Comparative Evaluation of Spark and Stratosphere

ZE NI

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:21

Comparative Evaluation of Spark and Stratosphere

ZE NI

Master's Thesis at SICS
Supervisor: Per Brand, SICS
Examiner: Associate Prof. Vladimir Vlassov, KTH, Sweden

TRITA xxx yyyy-nn

Abstract

Nowadays, although MapReduce is applied to the parallel processing on big data, it has some limitations: for instance, lack of generic but efficient and richly functional primitive parallel methods, incapability of entering multiple input parameters on the entry of parallel methods, and inefficiency in the way of handling iterative algorithms. Spark and Stratosphere are developed to deal with (partly) the shortcoming of MapReduce. The goal of this thesis is to evaluate Spark and Stratosphere both from the point of view of theoretical programming model and practical execution on specified application algorithms. In the introductory section of comparative programming models, we mainly explore and compare the features of Spark and Stratosphere that overcome the limitation of MapReduce. After the comparison in theoretical programming model, we further evaluate their practical performance by running three different classes of applications and assessing usage of computing resources and execution time. It is concluded that Spark has promising features for iterative algorithms in theory but it may not achieve the expected performance improvement to run iterative applications if the amount of memory used for cached operations is close to the actual available memory in the cluster environment. In that case, the reason for the poor results in performance is because larger amount of memory participates in the caching operation and in turn, only a small amount memory is available for computing operations of actual algorithms. Stratosphere shows favorable characteristics as a general parallel computing framework, but it has no support for iterative algorithms and spends more computing resources than Spark for the same amount of work. In another aspect, applications based on Stratosphere can achieve benefits by manually setting compiler hints when developing the code, whereas Spark has no corresponding functionality.

Keywords: Parallel Computing Framework, Distributed Computing, Cluster, RDDs, PACTs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the large-scale of data is expanded in magnitude unceasingly and inexorably, traditional sequence data processing on a single machine no longer satisfies the requirement of efficiency on one hand - a lesser amount of data participate in the computing process while the majority remains waiting until the previous participated data has been processed, for the single machine is hardly capable of handling the big order of magnitude data efficiently. One reasonable solution is that we can separate big data into multiple pieces which can be processed across multiple machines. The way of separating big data elicits other problems, including how to split big data, how to schedule the parallel processes, and how to guarantee the success of the execution. In addition, developers need to handle tedious concurrency issues. As a reaction to this inefficiency and complexity, a new programming model, namely MapReduce[7], has been proposed to execute parallel programs on large commodity clusters, hiding tedious and error prone operations of parallelization, fault-tolerance, data distribution and load balancing.

MapReduce is generally considered as a solution of many problems in vast amount of data processing job in the past. It provides the following features: locality-aware scheduling, load-balancing, and fault tolerance. MapReduce enables developers to focus on the sequential programming. The parallel is hidden in the abstraction of map and reduce and handled by framework. But MapReduce hardly deals with the cases where the formation of algorithms requires the arbitrary combination of a set of parallel operations, iterative jobs, and multiple inputs. Inevitably, running multiple times of map and reduce operations is adapted to solve the iteration problems. But reloading the same data again and again is an inefficient alternative to solve iterative problems.

Hence, two new parallel data-processing frameworks, Spark and Stratosphere[8], are proposed to overcome (partly) drawback of MapReduce from their individual concern. Both frameworks are inspired by the parallel thinking of MapReduce, simplifying the overhead of parallel programming and keeping the properties of fault-tolerance, data-distribution and load balance.

As compared to MapReduce, Spark is also capable of processing a specific class of application algorithms where a working set of data is repeatedly used for further execution or iterative job: invoking repeatedly user-custom functions over the same dataset. In the world of MapReduce, implementing the above algorithm needs to schedule as multiple jobs and in turn, each job must reload the some dataset from stable storage. To avoid needless duplication, Spark allows the dataset to be memory-resident - data sets are kept in memory - by modeling a new memory abstraction, called Resilient Distributed Datasets (RDDs). Instead of repeated I-O operation, Spark fetches the dataset once from the file system and directly accesses it from memory thereafter. Hence, it leads to a significant performance improvement in theory, but the experiment in Section 4.4.3 shows the poor result in performance. In addition, Spark provides more parallel operations (join and cogroup attached to map and reduce) than MapReduce, meeting the requirement of multiple operations in the field of interactive data analysis.

