task that generates the partition is finished. The locations of partitions are communicated to the JobTracker through the heartbeat process by each TaskTracker. The number of threads that are used to copy outputs of map tasks to a reduce task is determined by the parameter mapred.reduce.parallel.copies. The intermediate data generated by map tasks are copied into a buffer at a reduce task, whose size is determined by mapred.job.shuffle.input.buffer.percent, the percent of the heap of the reduce task to be allocated for the buffer. Similar to the map phase, when the amount of data in the buffer exceeds a threshold, the data in memory are spilled into the disk. Two parameters control the threshold, mapred.job.shuffle.merge.percent and mapred.inmem.merge.threshold.
After all required inputs for a reduce task are retrieved from map tasks, a background thread is used for merging the inputs into one sorted file. The `io.sort.factor` introduced previously also determines the number of streams used for merging. Then, the reduce phase computation starts and the outputs are written into HDFS.

The reduce phase of the R-Heavy implementation retrieves the partition — a set of `<key, value>` pairs — belonging to the same ICU prior to the execution of the reduce function. The data belonging to the same ICU may be generated by different map tasks. The reduce tasks retrieve the data from multiple locations and merge the data into one sorted data set. After the data have been merged, the reduce function conducts computations described in Algorithm 1 on each ICU. Each reduce task may process one or multiple ICUs.

Configuration parameters involved in a MapReduce workflow significantly influence the way to utilize available resources. A better parameter setting may bring a performance boost.

### 3.1.3 Other Hadoop Features

We have introduced parameters involved in the MapReduce programming model and a Hadoop job workflow. Other parameters related to task execution, I/O, and environment settings may influence performance of applications as well.

The MapReduce programming model allows a job to be distributed among multiple nodes to achieve space-parallel execution. During a job execution, one task may slow down the total execution time of an application when other tasks have already finished and the application is still waiting for the results from the slow task. If the execution time of a task is significantly longer than other parallel tasks, Hadoop tries to schedule the same task to be executed on a different node and accepts the results returned by the first node that completes the task. This procedure is referred to as speculative execution. The property significantly improves the performance of a Hadoop job so that a single slow task does not affect the total job execution time. The speculative execution is
not about launching two tasks for processing the same data block at the same time. Instead, it computes the task execution rate and determines which task needs to be scheduled on a different node. The parameters `mapred.map.task.speculative.execution` and `mapred.reduce.task.speculative.execution` are two flags enabling the speculative execution in the map and reduce phase respectively.

When the JobTracker assigns a task to a TaskTracker, the TaskTracker assigns the task to be executed in an individual Java Virtual Machine. If the TaskTracker creates one JVM for each task assigned to it, the total JVM creation time is significant for large amount of short lived tasks. The parameter `mapred.job.reuse.jvm.num.tasks` can enable the TaskTracker to reuse JVM for other tasks. However, multiple tasks do not share the same JVM at the same time. Instead, they are executed sequentially by the same JVM.

Another property of the Hadoop platform is lazy initialization of reduce tasks. The parameter `mapred.reduce.slowstart.completed.maps` decides the percent of the map phase finished when the reduce tasks are initialized. If reduce tasks are initialized in an early stage of a long-running job, the performance of the map phase may be degraded. The reason is that reduce tasks will share resources with map tasks, but reduce tasks cannot start without all map tasks having finished. Thus, lazy initialization could delay the initialization of the reduce phase execution so that the map phase can possibly utilize all available resources.

After a Hadoop cluster has been set up, multiple environment parameters need to be determined. Appropriately setting the parameters benefits both the application executed on the cluster and the resource utilization of the cluster. As shown in Figure 1-3, the TaskTracker works in each slave node. The number of tasks that a TaskTracker allows to run is configured by the parameters `mapred.tasktracker.map.tasks.maximum` and `mapred.tasktracker.reduce.tasks.maximum`. According to available resources, the principle of setting the number of tasks for each TaskTracker is to achieve optimal utilization. Usually, the TaskTracker and the DataNode both run as individual processes. To utilize
the available resources, the parameter can be estimated through $N_{\text{processor}} \times w - 2$, where $N_{\text{processor}}$ is the number of processors available on the node, $w$ is the number of processes running on the same processor, and 2 refers to the TaskTracker process and the DataNode process. We discuss our cluster settings in Chapter 4.

Table 3-1 and Table 3-2 summarize all configuration parameters introduced in this section. Studies show that tuning parameters bring significant performance improvement to applications. Experimental evaluation of this section are shown in Chapter 4.

### 3.2 Algorithmic Optimization

We have introduced the characterizations of the data to be analyzed in Section 2.1. One piece of information that each record carries is the time stamp indicating when the record is collected. Based on the time stamp information, the system retrieves the data from the database. The new data are generated every day and part of the old data are deleted. A typical statistical computation uses data collected during a “window” of several days (e.g. the last 14 days). Figure 3-2 shows that the statistical analysis is performed on the data from 14 days, Day-1 to Day-14. At the end of Day-15, the data from Day-1 are deleted and the Day-15 data are added. Then the computation is performed on the data from Day-2 to Day-15.

![Figure 3-2. The sliding window of statistical analysis. The window size is 14 days.](image-url)
In this Section, we take the time stamp information into consideration and propose a method that takes advantage of the previous data and computation results to algorithmically optimize the performance of the system.

### 3.2.1 Data Movement

There are three data mapping strategies that have been discussed in Section 2.3: arbitrary, per-iCU and per-dCU mapping. We propose a new data mapping strategy considering the time stamp information of each record, referred to as per-window mapping.

Per-window mapping maps the data belonging to the same time period to the same file. For example, the data collected from Day-1 are mapped to the same file, as shown in Figure 3-3. Since massive amounts of data can be collected each day, per-window mapping can be combined with other mapping strategies to decrease the file size.

![Figure 3-3. Data mapping strategy with regards to time stamp of each record](image-url)
Using this approach, the analysis system only retrieves newly generated data each day instead of retrieving all data that belong to the same computation window. Prior to the analysis, the system removes outdated data and adds new data to HDFS by a relatively short preprocessing procedure. The amount of data to be moved decreases by approximately 1/14 for the case that the window size is 14 days.

3.2.2 Algorithmic Optimization

If we keep some intermediate analysis results from previous computations, we can take advantage of precomputed results to decrease the computation time for the data window of interest. In this section, the Stage 1 of Algorithm 1, discussed in Section 2.2, is optimized by introducing a new algorithm.

3.2.2.1 The sliding-window algorithm

The Stage 1 of Algorithm 1 consists of multiple steps. Among those steps, sorting and computing the average and the standard deviation of each dCU are the most time-consuming parts. In each analysis run, only a relatively small amount of data are added into the whole data set. Taking Figure 3-2 as an example, at Day-15, the new data generated in Day-15 are added into the Day-15 analysis run, and the outdated data from Day-1 are deleted. If it is assumed that the amount of data generated in each day is the same, the newly added data set is 1/14 of the total data set to be analyzed. Thus, if the computation performs only on the newly generated data, the total computation time could be decreased.

We propose a sliding window algorithm to optimize the computation. It is assumed that the previous records are sorted and the average and the standard deviation of each dCU computation results are known. The sorting can be done by first sorting the small newly generated data set, and then using the mergesort algorithm [37] to merge the small data set to the previous sorted data set. Simultaneously, the algorithm checks time stamp information of each record and deletes outdated data when the record lies outside the computation window. SlidingWindowSort is the pseudo-code for the sorting algorithm.
procedure SlidingWindowSort(Array A (sorted array), Array U, Window w)
Sort U using any sorting algorithm
Allocate output array B
while i < A.length && j < U.length do
    if (A[i] <= U[j]) then
        if (A[i] time stamp is in w) then
            Copy A[i] to B
        end if
        i ++
    else
        Copy U[j] to B
        j ++
    end if
end while
if i < A.length then
    Copy the left elements in A to B if the element is in w
else
    Copy the left elements in U to B
end if
end procedure SlidingWindowSort

If the previous computation results are known, we can use the previous average and summation to compute the current average and standard deviation for each dCU.

Usually, the standard deviation for a data set is computed using a two-pass algorithm, as computed using Equations 3–1 and Equation 3–2. The first pass computes the average, and the second pass computes the standard deviation. One-pass algorithm is known in statistics textbooks [38] using Equation (3–3). Online-algorithm for computing the variance, the square of the standard deviation, can be dated back to 1962. Welford proposed an online-algorithm for computing the variance without the knowledge of previous data [39]. When a new number is added, the online-algorithm uses previous variance and mean to compute the new variance. Our approach is a combination of online-algorithm and one-pass algorithm. Instead of using the previous computed variance, we use the summation of squares of the previous data to compute the variance and the
standard deviation of the updated data. Again, Figure 3-2 is an example to show the computation procedure. We derive the computing results mathematically as follows.

\[ \mu = \frac{1}{N} \sum_{i=1}^{N} x_i \]  

(3-1)

\[ \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2} \]  

(3-2)

\[ \sigma = \sqrt{\frac{1}{N} \left( \sum_{i=1}^{N} x_i^2 - \frac{1}{N} \left( \sum_{i=1}^{N} x_i \right)^2 \right)} \]  

(3-3)

Let \( \mu_1 \) denote the average of Day-1, \( \mu_{15} \) the average of Day-15, \( \mu \) the average from Day-1 to Day-14 and \( \mu' \) the average from Day-2 to Day-15. Let \( N_1, N_{15} \) and \( N \) be the numbers of records in Day-1, Day-15 and Day-1 to Day-14, respectively. Then, \( \mu' \) is computed through

\[ \mu' = \frac{\mu \cdot N - \mu_1 \cdot N_1 + \mu_{15} \cdot N_{15}}{N - N_1 + N_{15}} \]  

(3-4)

It is not obvious how to use previous results to compute the standard deviation of each dCU. We can start from computing the variance of the data set from Day-2 to Day-15. \( v' \) is the variance from Day-2 to Day-15. Then,

\[ v' = \frac{\sum_{i \in \text{Day-15}} (x_i - \mu')^2}{N - N_1 + N_{15}} \]  

, where \( x_i \) is an element in the data from Day-2 to Day-15. The equation can be expanded as

\[ v' = \frac{\sum_{i \in \text{Day-15}} (x_i^2 - 2 \cdot x_i \cdot \mu' + \mu'^2)}{N - N_1 + N_{15}} \]  

. We define \( S^2 \) as the summation of squares of each element in a certain data set and \( S \) as the summation of all elements in the data set, then

\[ S^2 = \sum x_i^2, S = \sum x_i \]  

.
The variance can be computed through

\[ v' = \frac{S^2 - S_1^2 + S_{15}^2 - \mu' (S - S_1 + S_{15}) + (\mu' * N')^2}{N'} \],

where \( N' = N - N_1 + N_{15} \). The standard deviation, then, is derived through

\[ \sigma' = \sqrt{v'} = \sqrt{\frac{S^2 - S_1^2 + S_{15}^2 - \mu' (S - S_1 + S_{15}) + (\mu' * N')^2}{N'}} \]. (3–5)

So far, we have used the average, the summation and the summation of squares of the previous data to compute the average and the standard deviation of the updated data.

Algorithm 2, shown below, uses the sliding window approach to compute the average and the standard deviation with previous results. We assume that each element in the double-precision array contains a time stamp. When iterating the array, we can extract the outdated data by checking the time stamp of the element. \( A_i \) represents for the \( i \)th 1D double-precision array in the linked list \( inputList \), \( A_i[k] \) represents an element in the array \( A_i \). It is assumed that \( U_i \) is the \( i \)th 1D double-precision array in the linked list \( newData \) and it is the newly added data to the array \( A_i \). \( S_X \) is the summation of elements in array \( X \); \( S_X^2 \) is the summation of squares of each element in \( X \); \( \mu_X \) is the average of elements in \( X \); \( N_X \) is the number of elements in \( X \). Each array \( A_i \) is sorted and \( S_{A_i}, S_{A_i}^2, \mu_{A_i}, \) and \( N_{A_i} \) are known. The first stage of Algorithm 2 starts with sorting and computing the average, the summation and the summation of squares of the newly added data.

Then, the algorithm merges the newly added data to the previous data and removes the out-dated data by checking the time-stamp of each record. After merging, outlier limits are computed to identify outliers. Finally, the proposed algorithm, Equation 3–4 and Equation 3–5, is used to compute the average and the standard deviation of the updated data. The second stage remains unchanged.

3.2.2.2 Algorithm analysis

The first stage of Algorithm 2 is optimized using SlidingWindowSort algorithm and Equation 3–5 and Equation 3–4. After sorting, the percentile and outlier limits
Algorithm 2 Sliding window approach to equivalence test

**Input:** LinkedList<Double[]>> `inputList`, LinkedList<Double[]>> `newData`

for each DoubleArray `A_i` in `inputList`, each DoubleArray `U_i` in `newData` do

Sort `U_i`

Compute `S_{U_i}`, `S^2_{U_i}`, `\mu_{U_i}`, `N_{U_i}`

Merge `U_i` to `A_i`, extract outdated elements and store in `X_i`

Compute `S_{X_i}`, `S^2_{X_i}`, `\mu_{X_i}`, `N_{X_i}`

Compute the percentile values using `PERCENTILE` function of `X_i`

Compute the lower and upper outlier limits `LOL`, `UOL` using percentile values

Eliminate `X_i[k]` that is not in the range `[LOL, UOL]`

\[ N_i = N_{A_i} - N_{X_i} + N_{U_i} \]

Compute \[ \mu_i = \frac{\mu_{A_i} * N_{A_i} - \mu_{X_i} * N_{X_i} + \mu_{U_i} * N_{U_i}}{N_i} \]

\[ \sigma_i = \sqrt{\frac{S^2_{A_i} - S^2_{X_i} + S^2_{U_i} - \mu_i \ast (S_{A_i} - S_{X_i} + S_{U_i}) + (\mu_i \ast N_i)^2}{N_i}} \]

Compute the process capability index `C_{pk}[i]`

end for

Pick `X_i` with the largest `C_{pk}` as reference set `R`

Compute Euclidean distance \[ W_R[k] = \sqrt{(R[k] - \mu_R)^2} \]

Compute Euclidean distance \[ B_R[k] = \sqrt{(R[k] - \mu)^2} \]

Compute \[ \mu_{W_R}, \sigma_{W_R} \] and \[ \mu_{B_R}, \sigma_{B_R} \]

for each DoubleArray `X_i` in `inputList` do

Compute Euclidean distance \[ W_i[k] = \sqrt{(X_i[k] - \mu)^2} \]

Compute Euclidean distance \[ B_i[k] = \sqrt{(X_i[k] - \mu_R)^2} \]

Compute \[ \mu_{W_i}, \sigma_{W_i} \] and \[ \mu_{B_i}, \sigma_{B_i} \]

Do a pairwise Student’s t-test between \( W_R \) (or \( B_R \)) and \( W_i \) (or \( B_i \))

Final decision based on the result of t-test

end for

computations are the only ones to be done, which are not time-consuming. The standard deviation is used to be computed using a two-pass algorithm [38], which requires accessing each record twice. In our approach, we only need to access each record once.

An in-memory sorting algorithm has a time complexity \( O(n \log n) \), where \( n \) is the number of records to be sorted. Using SlidingWindowSort algorithm, we take advantage of the merge sort algorithm. The new approach results a time complexity \( O(n + k \log k) \), where \( n \) is the total number of records to be sorted and \( k \) is the number of newly generated records.
The two-pass algorithm for computing the standard deviation accesses each record twice, computing the average and computing the standard deviation. The sliding window algorithm, as well as online-algorithm, accesses each record once. In the situation when the memory space is limited, this algorithm can decrease memory access time by 50%. If the previous results are known, computing the standard deviation only requires computing the average and the summation of squares of the newly generated data set. The execution time of sliding window algorithm for computing the standard deviation is 1/14 of the original algorithm (assuming that the amount of data generated each day is the same).
CHAPTER 4
EXPERIMENTAL EVALUATION AND ANALYSIS

This chapter summarizes experimental results for the data analysis approaches discussed in previous chapters. Experiments were done to evaluate i) different data mapping strategies corresponding to different implementations; ii) different implementations’ scalability with regards to the large scale data set; iii) the effect on performance of different Hadoop configuration parameters; iv) the performance of the sliding window algorithm.

4.1 Data Mapping and MapReduce Implementation Experiments

The data collected from monitoring sensors can increase in multiple dimensions. The data size (in terms of records), \( S \), depends on the number of iCUs, \( I \), the number of dCUs in each iCU, \( D \), and the size of each dCU, \( R \).

\[
S = \mathcal{F}(I, D, R)
\]  

(4–1)

In following experiments, it is assumed that the size of each dCU and the number of dCU of each iCU are the same. Therefore, Equation 4–1 becomes

\[
S = I \times D \times R.
\]

(4–2)

The experiments are set up on a local cluster consisting of 9 physical nodes, one master node and eight slave nodes. One node dedicates to be the master node. Each physical node contains 8 Intel Xeon 2.33GHz processors and 16GB memory. Cloudera Distribution of Hadoop (CDH) 4.2.1 is installed on the local cluster.

4.1.1 Data Mapping

Different data mapping strategies are described in Chapter 2. Experiments are set up to compare different strategies corresponding to different implementations.
4.1.1.1 Experiments set up

We assume the number of dCUs in each iCU, $D$, is a constant, 6 (there are approximate 6 EUs for each recipe and item). We conduct experiments following Table 2-3 by increasing the size of each dCU, $R$, starting from 1 MB to 24 MB. For the many-to-one scenario, the factor is chosen to be 2, meaning that each map task handles two files. The size of the data, $S$, is 30GB. As $R$ increases, the number of iCUs, $I$, decreases.