Turning the discussion from Spark to another data-intensive processing framework, Stratosphere is designed for generic parallel processing framework. Stratosphere has not only inherited synchronization, fault-tolerance, and load-balance from preferred properties of MapReduce and it also extends parallel operations like join and cogroup methods. Likewise, Stratosphere presents a programming model of Parallelization Contracts (PACTs) (PACTs are a series of second-order functions that parallelize user function) and a scalable parallel execution engine called Nephele. More attractively, by setting a set of compiling hints manually, Stratosphere compiler optimizes the execution plan, excluding duplicated shuffle operations and adopting low-cost tactics of network delivery.

In summary, MapReduce, Spark, and Stratosphere have some common features: locality-aware scheduling, load-balancing, and fault tolerance. Spark and Stratosphere extend MapReduce, providing more rich parallel operations (e.g. join, cogroup). Spark proposes the caching mechanism and supports the interactive analyses. Stratosphere proposes a set of optimization mechanisms.

Chapter 2

Background

2.1 Parallel Computing

Parallel computing is a new computational way in which multiple compute nodes participate and simultaneously solve a computational problem. Figure 2.1 demonstrates the computational process of serial computation, while Figure 2.2 presents the mechanism of parallel computing. Therefore, as compared to serial computation, parallel computing has the following features: multiple CPUs, distributed parts of the problem, concurrent execution on each compute node.

2.2 Computing Cluster

Computing cluster is a set of low-level performance servers that are interconnected by local area networks (LAN) as compared to supercomputer or High Performance Computing(HPC). Each server is assembled the unified distributed file system on their own operating system and managed by an active master server by a series step of election from all servers. As usual, in the cluster environment, there are two roles that are master server and slave servers. Certainly, master server may

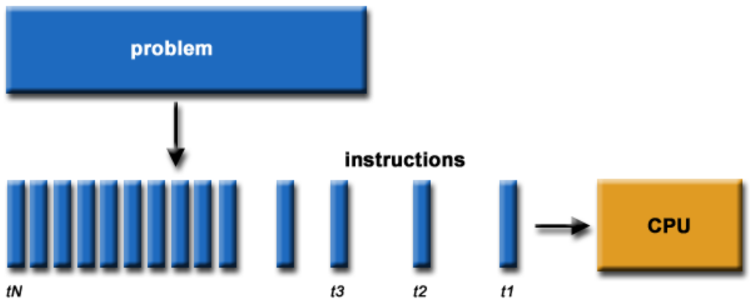


Figure 2.1. Demonstration of serial computation[1]