Since the R-Heavy implementation can be considered as the M-Heavy implementation and an extra function that groups the data belonging to the same iCU together, it is the only implementation that can be used in the arbitrarily mapping strategy. Intuitively, the execution time of programs using arbitrarily mapping strategy is longer than using other mapping strategies. Thus, we only consider the per-iCU and per-dCU mapping strategies.

4.1.1.2 Results analysis

For comparison purposes, we show experimental results in two different figures. Figure 4-1 compares the execution time of different data mapping strategies for each implementation. For the R-Heavy implementation, when the size of dCU is less than 16 MB, the per-iCU many-to-one mapping strategy outperforms other mapping strategies. The MR-Balanced implementation shows similar behavior. For the M-Heavy implementation, the execution time changes slightly as the size of dCU changes. All three implementations show an very slightly increasing trend after the size of dCU exceeds 12 MB. Since a per-dCU one-to-one mapping will generate many more files than a per-iCU many-to-one mapping, the Hadoop platform generates more map tasks in the per-dCU one-to-one scenario than it does in the latter. Thus, the execution time for the per-dCU one-to-one mapping strategy is longer than the time taken for the per-iCU case. However, as the size of each dCU increases, the task execution time, including computation and data movement time, dominates the total execution time, which results in similar execution time for all mapping strategies.
Figure 4-1. Different data mapping strategies for each implementations

Figure 4-2 compares the execution time of different implementations for each data mapping strategy. Different implementations show similar behaviors for the same data mapping strategy. The per-iCU many-to-one strategy shows a very slightly increasing trend when the size of dCU exceeds 8 MB. As the size of dCU grows, each map task handles larger and larger files. Since the computation is memory-intensive, the data size of each task will be bounded by memory. The data size for each task in the per-iCU mapping strategy is larger than other mapping strategies. Thus, it is the first case that shows up the increasing trend of time-consumption as the size of dCU increases. If the size of dCU is too large such that a single iCU cannot be computed in a single node, a parallel algorithm must be designed to parallelize the execution. We have proposed an iterative MapReduce parallel algorithm in Appendix B.

4.1.2 Large Scale Data Sets

In order to capture the scale of monitoring data in the real-world manufacturing system, the experiments assumed that 1 TB data are to be analyzed each day.

4.1.2.1 Experiments set up

Monitoring data for 25 iCUs were generated by modifying actual values from records collected from a semiconductor manufacturing system [6]. It is assumed that $R$ and $D$ are constant. From this initial data set, the data size is growing by increasing the number of iCUs based on Equation 4-2. The $I$ is in the range from 625 to 2,000,000. The data size is increased from 0.32 GB to 1 TB correspondingly.
Figure 4-2. Different implementations for each data mapping strategy.

Since the size of each dCU is less than 1 MB, we use per-iCU many-to-one data mapping strategy with a many-to-one factor of 125. The file size ranges from 54 MB to 73 MB. The DFS block size is 128 MB. The number of reduce tasks is set to 8.

4.1.2.2 Results analysis

Figure 4-3 shows several experimental results. As the data size increases, the computation time increases linearly as well. The M-Heavy implementation outperforms the R-Heavy implementation and the MR-Balanced implementation.

The map phase execution time of the MR-Balanced and the M-Heavy implementation are approximately the same. The map phase execution time of the R-Heavy implementation is shorter than other implementations, because the map phase of R-Heavy implementation only filters and categorizes records without any computation.

The reduce phase execution time of the M-Heavy implementation is shorter than that of other implementations, since the reduce phase of the M-Heavy implementation is
absent. The reduce phase execution time of the R-Heavy implementation is the longest because all computations are done in this phase.

4.2 Hadoop Configuration-Parameters Tuning Experiments

We evaluate the effect of different configuration parameters, shown in Table 3-1 and 3-2, on the performance of the R-Heavy implementation. As discussed in Chapter 3, we tuned the R-Heavy implementation for reasons that the R-Heavy implementation works for any mapping strategy and it takes advantage of the shuffling phase of a typical Hadoop job. Table 4-1 shows default values of configuration parameters of an R-Heavy job. The default value set is a typical setting for a Hadoop cluster. We have used this setting for previous experiments. In our experiments, if a specific parameter is tuned, other parameters use the values in Table 4-1.

4.2.1 Parameters of the MapReduce Programming Model

The MapReduce programming model has two main phases, the map phase and the reduce phase. Thus, the number of map tasks and the number of reduce tasks are two important factors influencing the performance of a MapReduce program. In this section, these two factors are tuned and discussed.

A 100 GB data set is generated for testing purposes. Each file contains data from different iCUs and dCUs. The number of map tasks, representing by `mapred.map.tasks`, is determined by both the parameter `dfs.block.size` and the file size of each file. It is assumed
Table 4-1. The default values of configuration parameters in Table 3-1 and 3-2

<table>
<thead>
<tr>
<th>Name</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapred.map.tasks</td>
<td>800</td>
</tr>
<tr>
<td>dfs.block.size</td>
<td>128 MB</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>16</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>200 MB</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>0.8</td>
</tr>
<tr>
<td>io.sort.factor</td>
<td>64</td>
</tr>
<tr>
<td>mapred.compress.map.output</td>
<td>true</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>10</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>0.7</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.66</td>
</tr>
<tr>
<td>mapred.inmem.merge.threshold</td>
<td>1000</td>
</tr>
<tr>
<td>mapred.map.tasks.speculative.execution</td>
<td>false</td>
</tr>
<tr>
<td>mapred.reduce.tasks.speculative.execution</td>
<td>false</td>
</tr>
<tr>
<td>mapred.job.reuse.jvm.num.tasks</td>
<td>1</td>
</tr>
<tr>
<td>io.file.buffer.size</td>
<td>64 KB</td>
</tr>
<tr>
<td>mapred.reduce.slowstart.completed.maps</td>
<td>0.8</td>
</tr>
<tr>
<td>mapred.tasktracker.map.tasks.maximum</td>
<td>8</td>
</tr>
<tr>
<td>mapred.tasktracker.reduce.tasks.maximum</td>
<td>4</td>
</tr>
<tr>
<td>mapred.child.java.opts</td>
<td>1536 MB</td>
</tr>
</tbody>
</table>

that the file is splittable and the splitting size is the same as the block size of HDFS. The relationship can be represented by

\[
M = \sum_{i=0}^{N} \left( \text{ceil} \left( \frac{S_i}{B} \right) \right)
\]  

(4-3)

where \( M \) denotes the number of map tasks, \( N \) the total number of files, \( S_i \) the file size of the \( i \)th file and \( B \) the block size of HDFS. The R-Heavy implementation, like many other Hadoop applications, will assign each map task to an independent block of data. For example, if a file size is 10 GB and the block size is 1.5 GB, the number of blocks is 7, thus, the number of map tasks launched for the file is 7. In this experiment, \( N \) is 10, and each file is 10 GB. By changing the block size, the number of map tasks launched is changed as well.

Figure 4-4 shows the experimental results of the influence of the number of map tasks on the execution time of the program. As the block size increases, the number of map
Figure 4-4. Influence of the number of map tasks launched. The dots in the figure show the experimental results. Each dot contains three dimensions, the number of map tasks, the HDFS block size and the execution time. The execution time is shown using different shades of gray. The darker the area is, the less time the job takes to run.

As represented by Equation 4–3, the number of map tasks for each file is the ceiling of the file size divided by the block size of HDFS. Thus, in some cases, the data are not evenly distributed among map tasks. The dots in the figure show the experimental results. Each dot contains three dimensions, the number of map tasks,
the HDFS block size and the execution time. The vertical axis is the number of map tasks launched, and the horizontal axis is the block size of HDFS. The shades of gray represent the amount of time it takes for the job to run. The darker the area is, the shorter the amount of time the job takes. Though the number of map tasks decreases as the size of HDFS block increases, the execution time does not show any trend. Because of unevenly distributed data, some of map tasks take longer time than other map tasks in a job. If the case when data are evenly distributed is considered, however, the execution time decreases as the number of map tasks decreases, as shown in Figure 4-5. In this case, we consider the block size 64 MB, 128 MB, 256 MB, 512 MB, 1024 MB, 2048 MB and 3072 MB, which are used in order to evenly distribute the data from the same file among multiple blocks. Correspondingly, the number of map tasks decreases from 1600 to 40. The minimal execution time is achieved when the number of map tasks is 50 with the corresponding block size 1024 MB. As the number of map tasks continues decreasing, the execution time increases, in contrary, which is shown at the beginning of the horizontal axis in Figure 4-5. This increasing trend indicates the problem of too large granularity of the input data.