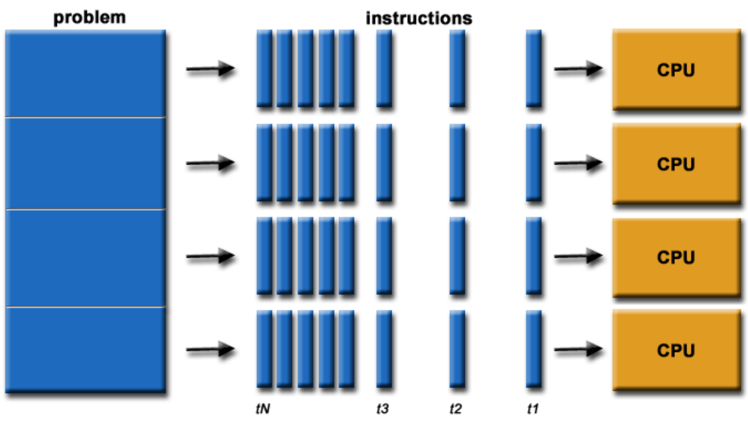
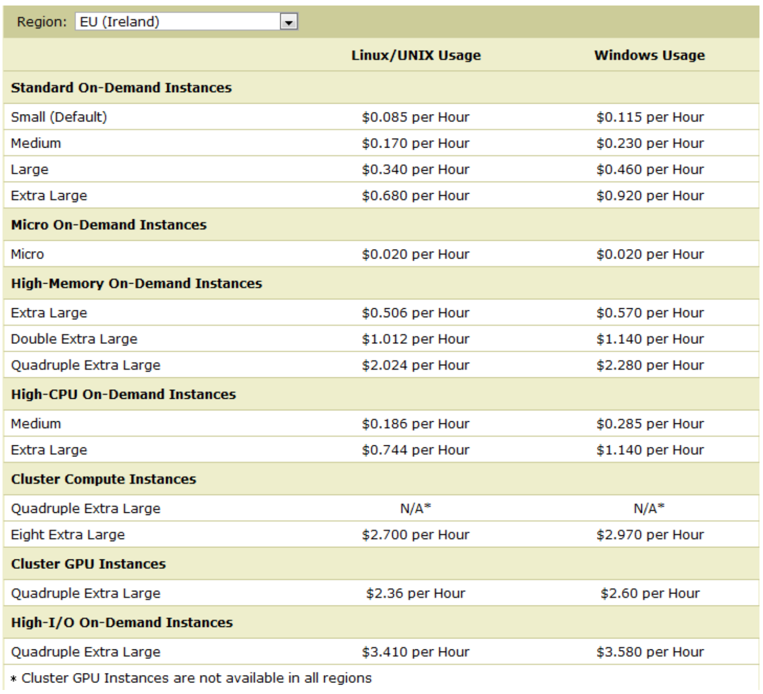


Figure 2.2. Demonstration of parallel computing[1]

be further divided to active master and standby master (passive master) in order to meet the requirement of fault tolerance and master server may be single point of failure otherwise. Generally, the set of servers can be considered as a single powerful machine and the internal mechanisms are hidden from the view of users. Undoubtedly, computing cluster has the features of relatively lower cost than HPC and more performance over ordinary servers.

2.3 Computing Cloud

Computing cloud is a set of web services that all the computing resources, no matter hardware or software, are treated as in the whole cloud architecture. Accordingly, related concepts are proposed. As compared to computing cluster, in the world of cloud, the concept of virtual machines (VMs) is proposed and makes VMs replace physical servers in computing cluster. The VMs are described as a sort of software containers that run on a general operating system. It acts as sort of sandbox that is allocated to specified number of virtual CPU, RAM, hard disk and network interface. The advantages of utilizing virtual machines ease to meet the requirement of resource uniformity in the cluster environment (All the servers are asked to have the identical hardware configuration and software on cluster servers are deployed in the way of the same directory structures). Today, one of the most known computing cloud is Amazon EC2[9]. Amazon provides the related rented web services with virtual proven compute resource in their cloud. Users only pay when they use their compute resources and count fee in terms of minutes and types of compute resources. For example, Figure 2.3 is quotation in the region of EU for renting Amazon EC2 services. Besides, applying to the dedicated cloud operator, for example, Amazon EC2, can skip the installation of dedicated cluster servers and reduce the cost of maintenance of cluster servers. When users need much higher computing resources, they are simple to choose another type with higher computing resources from cloud



Region: EU (Ireland)

	Linux/UNIX Usage	Windows Usage
Standard On-Demand Instances		
Small (Default)	$0.085 per Hour	$0.115 per Hour
Medium	$0.170 per Hour	$0.230 per Hour
Large	$0.340 per Hour	$0.460 per Hour
Extra Large	$0.680 per Hour	$0.920 per Hour
Micro On-Demand Instances		
Micro	$0.020 per Hour	$0.020 per Hour
High-Memory On-Demand Instances		
Extra Large	$0.506 per Hour	$0.570 per Hour
Double Extra Large	$1.012 per Hour	$1.140 per Hour
Quadruple Extra Large	$2.024 per Hour	$2.280 per Hour
High-CPU On-Demand Instances		
Medium	$0.186 per Hour	$0.285 per Hour
Extra Large	$0.744 per Hour	$1.140 per Hour
Cluster Compute Instances		
Quadruple Extra Large	N/A*	N/A*
Eight Extra Large	$2.700 per Hour	$2.970 per Hour
Cluster GPU Instances		
Quadruple Extra Large	$2.36 per Hour	$2.60 per Hour
High-I/O On-Demand Instances		
Quadruple Extra Large	$3.410 per Hour	$3.580 per Hour

* Cluster GPU Instances are not available in all regions

Figure 2.3. Quotation in the region of EU for Amazon EC2 services[3]

operator and instead upgrade their cluster servers.