As discussed previously, the JobTracker tries to assign the task where the input data are located. If, on the other hand, the node that stores the input is busy, the input has to be transferred to another node. The penalty of data transmission is low when the size of transmitted data is small. As the block size increases, the time it takes to transmit the data for this situation is much longer. Thus, the time for input data transmission lags the total execution time of map tasks.

The parameter mapred.reduce.tasks decides how many reduce tasks are launched for a job. In our experiments, this parameter is varied from 1 to 128. Figure 4-6 shows the execution time for different numbers of reduce tasks. The execution time of the map phase

---

1 The time it takes to transfer the input file from the source node to the target node is much longer than the time it takes to perform computations locally.
Figure 4-5. Influence of the number of map tasks (evenly data distribution)

does not change much during experiments. When the number of reduce tasks is relatively small, the execution time of the reduce phase decreases as the number of reduce tasks increases, which contributes to decreasing of the total execution time of a job. As the number of reduce tasks increases, the utilization of the resource of the cluster increases. When the number of reduce tasks exceeds 20, the execution time of the reduce phase remains unchanged. When the number of reduce tasks is relatively large, the execution time of the reduce phase shows a very slightly increasing trend, because increasing the number of reduce tasks results in increasing the system overhead. When the system resource is fully utilized, increasing the number of reduce tasks will bring slightly more overhead.

4.2.2 Parameters of Hadoop Workflow

The shuffling phase plays an important role in a MapReduce job. Several configuration parameters involved are proven to significantly influence the performance of a MapReduce job.

\(\textit{io.sort.mb} \) and \(\textit{io.sort.spill.percent} \) decide the buffer size of the output of a map task. The principle for setting these two parameters is that as long as the memory is available, the buffer size should be set as large as possible. Figure 4-7 shows the experimental
results of the influence of these two parameters on the performance of the R-Heavy implementation. The result goes against the expectation so that as both parameters increase, the execution time of the program increases as well. When the io.sort.mb is set to 50 MB and the io.sort.spill.percent is set in the range between 0.3 and 0.9, the execution time is decreased by 13 % of the worst case scenario in this experiment.

Figure 4-7. Influence of the buffer size of the output from the map function

io.sort.factor represents the number of streams that are used for merging the output of a map task or the input of a reduce task. Theoretically, the larger the parameter is, the less time it takes to merge the output of map tasks and the input of reduce tasks. Figure 4-8 shows the experimental results. Local minimal points are located at 16, 64, 128... The total resources available in the cluster are 64 cores. Thus, the optimal performance of the program is achieved at the point where the merging phase utilizes all available resources.
and there are no extra streams allocated for this phase. In this experiment, the value is in the range $[60, 75]$.

![Graph showing influence of io.sort.factor on execution time]

Figure 4-8. Influence of the parameter $io.sort.factor$ on the execution time

After the map phase, the outputs of map tasks are copied to the corresponding reduce tasks. The number of threads that are used for copying intermediate data is determined by the parameter $mapred.reduce.parallel.copies$. Figure 4-9 shows that the map phase execution time is not influenced by this parameter. The reduce phase execution time is minimal when the number of threads is 10. Figure 4-9 suggests that choosing a number of threads in the range $[5, 20]$ will yield better execution time. As the parameter keeps increasing, the execution time increases as well. When the resource is fully-utilized, additional threads bring only overhead rather than performance improvement.

The results of the map phase are first copied to a buffer, a block of the heap deployed in each reduce task process of a reduce task. When the buffer is full, outputs from map tasks are spilled into disk. Three parameters, $mapred.job.shuffle.input.buffer.percent$, $mapred.job.shuffle.merge.percent$ and $mapred.inmem.merge.threshold$, influence the size of the buffer. For the R-Heavy implementation, the computation takes place in the reduce phase which requires sufficient memory space. Thus, the size of the buffer used to temporarily store the output of each map task is limited so that the rest of the memory is sufficient for computations. Figure 4-10 and 4-11 show two groups of
Figure 4-9. Influence of the parameter `mapred.reduce.parallel.copies` on the execution time experimental results. One shows the effect of `mapred.job.shuffle.input.buffer.percent` and `mapred.job.shuffle.merge.percent` on the execution time, and the other shows the effect of `mapred.job.shuffle.input.buffer.percent` and `mapred.inmem.merge.threshold` on the execution time. If any of three parameters is set too large, the computation cannot finish due to the lack of memory space. The stars in the figure indicate that the experiments cannot be finished due to the lack of memory space for the reduce task computation.

Figure 4-10. Influence of the reduce phase buffer size (percent). The stars in the figure indicate that the experiments cannot be finished due to the lack of memory space for the reduce task computation.

4.2.3 Other Hadoop Features

`mapred.job.reuse.jvm.num.tasks` is the parameter that decides how many tasks share a Java virtual machine. Since each task uses an independent JVM, tasks within the same node share a JVM sequentially. When the parameter is less than 10, the execution time
Figure 4-11. Influence of the reduce phase buffer size (records)
of the map phase decreases as the number of tasks sharing a JVM increases whereas the
execution time of the reduce phase increases. When the parameter value is around 5, the
total execution time is minimal.

Figure 4-12. Influence of the parameter `mapred.job.reuse.jvm.num.tasks` on execution time

`mapred.reduce.slowstart.completed.maps` is the parameter whose value is the
proportion of the map phase that has finished when the reduce phase starts. Typically,
this is set to a large value such that all available resources are dedicated to the map phase
for a large job. Figure 4-13 shows the experimental results of measuring the influence of
this parameter on the execution time. As the parameter increases, the execution time
increases as well. After the parameter exceeds 0.3, the execution time starts decreasing.
When the parameter is in the range [0.7, 0.8], the execution time is minimal. Then it
increases again.
Figure 4-13. Influence of the parameter `mapred.reduce.slowstart.completed.maps` on execution time

The parameter `mapred.compress.map.output` is a boolean value deciding whether to compress the outputs of map tasks, and the parameters `mapred.map.tasks.speculative.execution` and `mapred.reduce.tasks.speculative.execution` enable the speculative execution in the corresponding map or reduce phase. Experimental results showing the impact of those parameters on the execution time are summarized in Table 4-2. From the table, when the intermediate data generated by a Hadoop job are compressed and the speculative execution is enabled, the job performs better than the scenario when all of three parameters are set to false.

<table>
<thead>
<tr>
<th>Configuration parameter</th>
<th>Execution time (true)</th>
<th>Execution time (false)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mapred.compress.map.output</code></td>
<td>433.168</td>
<td>497.326</td>
</tr>
<tr>
<td><code>mapred.map.tasks.speculative.execution</code></td>
<td>418.156</td>
<td>433.168</td>
</tr>
<tr>
<td><code>mapred.reduce.tasks.speculative.execution</code></td>
<td>417.247</td>
<td>433.168</td>
</tr>
</tbody>
</table>
4.2.4 Put It All Together

There are more than 190 different configuration parameters for each Hadoop job [21]. Among all parameters, those in Table 3-1 and Table 3-2 usually have significant influence on the performance of a job. We have evaluated the influence of those parameters on the execution time of the R-Heavy implementation. Experimental results show improvements when tuning some of parameters. As shown in Section 4.2.2, parameters interfere with each other such that when a parameter is changed to a different value, the optimal value of another parameter may change as well. For example, when the `io.sort.mb` is set to 50, the optimal performance of the R-Heavy implementation is at the time that `io.sort.spill.percent` is 0.6. When the `io.sort.mb` is set to 300, the optimal performance of the R-Heavy implementation is at the time that `io.sort.spill.percent` is 0.1. We try to minimize the interference by tuning possible related parameters together, for example, `io.sort.mb` and `io.sort.spill.percent`. However, other parameters may still influence each other. Better methodologies or more experiments are required in the future research.