2.4 Data dependency in parallel computing

Data dependency plays an important role in parallel computing, for the size of data set in parallel computing is massive (e.g. TB, PB). It is not efficient to transfer big data through network. In turn, as compared to massive data sets, the size of program statements is small. The thinking of data dependency is that program statements are delivered through network to compute nodes that have relevant data splits.

2.5 Load Balancing in parallel computing

Load balancing is used to evenly distribute workload across compute nodes. By using load balancing, the most optimized performance in parallel computing can be obtained ? all the tasks are roughly allocated the same amount of workload and completed synchronously.

In general, load balancing can be implemented by two approaches. One is to evenly distribute the data set across compute machines. Another one is that compute tasks are allocated by an extra component named Job scheduler. The job scheduler has an ability to dynamically distribute tasks in terms of current status of resource usage across compute machines.

2.6 Granularity in parallel computing

Granularity represents a measured degree about how large a data split is broken down. Granularity can be categorized to Fine-grain Parallelism and Coarse-grain Parallelism. Table 2.1 shows the difference between fine-grain parallelism and coarse-grain parallelism.

	Fine-grain Parallelism	Coarse-grain Parallelism
Block Size	Small	Large
Traffic frequency	High	Low
Load Balancing	Easy	Hard
Overhead and performance	More overhead	less performance

Table 2.1. Comparison between Fine-grain parallelism and Coarse-grain parallelism

2.7 MapReduce

MapReduce is a programming model proposed by Google to solve the complex issues of how to compute in parallel, distribute big data in balance, and implement fault tolerance over a large scale of machines. To achieve the goal, there are two primitive parallel methods (map and reduce) predefined in MapReduce programming model. Both map and reduce are user-specified methods and hence users empower actual function against input data. Map that is fed on a set of input key/value pair can generate a set of intermediate key/value pair. And then the MapReduce framework is accountable to pass a set of intermediate key/value pair with the same intermediate key to reduce. Reduce merges all intermediate values with the same intermediate key.

In the context of distributed processing, it is often mentioned two aspects: fault tolerance, locality. How does the MapReduce programming model develop features? First and foremost, fault tolerance in MapReduce is provided from three aspects: worker failure that can be monitored by passing heartbeat message periodically, master failure that can be guaranteed by the approach of checking point, map and reduce in the presence of failures that are solved by atomic commits of map and reduce task outputs. Checking point is an approach for applications to recover from failure. By taking the snapshot of current intermediate results, resource allocation and related information to file system, application can be rolled back to certain point of checking point in timeline. Although the way of utilizing checking point has the above benefit, checking point, on the other hand, need frequently update for synchronization on all servers. Secondly, locality is relied on distributed file system

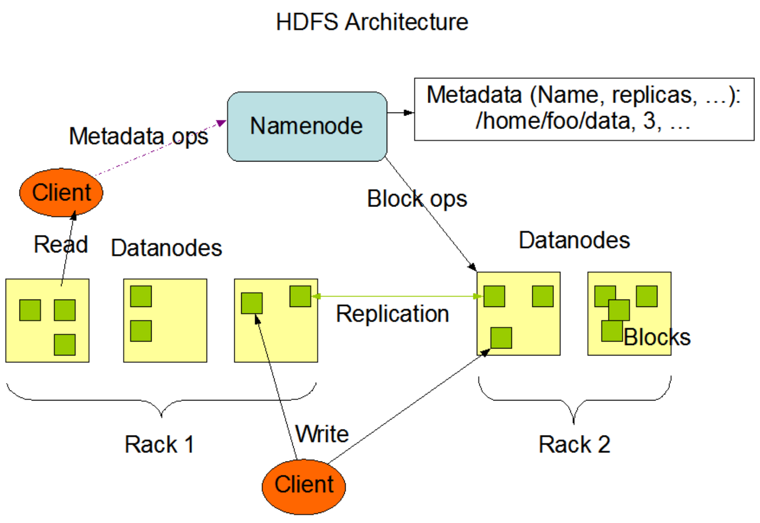


Figure 2.4. HDFS architecture[4]

(GFS/HDFS) that can fairly divide input file into several splits across each worker in balance. In another word, MapReduce is built on the distributed file system and executes read/write operations through distributed file system. The next section will introduce more concrete principle and features of distributed file system by one of the most representative distributed file system, HDFS.