In this section, a comparison is carried out on the R-Heavy implementation between the execution time prior to tuning the configuration parameters and the execution time after tuning the parameters. According to the experiments above, the values of configuration parameters after tuned are summarized in Table 4-3. When a parameter has very slight influence, or no influence, on the performance of a job, the default value in Table 4-1 is used. Otherwise, the value is chosen based on the experimental results presented in the previous sections. Figure 4-14 shows experimental results of the comparison. After tuning configuration parameters, the total execution time to process 1 TB data decreases by approximately 38%. Both the map phase and the reduce phase execution time decrease due to changing the configuration parameters. Although configuration parameters influence each other, the static tuning methodology used in the experiment still brings an improvement around 38%. 
Table 4-3. The values of configuration parameters after tuned

<table>
<thead>
<tr>
<th>Name</th>
<th>Tuned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapred.map.tasks</td>
<td>100-1100</td>
</tr>
<tr>
<td>dfs.block.size</td>
<td>1 GB</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>16</td>
</tr>
<tr>
<td>io.sort.mb</td>
<td>50 MB</td>
</tr>
<tr>
<td>io.sort.spill.percent</td>
<td>0.6</td>
</tr>
<tr>
<td>io.sort.factor</td>
<td>64</td>
</tr>
<tr>
<td>mapred.compress.map.output</td>
<td>true</td>
</tr>
<tr>
<td>mapred.reduce.parallel.copies</td>
<td>10</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>0.7</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.66</td>
</tr>
<tr>
<td>mapred.inmem.merge.threshold</td>
<td>1000</td>
</tr>
<tr>
<td>mapred.map.tasks.speculative.execution</td>
<td>true</td>
</tr>
<tr>
<td>mapred.reduce.tasks.speculative.execution</td>
<td>true</td>
</tr>
<tr>
<td>mapred.job.reuse.jvm.num.tasks</td>
<td>5</td>
</tr>
<tr>
<td>io.file.buffer.size</td>
<td>64 KB</td>
</tr>
<tr>
<td>mapred.reduce.slowstart.completed.maps</td>
<td>0.8</td>
</tr>
<tr>
<td>mapred.tasktracker.map.tasks.maximum</td>
<td>8</td>
</tr>
<tr>
<td>mapred.tasktracker.reduce.tasks.maximum</td>
<td>4</td>
</tr>
<tr>
<td>mapred.child.java.opts</td>
<td>1536 MB</td>
</tr>
</tbody>
</table>

Figure 4-14. Comparison of the execution time before and after tuning of configuration parameters to the R-Heavy implementation

4.3 Algorithmic Optimization Experiments

A sliding-window algorithm was proposed in Section 3.2 to optimize the first stage of Algorithm 1. Since both algorithms require the data to be analyzed to be collected in one node, experiments were conducted in a single machine. First, we evaluate the correctness of our approach to compute the average and the standard deviation of a set of
records. Second, we compare the execution time of the first stage of Algorithm 1 and the sliding-window algorithm.

4.3.1 Numerical Analysis of Standard Deviation Computation

Mathematically, the results of the new approach should be the same as the two-pass algorithm. However, they may vary numerically when computations are performed in computers because of rounding errors. We evaluate the approach by comparing the results of the two-pass algorithm with those of using the sliding-window approach.

4.3.1.1 Experiments set up

The machine used for testing is one of machines in the local cluster containing 8 Intel Xeon 2.33 GHz processors and 16 GB memory, with Java 1.7 installed. We use a random generator in Java standard library to generate data within the range [1, 10] and [1, 100]. Both the computation results and the execution time are compared between the two-pass algorithm and the sliding-window approach.

4.3.1.2 Results analysis

Tables 4-4 and 4-5 show the average and the standard deviation computation results for the number of records ranging from 15 to 150,000,000. Records are also randomly generated. The input consists of 15 units and each unit contains the same amount of records. The two-pass algorithm computes the average and the standard deviation for units 2 through 15 using Equation 3–1 and Equation 3–2, while the sliding-window approach assumes the results of 1st unit and units 1 through 14 are known and uses Equations 3–4 and Equation 3–5 to compute the results for units 2 through 15.

Both average and standard deviation computation results show rounding errors that increase with the number of records. However, the total number of records for each measurement unit (or dCU) usually much less than 150,000,000. The maximum error $10^{-8}$ is acceptable for the analysis system. If the number of records for each dCU increases dramatically, known numerical techniques can be used to decrease the rounding errors [38].
Table 4-4. Comparison of averages of records calculated using the two-pass algorithm and the sliding-window approach for different numbers of records

<table>
<thead>
<tr>
<th>No. Records</th>
<th>Average(two-pass)</th>
<th>Average(sliding window)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>4.220502472468472</td>
<td>4.220502472468472</td>
</tr>
<tr>
<td>150</td>
<td>4.928343844303417</td>
<td>4.928343844303417</td>
</tr>
<tr>
<td>1500</td>
<td>5.138686016771322</td>
<td>5.138686016771323</td>
</tr>
<tr>
<td>15,000</td>
<td>4.988890048565302</td>
<td>4.988890048565292</td>
</tr>
<tr>
<td>150,000</td>
<td>50.00357768201342</td>
<td>50.00357768201384</td>
</tr>
<tr>
<td>1,500,000</td>
<td>50.00160774702093</td>
<td>50.00160774702093</td>
</tr>
<tr>
<td>15,000,000</td>
<td>50.00645400430462</td>
<td>50.00645400430462</td>
</tr>
<tr>
<td>150,000,000</td>
<td>50.0016727378661</td>
<td>49.99716727378661</td>
</tr>
</tbody>
</table>

Table 4-5. Comparison of standard deviations of records calculated using the two-pass algorithm and the sliding-window approach for different numbers of records

<table>
<thead>
<tr>
<th>No. Records</th>
<th>Standard deviation(two-pass)</th>
<th>Standard deviation(sliding window)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2.553344106769994</td>
<td>2.553344106769994</td>
</tr>
<tr>
<td>150</td>
<td>3.006359919401285</td>
<td>3.006359919401288</td>
</tr>
<tr>
<td>1500</td>
<td>2.8565035556747227</td>
<td>2.8565035556747183</td>
</tr>
<tr>
<td>15,000</td>
<td>2.89133980641171</td>
<td>2.89133980641179</td>
</tr>
<tr>
<td>150,000</td>
<td>28.856446505767977</td>
<td>28.85644650576638</td>
</tr>
<tr>
<td>1,500,000</td>
<td>28.854900694855814</td>
<td>28.854900694855992</td>
</tr>
<tr>
<td>15,000,000</td>
<td>28.864628665002964</td>
<td>28.864628665002964</td>
</tr>
<tr>
<td>150,000,000</td>
<td>28.865893630970138</td>
<td>28.865893631013957</td>
</tr>
</tbody>
</table>

Figure 4-15 shows the execution time of both the two-pass algorithm and the sliding-window approach to compute averages and standard deviations. I/O is not considered in this experiment. The execution time of four cases are shown in the figure. First, we measure the execution time for the two-pass algorithm (referred to as Two-pass total). Second, the total execution time, including both the time for computing previous result and the time for computing the new average and standard deviation, of the sliding window approach are measured (referred to as Sliding-window total). Third, it is assumed that the computation results from Day-1 to Day-14 are known (referred to as Sliding-window std). Fourth, it is assumed that the computation results from Day-1 to Day-14 and each day are both known (referred to as Sliding-window daily).

As shown in the figure, if the previous computation results are known, the sliding-window algorithm outperforms the two-pass algorithm significantly. The case Sliding-window std
Figure 4-15. The execution times of the two-pass algorithm and the sliding window approach to compute averages and standard deviations.

assumes that $N$, $\mu$, $S$ and $S^2$ in Equation $3-5$ are known. The case Sliding-window daily further assumes that $N_1$, $\mu_1$, $S_1$ and $S_1^2$ in Equation $3-5$ are known. Since case Sliding-window daily has the results for Day-1, the computation time increases a little slower than the case Sliding-window std when the number of records increases. Since the sliding window approach complicates the algorithm, the execution time of the case Sliding-window total is longer than the case Two-pass total. As the number of records increases, the total execution time of two-pass algorithm is about 14 times longer than the sliding-window approach with the assumption that previous computation results are known.

4.3.2 Sliding-Window Algorithm Performance Analysis

In this section, we evaluate the first stage of the sliding window algorithm by comparing the execution time of the first stage of Algorithm 1 and the sliding-window algorithm.

4.3.2.1 Experiments set up

Several assumptions are made when evaluating the sliding-window algorithm. For simplicity, each record generated in this section contains an integer as a time stamp in the range from 1 to 15. It is assumed that the computation for records whose time
stamps are in the range [1, 14] is finished and results are stored in files. Different ordering strategies are used to store sorted records, a global-order and a time-stamp-order. In the global-order strategy, records are stored in sorted order without consideration of the time stamp. In the time-stamp-order strategy, records are first ordered by the time stamp, then ordered by the value to be analyzed. Figure 4-16 shows two ordering strategies. R0, R1, R2, ... are sorted records. Each record is associated with a time-stamp T1, T2, T3,... In the left file in Figure 4-16, records are ordered according to value without consideration of their time-stamps. In the right file, records are first sorted by time-stamps, then sorted by values.

Figure 4-16. The global-order strategy and the time-stamp-order strategy

According to different data ordering strategies and different algorithms, there are four cases to be considered, Algorithm 1 with the global-order strategy (or Alg.-1 Sort), Algorithm 1 with the time-stamp-order strategy (or Alg.-1 Time), Algorithm 2 with the global-order strategy (or Alg.-2 Sort), and Algorithm 2 with the time-stamp-order strategy (or Alg.-2 Time).

4.3.2.2 Results analysis

According to the analysis in Section 3.2.2.2, sorting and the standard deviation computation are most time-consuming computations in the first stage of Algorithm 1 and 2. Thus, the sorting and standard deviation computation time are measured separately
in Figure 4-17. We also consider the computation time versus the input/output time, as shown in Figure 4-18. The total execution time of different cases are compared in Figure 4-19. For comparison purposes, results are also shown by different cases in Figure 4-20.

![Figure 4-17. Sorting and the standard deviation computation time](image)

Based on SlidingWindowSort algorithm discussed in Section 3.2.2.1, we implemented merge sort in the case Alg.-2 Sort and multi-way merge sort in the case Alg.-2 Time. For cases Alg.-1 Sort and Alg.-1 Time, the sort method in Java is used. As shown in Figure 4-17-A, The sorting time for cases Alg.-2 Sort and Alg.-1 Sort are shortest, because most of the records are sorted. The case Alg.-2 Time performs worst. Multi-way merge sort that we implemented cannot beat the sorting algorithm implemented in Java.

The results of the standard deviation computation time are similar to the results in Figure 4-15. In Section 4.3.1.2, only the computation of the standard deviation and the average are performed on records while in this section, the stage 1 of computation algorithm is performed. From Figure 4-17-B, when most of the records are sorted, the computation time for the average and the standard deviation are shorter. In addition, the sorting takes longer time than computation of the standard deviation.

The total computation time is the combination of the sorting time and the standard deviation computation time, as shown in Figure 4-18-A. The case Alg.-2 Sort runs faster
than any other cases. As the number of records increases, Alg.-1 Sort runs approximately two times slower than Alg.-2 Sort. Since the sorting time of Alg.-2 Time is longer than other cases, it causes the total computation time of Alg.-2 Time to be the slowest case. With regards to the input/output time consumed by programs, they are approximately the same for all cases. Comparing the scales of y-axis in Figure 4-18, the I/O time dominates the program’s execution time.

Figure 4-18. Computation time and I/O time measurement

Figure 4-19. The total execution time for four cases
The total execution time of four cases are presented in Figure 4-19. Since I/O dominates the execution time, the total execution time of four cases are very close to each other. The total execution time of the case Alg.-2 Sort and Alg.-2 Time are slightly shorter than the case Alg.-1 Sort and Alg.-1 Time.

Figure 4-20. Execution time for different cases

In order to compare different parts of computation time and I/O time, all results are presented from a different perspective in Figure 4-20. All execution time for the same case are summarized in the same figure. Figure 4-20-A shows the case Alg.-1 Sort, Figure 4-20-B shows the case Alg.-1 Time, Figure 4-20-C shows the case Alg.-2 Sort and
Figure 4-20-D shows the case Alg.-2 Time. The execution time is mainly decided by I/O processing time. As shown in four cases, I/O time is at least five times longer than the total computation time. The sorting time dominates the computation time, which is clearly visible in Figure 4-20-D.

In case that experiments are influenced by CPU cache or memory, a new file is generated for each experiment. Thus, every experiment can be considered as an independent event. In addition, all results are averages of five experimental results.
CHAPTER 5
CONCLUSIONS

The purpose of this research work is to build a better statistics-based system that allows for large-scale data analysis. We have implemented the system on the Hadoop platform and the implemented system shows a better performance when compared to the legacy system. Optimization of the system is from two perspectives, tuning different Hadoop configuration parameters and optimizing the computation algorithm.

In this chapter, we conclude the research work with a summary of the thesis, a discussion about implementations and experiments and possible directions for future research.

5.1 Summary

In this thesis, we present SMMDAS on Hadoop. Chapter 1 introduces the legacy SMMDAS for the current semiconductor manufacturing process in order to show the requirements of a scalable, efficient, fault-tolerance distributed system for massive data analysis during the semiconductor manufacturing procedure. Hadoop, as a software platform that allows for massive data analysis across clusters of computers, is chosen to be the underlying platform. The background on the MapReduce programming model and the Hadoop job workflow is discussed in Chapter 1 as well. Different data mapping and implementation strategies are presented in Chapter 2. Chapter 3 discusses optimization of the system. Chapter 4 presents all experimental evaluation results.

5.1.1 Data Mapping

The SMMDAS is designed and implemented using a data-oriented approach. We propose three different methods of mapping data from the database to files: arbitrarily, per-iCU and per-dCU, and three different methods of mapping data from files to Hadoop: one-to-one, many-to-one, and one-to-many. When the size of each iCU is small, experimental results show that the per-iCU many-to-one data mapping strategy outperforms other data mapping strategies. Since the size of each iCU is small, all
computations in each iCU can be finished in memory. The per-iCU many-to-one data mapping strategy can reduce the number of map tasks launched during a job execution so that the performance of the system is improved. As the size of each dCU increases, we have observed that computations in each iCU cannot be finished in memory. Thus, an iterative MapReduce algorithm, consisting of multiple MapReduce jobs in each analysis job, may need to be considered for the system, as discussed in Appendix B.

5.1.2 Computation Distribution

To map the computation to the Hadoop platform, we propose three different implementations: M-Heavy, R-Heavy and MR-Balanced. For the M-Heavy implementation, all computations of Algorithm 1 of each iCU are performed in the map phase, while for the R-Heavy implementation, all computations of each iCU are performed in the reduce phase. In the MR-Balanced implementation, the first stage of Algorithm 1 is performed in the map phase and the second stage is performed in the reduce phase.

Different implementations can benefit different situations. Table 2-3 in Chapter 2 summarizes the limitations of different data mapping strategies on different implementations. Table 5-1 compares different implementations by a summary of advantages and disadvantages of the implementations. When the size of each iCU is small, the computation can be performed in memory in the map phase. The M-Heavy implementation decreases intermediate data transmission between map and reduce phases, thus outperforms other implementations. When the size of each dCU increases, the computation cannot fit in memory any more. The MR-Balanced implementation distributes computations between the map and reduce phase so that computations can be finished in one MapReduce job. However, the M-Heavy and the MR-Balanced implementations both require specific data mapping strategies. The R-Heavy implementation does not limit on data mapping. Thus, it is useful when designing a Hadoop-only analysis system such that the database is removed.
Table 5-1. The advantages and disadvantages of different implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-Heavy implementation</td>
<td>The execution time is shortest</td>
<td>Needs help from the database for grouping the data</td>
</tr>
<tr>
<td>MR-Balanced implementation</td>
<td>Require less memory space for each task</td>
<td>Needs help from the database for grouping the data</td>
</tr>
<tr>
<td>R-Heavy implementation</td>
<td>The execution time is longest</td>
<td>Does not need support from the database</td>
</tr>
</tbody>
</table>

5.1.3 System Optimization

The system is optimized from two perspectives. First, different Hadoop configuration parameters are tuned for the R-Heavy implementation. Second, an algorithmic optimization is carried out for the first stage of the computation. On the one hand, there are a lot of configuration parameters to each job executed on the Hadoop platform. We have analyzed the workflow of a Hadoop job in details so that parameters that may have a significant influence on the performance of a Hadoop job are tuned. On the other hand, in the first stage of the computation, the most time-consuming computations are the sorting and computing the standard deviation. We consider re-using the results of the previous computation and proposed a sliding-window algorithm to optimize Algorithm 1.

5.1.4 Experimental Results Analysis

In the legacy system, the computation performed on 1 TB data set takes around 8 to 10 hours [6]. In our experiments, the M-Heavy implementation using per-iCU many to one mapping strategy runs for 1 hour and 40 minutes for analyzing 1 TB data set. Using the static approach to tune the Hadoop configuration parameters, the R-Heavy implementation takes 1 hour and 13 minutes for analyzing 1 TB data set. Thus, performance of the system is improved by up to 82.7 % in the current experiments. In addition, results in Chapter 4 show that the execution time of the first stage computation is improved by 50 % thanks to the algorithmic optimization. The experiments, however,
reveal the fact that IO time dominates the execution time of the whole program. Future research needs to be done to improve the IO execution time.

5.2 Towards a Better System

In the legacy SMMDAS, the data collected from the manufacturing process are first stored in a file system. Then, the data are transferred from the file system to a database. Before the analysis, the data are moved from the database to another file system and computing nodes retrieve data from the file system. After analysis, the data are transferred to another database. The legacy system architecture is shown in Figure 5-1-A. The current system is to replace the legacy computing nodes with a Hadoop cluster, as shown in Figure 5-1-B. However, the data to be analyzed are still transferred from sensors to HDFS through several steps. In this section, we propose an architecture that is completely different from the legacy system architecture so that the database is removed in the system design.

As shown in Figure 5-1-C, the proposed system can be considered as a Hadoop cluster without involvement of the database. The data collected from the manufacturing process are stored directly in HDFS. As discussed in previous chapters, the R-Heavy implementation extracts useful information during the map phase and analyzes the data during the reduce phase. The SMMDAS can be built without multi-step data transmission. A background script may be required to periodically remove useless data. If functionalities of the database are required, a high-level software component, Hive, is installed on the Hadoop cluster. Hive is a data warehouse solution for large-scale data management on the distributed file system [40]. It is a high-level tool that supports a SQL-like programming language. Applications built on Hive can rely on the SQL-like interface such that low-level map/reduce tasks are automatically generated by the software component.