2.8 HDFS

HDFS[10] is an abbreviation of Hadoop[11] Distributed File System that is deployed on computing cluster. HDFS is open source project of Google File System (GFS). As considered that GFS is proprietary, in this thesis, we will rely on HDFS at the later sections of theory explanation and experiments. By using HDFS on the cluster, the HDFS are known as the following features: fault tolerance, data integrity, high throughput, large dataset, streaming access, data load balance and simple coherency model, "Moving Computation is Cheaper than Moving Data", and portability across heterogeneous hardware and software platforms. The HDFS resorts to simple master-slaves architecture that ease to design and implement. Based on this architecture, any crashed slave node can be checked out by sending heartbeat messages periodically to master node. Once certain slave node fail to send heartbeat message to master node within regulated time period, master node would mark that slave node inactive. Then, slave servers are asked to replicate data. Another case in HDFS that may be asked to replicate data occurs when free space on certain servers are below the threshold specified on the configuration file, those data on overloaded servers can be redistributed to other data servers. It would ensure that data on HDFS is distributed fairly on each data slave servers in order to prevent certain data servers overloaded. Besides, for distributed file system, robustness is also considered through replicating multiple pieces of data across various servers. In addition, one more thing in the respect of fault tolerance is referred. The principle of fault tolerance in HDFS refers to checkpoint and then HDFS recovers data from checkpoint at certain time point when certain slave server is crashed down.

HDFS is not used to work alone. In general, on top of HDFS, parallel applications can retrieve data from HDFS and do computational procedure and finally write results back to HDFS. On the cluster environment, the purpose of deploying the HDFS is to enable data distributed on various servers manipulate like all data are stored in a single hard disk.

2.9 Dryad

Dryad is a general-purpose parallel programming model (execution engine) for developing distributed parallel applications. It provides more parallel operations than MapReduce and supports for entering multiple parameters of inputs and achieving many outputs. In addition, Dryad generates a flexible execution plan by a wide range of parallel operations (e.g. Filter, Aggregate).

The major difference between Dryad and Spark is the same difference between Stratosphere and Spark, in that Dryad cannot efficiently solve the problems of iterative algorithms. In another respect, the difference between Dryad and Stratosphere is that Stratosphere can perform better automatic optimization than Dryad. In addition, by referring to the infrastructure of Stratosphere and Dryad, Stratosphere has one more level of abstraction than Dryad (PACTs) through which Stratosphere can provide a suitable parallel data flow and choose the best shipping strategy.

2.10 Hyracks

Hyracks is a parallel computing framework that is used to simplify the development of parallel applications. It also provides multiple operators (mapper, sorter, and joiner) and connectors (Hash-Partitioner, Hash-Partitioning Merger, Range-Partitioner, Replicator, and Connector). The difference between Hyracks and Spark is that Hyracks does not support iterative algorithms like Spark. As compared to Spark and Stratosphere, Hyracks is more compatible with Hadoop. The difference between Stratosphere and Hyracks is that Stratosphere provides one more abstraction (namely PACTs) through which Stratosphere can generate more efficient work flow. By taking advantage of Hyracks-to-Hadoop Compatibility Layer, Hyracks can execute existing Hadoop programs.

Chapter 3

Spark and Stratosphere Parallel Programming Models and Execution Engines

It is known that a single computer cannot efficiently process massive data sets, since the single computer has relatively limited resources. One reasonable approach is to distribute multiple tasks to different machines and coordinate them to complete all the tasks. This leads to more complexity in the computational process, including the scheduling of tasks, task synchronization and so forth. Parallel programming models like Spark and Stratosphere are designed to overcome the complexity by hiding process communication and problem decomposition.