The proposed architecture reduces the data transmission between file systems and databases. In addition, we envision two other possible improvements.
5.2.1 Automatically Tuning Hadoop Configuration Parameters

Experiments in Chapter 4 show that Hadoop configuration parameters significantly influence the performance of the system. There are 190 or more different parameters for each Hadoop job. Manually adjustment is impossible for different jobs. Thus, it is necessary to investigate an automatic analysis system. Related works have been done towards automatic optimization of Hadoop applications [34, 41], which may inspire future works to achieve optimal configuration parameter settings.
5.2.2 Using Solid-State Drives

In Chapter 4, our experimental results show that the disk I/O time dominates the total execution time of the SMMDAS. Hadoop tries to assign tasks to where the data are located so that the data transmission time between nodes is minimized. However, the disk I/O time cannot be decreased using the Hard Disk Drive (HDD). Solid-State Drive (SSD) is proven to offer exceptional performance than HDD. Works have been done [42] to compare the performance and the price between SSD and HDD. For the sequential read applications, the performance ratio of SSD and HDD is around 2. For the random access applications, the performance ratio of SSD and HDD is around 60. In most Hadoop applications, the reading and writing take place sequentially. Other works [43, 44] also compare the performance of a Hadoop cluster using between SSD and HDD. With appropriate settings in [43], the execution time of I/O bounded applications is improved by 3 using SSD.

For the SMMDAS, the disk I/O is around 10 times more than the computation time, as shown in Chapter 4. Current cluster is constructed of HDD. Thus, if we replace the HDD with SSD and set configuration parameters of applications appropriately, the total execution time is predicted to be improved by 3. State-of-the-art technologies [44] have shown the speed up with Violin’s scalable flash memory array to HDD is 7 for sequential read applications. The write speedup is 2.5, respectively. Since the amount of read operations are much more than write operations in the SMMDAS, the speed up with the scalable flash memory array storage is predicted to be around 5.

In summary, many Hadoop applications are IO bounded [1]. Our implementations, especially when optimizing system towards real-time, also reveal the fact that IO time dominates the total execution time of the analysis system. SSDs are proven to perform faster execution time than HDD [42]. Studies have applied SSD in a Hadoop cluster [43, 44] and shown a performance improvement because of decreasing the I/O time. Thus,
upgrading hardware is another possible optimization strategy to reduce the execution time of the SMMDAS.
APPENDIX A
SETTING UP A HADOOP CLUSTER USING CLOUDERA MANAGER

This appendix shows a basic tutorial of how to set up a Hadoop cluster using the Cloudera Manager. For testing purposes, the experimental installation is taken place on VMware ESXi server on a local cluster. It starts from creating a Virtual Machine (VM), installing operating system then setting up a cluster. The VMs creation and replication can be ignored if physical machines are used for setting up the cluster.

A.1 Introduction

A typical Hadoop cluster consists of one master node and multiple slave nodes. There are multiple ways to set up a Hadoop cluster. Cloudera Manager [45] provides user an easy way to set up, manage and maintain a Hadoop cluster. In this tutorial, we set up an experimental Hadoop cluster on VMware ESXi server. The cluster consists of one master node and two slave nodes. The master node is a dedicated VM.

A.2 VM Creation and Operating System Installation

In this section, we use vSphere Client to create VMs on VMware ESXi server [46]. The details of how to create a VM on VMware ESXi server are skipped. After creating a VM, an operating system is required to be installed. We choose CentOS 6.4. Followings detail the installation of the operating system and environment setup.

A.2.1 Download CentOS

Choose a distribution image from CentOS website. We downloaded CentOS-6.4-x86 64-minimal.iso. The minimal installation is appropriate for a server, which does not contain a GUI.

A.2.2 Install CentOS 6.4

First, power on the VM that created on the cluster and then open the console.

Second, at the right most toolbar of the console, choose “connect/disconnect the CD/DVD devices of the virtual machine”.
Figure A-1. The welcome page from CentOS image

Third, choose an appropriate way to connect to the data store where the image of
the OS is stored. In the experimental setup, we downloaded the OS to the local machine.
Thus, “connect to ISO image on local disk...” is chosen to connect the VM to the local
storage.

Fourth, press any key in the console, the system will boot from the image, as shown
in Figure A-1.

Fifth, choose “Install or upgrade an existing system”.

Sixth, follow the instructions to install CentOS on the VM.

A.2.3 Configure CentOS

After installation of CentOS, reboot the system. Then configure the system
environment. To be able to make changes to system configurations conveniently, login
as the root user.

First, configure the network. Edit the script file /etc/sysconfig/network-scripts/ifcfg-
eth0. Replace contents of the file with the following:

DEVICE=eth0
HWADDR=xx:xx:xx:xx:xx:xx (Use default value)
TYPE=Ethernet
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO= dhcp

In this experimental setup, we used dynamic IP assignments from DHCP. The IP address can also be set statically. After set up the IP address, restart the network service by the command

$ service network restart

Test the network connection by the command

$ ping www.google.com

![Ping test to Google](image)

Figure A-2. Successfully set up the network connection

Figure A-2 shows the command interface if the network is set up successfully.

Second, update the installed operating system by the command

$ yum update

Third, install Perl by the command

$ yum install perl

Fourth, install NTP (Network Time Protocol is used to synchronize all hosts in the cluster) by the command

$ yum install ntp

Fifth, edit the file `/etc/ntp.conf` by replacing existing NTP servers with the local NTP servers. In this experimental setup, the NTP servers used in University of Florida are listed as follows.

server ntps2-3.server.ufl.edu
server ntps2-2.server.ufl.edu
Sixth, disable selinux by editing the file `/etc/selinux/config`.

Seventh, in order to communicate between nodes, disable the firewall by

```bash
$ service iptables save
$ service iptables stop
$ chkconfig iptables off
```

Eighth, set the `hostname` and `fqdn`. Set the hostname by editing the file `/etc/sysconfig/network`.

```
HOSTNAME=master
```

Then, edit the file `/etc/hosts`. The IP address 127.0.0.1 corresponds to the local IP while “master IP” and “slave IP” should be set to the IP addresses that are used for accessing the master node and slave nodes. If hosts cannot be resolved by the DNS, we have to specify every hosts in the file `/etc/hosts`.

```
127.0.0.1 localhost.localdomain localhost
"master IP" hello.world.com master
"slave IP" hello1.world.com slave1
```

Ninth, reboot the system by the command

```
$ reboot now
```

Tenth, install the SSH server and client. In this case, the OS distribution contains openssh server and client. If the distribution installed does not contain the SSH package, refer to CentOS SSH Installation and Configuration for installation of the SSH server and client.

Eleventh, create a SSH private/public key pair. This step can be skipped if you would like to SSH with a password. Create a SSH private/public key pair by the command

```
$ ssh-keygen
```

Copy the public key to every remote host by

```
$ ssh-copy-id -i /root/.ssh/id_rsa.pub remote-host
```
Test the SSH connection without a password by

$ ssh remote-host

A.2.4 Replicate VMs

In order to maintain the system settings at the beginning, create a snapshot of the VM that we have been configured so far. Copy two VM instances on the same cluster.

One problem that we have encountered when copying the VM instances is that eth0 is automatically mapped to the old mac address. We need to manually edit the corresponding files. The file /etc/udev/rules.d/70-persistent-net.rules should be edited from Figure A-3 to Figure A-4.

Figure A-3. The content of the file 70-persistent-net.rules before updating

```
This file was automatically generated by the /lib/udev/write_net_rules
program, run by the persistent-net-generator.rules file.
# You can modify it, as long as you keep each rule on a single
# line, and change only the value of the NAME: key.
# PCI device 0000:80:00.0 (e1000)
SUBSYSTEM="net", ACTION="add", DRIVERS="e1", ATTR(address)="00:50:56:0b:08:0:1" , ATTR(type)="1", KERNEL="eth", NAME="eth"  
PCI device 0000:80:00.0 (e1000)
SUBSYSTEM="net", ACTION="add", DRIVERS="e1", ATTR(address)="00:50:56:0b:08:0:1" , ATTR(type)="1", KERNEL="eth", NAME="eth"  
```

Figure A-4. The content of the file 70-persistent-net.rules after updating

Then, the file /etc/sysconfig/network-scripts/ifcfg-eth0 should be updated so that HDADDR is consistent with the one in the file /etc/udev/rules.d/70-persistent-net.rules

Reboot the system by the command

$ reboot now
Then edit the hostname of each node correspondingly by editing the file 
/etc/sysconfig/network and /etc/hosts.

A.3 Cloudera Manager and CDH Distribution

First, download Cloudera Manager from its website.