Process communication can be implemented by two approaches: shared memory and message passing. For instance, OpenMP[12] is multi-platform shared-memory parallel programming system and Open MPI[13] is a message passing system with the features of fault tolerance, many job scheduler support, high-performance on all platform.

Problem decomposition describes how to formulate processes, classifying task parallelism or data parallelism. Task parallelism emphasizes how to associate processes with sub-problems, whereas data parallelism focuses on how to associate compute nodes with split data. The job schedulers of Spark and Stratosphere plan the job graph by the approach of task parallelism. The execution plans in Spark and Stratosphere show the allocation of the compute nodes in the way of data parallelism.

In the Section 3.1 and Section 3.2, we will present respective unique features of Spark programming model and Stratosphere programming model. In the Section 3.3, we will compare their common concepts in their programming models and similar mechanisms in their execution engines.

3.1 The Spark Programming Model

The spark[14] programming model is inspired by the parallel abstraction of MapReduce as discussed in the Introduction Chapter. Keeping favorable properties of MapReduce, such as parallelization, fault-tolerance, data distribution and load balancing, Spark adds support for iterative classes of algorithms, interactive applications and algorithms containing common parallel primitive methods like join and match.

3.1.1 Resilient Distributed Datasets (RDDs)

When distributed applications on top of MapReduce are developed to solve iterative problems, intermediate results cannot be directly used by the following iterations. They need to be flushed to hard disk and reloaded at the next iteration. This leads to frequently I-O read/write operations and hence results in the inefficient performance. In another aspect, the MapReduce loop control is implemented by an outer driver program. The driver program could be another Java class containing a for loop statement. There is an invocation to the MapReduce program inside the for loop statement.

Formally, RDDs[15] are read-only, collections of objects partitioned across a group of machines. It resembles Distributed Shared Memory (DSM) by preserving calculated results to the memory. RDDs can be understood as read-only DSM. RDDs can be only created/written in four ways in Spark:

1. Importing a file from file system.

2. Making Scala collection slice distributed pieces by calling the method "parallelize".

3. Transforming existing RDDs to another.

4. Adjusting persistence of existing RDDs. It has two options: cache action and save action. The cache action keeps data in the memory, whereas the save action keeps data in the distributed file system.

By using RDDs, intermediate results are stored in memory and reused for further performing functions thereafter, as opposed to being written to hard disk. Especially in the case of looping jobs, the performance would be improved. In Spark, users can manually specify if working sets are cached or not. The runtime engine would manage low-level caching mechanism like how to distribute cache blocks.

RDDs can be organized by the Spark scheduler to form a directed data flow. The direct data flow of Spark is different from the data flow of the traditional computing framework like MapReduce. The data flow of Spark can integrate looping flows through which intermediate results can be reused. Therefore, developers need not append extra driver programs to control iterations.

3.1.2 Parallel Operations

Map-reduce operations are proven to implement many algorithms in common application domains. For example, Distributed Grep, Count of URL Access Frequency, Reverse Web-Link Graph, Term-Vector per Host, Inverted Index, and Distributed Sort[7]. But, to implement algorithms that consist of complex operations (for example, join, group) by using MapReduce, developers have to construct a set of successive map-reduce operations to express the complex parallel functions like join, group and so forth. To overcome this issue, Spark extends MapReduce, providing with more parallel primitive methods. These parallel operations are summarized to two types: transformations, which are operations to create a new RDD from an existing RDD, and actions, which return a value to a driving program or output to stable storage.

All the parallel operations can be freely combined and integrated to a job graph. Thus, developers do not have to abide the fixed order (map-reduce) like MapReduce and can arbitrarily arrange the execution order. For example, outputs of Map method from two different data source can be joined by Join method and passed to Reduce method for aggregating results.

3.1.3 Shared Variables

Shared variables can be viewed as "global" variables. We consider the variables that are usually not updated in remote machines as shared variables. It is meaningless to propagate them back to the driver program as parallel methods run. Spark exposes two types of restricted shared variables:

Broadcast Variables are mainly used to target hot-data which is frequently fed into the computation by only once broadcasting values to participating nodes rather than carrying those values with function closure every time. The way of using broadcast variables eases data distribution and optimizes network performance, by consuming less network transfer time and reducing duplicated network traffic.