Second, change the permission of the package by

$ chmod u+x cloudera-manager-installer.bin

Third, run the installation package by

$ ./cloudera-manager-installer.bin

Fourth, access to the Cloudera Manager using any web browser through the IP address http://ClouderaManagerNode:7180. The default Username and Password are both “admin”. Figure A-5 shows the login web interface of the Cloudera Manager.

![Cloudera Manager Login Interface](image)

Figure A-5. The login web interface of the Cloudera Manager

Fifth, continue to the page and specify hosts for the CDH cluster installation. Type the IP addresses or hostnames of the nodes for the Hadoop cluster. Figure A-6 shows the web interface where nodes are added into the cluster. In this experiment, we added one master node and two slaves nodes to the cluster. After the initial cluster setup, new nodes still can be added to (or removed from) the cluster through the Cloudera Manager.
Figure A-6. The web interface for user to specify the nodes on which the Hadoop will be installed

Continue with the default settings. Install using Parcels, and install both CDH and IMPALA. Figure A-7 shows the web interface on which users can specify the installation package and other appropriate installation options.

Figure A-7. The web interface for user to specify the installation package and other options

Since root access is required to install CDH, specify the SSH credentials by typing the password of the root, or upload the private key of the root. Note that we tried to use a sudo user other than root with ssh private/public key access. The Cloudera Manager,
however, cannot finish the credential verification. Figure A-8 shows the web interface for users to specify the appropriate credential method to access to the cluster.

![Cluster Installation](image)

Figure A-8. The web interface for user to specify the appropriate credential method to access to the cluster

There may be failed nodes when installing CDH. Figure A-9 is an example of failed installation. In this particular case, we did not install Perl on CentOS. After installing Perl, the failed node works well. The process will continue installing with Parcels. If the installation process goes well, it should look like the interface shown in Figure A-10.

After the installation of the package, continue with Host inspectors, as shown in Figure A-11.

This process will fail if slave nodes cannot find the way back to the master node. Make sure to specify the IP address of each host if they cannot be resolved by DNS. Choose services to be installed on the cluster. Run the Inspect role assignments, as shown in Figure A-12.

Continue with the “Use Embedded Database”. Wait until all selected services have been started. After all services are installed successfully, the cluster could be started following the instruction on the web interface, shown in Figure A-13.
Figure A-9. The web interface when the installation fails

Figure A-10. The web interface when the installation process goes well

Figure A-11. The web interface when installation of the package is successful
Then, the CDH Hadoop cluster is ready to be used. Figure A-14 shows the web interface when all services started are working appropriately.
Figure A-14. The web interface showing the status of services available
APPENDIX B
A PARALLEL ALGORITHM FOR STATISTICS-BASED ANALYSIS

If the size of each iCU is small enough, the sequential algorithm (Algorithm 1) can be finished on a single node. If the size of each iCU is very large, however, the sequential algorithm cannot be finished on a single node. Then parallelization the computation is required to finish the analysis job. Many statistics functions can be parallelized. Following is the proposed iterative MapReduce parallelized algorithm.

Similar to Algorithm 1, the input of the procedure is a linked list (inputList). Since the analysis is performed independently on each iCU, the data in the same inputList share the same $K_r$. The $i$th element in the list is a 1D double precision array, $A_i$, representing a dCU. Each value in $A_i$ corresponds to an AVG value extracted from the records that share the same $K_i$. In this parallelization algorithm, each $A_i$ is further partitioned into $m$ blocks, and $A_{ij}$ denotes the $j$th block of $A_i$.

Since it is assumed that the size of each dCU is too large to be computed on a single node, the algorithm first partitions $A_i$ into several small blocks so that each block of $A_i$ can be processed on a single node. For each $A_{ij}$, the algorithm sorts the data set locally, and computes the average, summation, and summation of squares of $A_{ij}$. This procedure is done independently on each $A_{ij}$, which can be mapped to the map phase execution of a MapReduce job. Second, after all blocks in $A_i$ are sorted, computation of the average and the standard deviation of $A_i$ can be easily derived by Equation 3–4 and Equation 3–5. Also, a multi-way merge sort can be used for sorting $A_i$. Then, the algorithm computes the percentile values of $A_i$ and computes outlier limits using percentile values. This procedure can be mapped to the reduce phase execution of the MapReduce job mentioned previously. Third, the algorithm partitions $A_i$ into small blocks, and eliminates outliers from each $A_{ij}$, which can be mapped to the map phase of a new MapReduce job.

So far, the Stage 1 of Algorithm 1 is finished using the iterative MapReduce algorithm, Algorithm 3, accordingly.
Algorithm 4 starts with picking the data set whose $C_{pk}$ is largest as the reference data set. Then, it uses the average values from the previous stage to compute the average of all records belonging to the same iCU except for the reference data set. Second, the algorithm partitions each $A_i$ into several small blocks, and computes the Within-Distance and Between-Distance of each records. The procedure can be mapped to the map phase of a new MapReduce job. After computing the Euclidean distances of records, the algorithm computes the average and the standard deviation of Euclidean distances based on Equation 3–4 and Equation 3–5. Finally, the algorithm starts a pairwise Student’s T-test using previous computed results and makes a final decision.

The proposed parallel algorithm contains three MapReduce jobs and each of the MapReduce jobs are iteratively executed on the cluster. Frequently creating multiple MapReduce jobs may bring significant overhead. Twister, proposed by Ekanayake et al. [20], as one of the optimized MapReduce extensions, can be used for solving the problem.
**Algorithm 3** Procedure of equivalence test (iterative MapReduce) - Part 1

**Input:** LinkedList<Double[]> \( inputList \)

\( A_i \) represents for the \( i \)th 1D double-precision array in \( inputList \)

\( A_i[k] \) represents for an element in \( A_i \)

\( A_{ij} \) represents for the \( j \)th block of \( A_i \)

for each DoubleArray \( A_i \) in \( inputList \) do

Divide \( A_i \) into \( m \) blocks

for each block \( A_{ij} \) in \( A_i \) do

Sort \( A_{ij} \)

Compute \( \mu_{ij}, S_{ij} \) and \( S_{ij}^2 \)

end for

Compute the average \( \mu_i = \frac{\sum_{j=1}^{m} \mu_{ij} * N_j}{\sum_{j=1}^{m} N_{ij}} \)

Compute the standard deviation \( \sigma_i \) based on Equation 3-5

Merge sort \( A_i \)

Compute the percentile values of \( A_i \)

Compute the lower and upper outlier limits LOL, UOL using percentile values

Divide the sorted \( A_i \) into \( m \) blocks

for each block \( A_{ij} \) in \( A_i \) do

Eliminate all \( A_{ij}[k] \) that are not in the range [LOL, UOL]

Compute the new average \( \mu'_{ij} \)

end for

Compute the process capability index \( C_{pk}[i] \)

end for
Algorithm 4 Procedure of equivalence test (iterative MapReduce) - Part 2

Pick $A_i$ with the largest $C_{pk}$ as the reference set $R$

Compute $\mu' = \frac{\sum \mu'_{ij}}{\sum N'_{ij}} (A_i \neq R)$

for each DoubleArray $A_i$ in inputList do

Divide $A_i$ into $m$ blocks

for each block $A_{ij}$ in $A_i$ do

if ($A_i == R$) then

Compute Euclidean distance $W_{Rj}[k] = \sqrt{(R_j[k] - \mu'_R)^2}$

Compute Euclidean distance $B_{Rj}[k] = \sqrt{(R_j[k] - \mu''_R)^2}$

Compute $\mu''_{ij}, S_{ij}$ and $S_{ij}^2$

else

Compute Euclidean distance $W_{ij}[k] = \sqrt{(A_{ij}[k] - \mu'_{ij})^2}$

Compute Euclidean distance $B_{ij}[k] = \sqrt{(A_{ij}[k] - \mu''_R)^2}$

Compute $\mu''_{ij}, S_{ij}$ and $S_{ij}^2$

end if

end for

Compute the average $\mu''$ based on Equation 3–4

Compute the standard deviation $\sigma''$ based on Equation 3–5

Do a pairwise Student’s t-test $W_R$ (or $B_R$) and $W_i$ (or $B_i$)

Final decision based on the result of t-test

end for
REFERENCES


BIOGRAPHICAL SKETCH

Wenjie Zhang is born in Dalian China. She obtained her bachelor’s degree from Liaoning University (China). She joined University of Florida, Department of Electrical and Computer Engineering, in 2011, with an emphasis in computer engineering.

Since the summer of 2012, she joined Advanced Computing and Information Systems Laboratory. She has worked on IT aspects of the DARPA/REPAIR project, which is a project focusing on research of mechanisms for repair of brain injuries. She is currently involved in research on parallel and distributed semiconductor data analytics, a project of the NSF Center for Cloud and Autonomic Computing (CAC) supported by Samsung Inc. Her interests include cloud computing and large-scale data analysis in distributed environment.