Accumulators are global variables that can be changed by triggering associated operations ("add" operations) with a parameter. Hence, accumulators can be operated in parallel. By using accumulators, Spark users can implement the same counters as in MapReduce. By default, Spark provides the implementation of "add" operation with one parameter of type Int and Double. Developers can overload other "add" operations with different type of parameters.

3.2 The Stratosphere Programming Model

Stratosphere programming model inherits from MapReduce, providing more primitive parallel operations and optimized mechanism to improve efficiency. The Stratosphere scheduler generates acyclic data flows (details in Section 3.3.1) and guide execution engine knows how to organize and run tasks. The main abstraction in Stratosphere is Parallelization contracts[16][17][18] (later called PACTs for simplification), classifying three types: Input Contract, User Function, Output Contract. PACTs are data processing objects in the data flow. Splits of input data go through nodes for transformation, computation, and optimization in parallel.

3.2.1 Input Contract

Input contracts mainly reflect how inputs to compute nodes are structured. They belong to second-order functions in functional programming used to handle task-specified functions. Like Spark, PACTs in Stratosphere include classical map and reduce as well as cross, match, and so on. In addition, those multiple parallel operations are also allowed to assemble at any logic order of combination.

3.2.2 User Function(UF)

UFs are functions where developers can program how to transform from inputs to outputs. In the UFs, developers can design how the data should be formatted, how to do computations, and what form of outputs are generated.

3.2.3 Output Contract

Output contracts are optional components acting as compiler hints to achieve performance improvement. This way of optimization in Stratosphere will be revealed in the Section 3.3.2. At present, Stratosphere lists three keywords for output contract: SAME-KEY, SUPER-KEY, and UNIQUE-KEY. SUPER-KEY and UNIQUE-KEY are analogous semantics as in SQL. SAME-KEY signifies the keys of datasets are not changed through a PACT node ranging from input to output.

3.3 Comparing Spark and Stratosphere Programming System

We compare Spark and Stratosphere in the following four aspects: data flow model, optimization mechanism, parallel methods, and fault tolerance.

3.3.1 Directed Data Flow Model

Data flow model provides a straightforward approach to construct the job graph where each node represents well-defined functions about how to transform data and compute intermediate result. It can be categorized two classes: cyclic data flow model and acyclic data flow model. Although the cyclic data flow model has more benefit in the handling the looping algorithms than acyclic data flow model, the cyclic data flow model requires the smarter job scheduler by which a feasible executable job graph could be made. But the job scheduler that could satisfy the cyclic data flow model requires the ability to analyze the algorithms and terminate the execution at the right time. For this reason, Spark and Stratosphere develop their job schedulers with acyclic data flow. But there is one difference between their job schedulers about the acyclic data flow. Stratosphere is derived from traditional parallel computing framework (e.g. MapReduce). The acyclic data flow in Stratosphere is a linear job graph. In turn, Spark references to partial idea from cyclic data flow and generate a linear graph that mixes with partial loop. The Spark data flow can solve problems that traditional parallel computing frameworks need an additional driver program for iterative algorithms. Spark supports continuous interactive operations by taking advantage of a modified Scala shell. On the other hand, the Stratosphere data flow model is a single linear job graph and hence eases to achieve the maximum optimization.

3.3.2 Optimization Mechanism

As the Section 3.3 referred, Stratosphere defines a set of rich optimization mechanisms in PACTs compiler as well as output contracts. By contrast, Spark lacks of related optimization strategies. For this reason, we will mainly focus on the discussion of Stratosphere's optimization mechanisms.

To unveil the mysteries of optimization, we firstly start from the PACTs compiler which consists of two components: the Optimizer and the Job Graph generator. The job graph generator will not be deeply explored, because it does not associate with Stratosphere optimized mechanisms and merely accomplishes the transition from optimized plan to Nephele DAG (a represented form for Stratosphere execution engine).

Now, we start to explore Stratosphere optimizer. At the preliminary phase of optimization, the Stratosphere optimizer estimates the size of outputs on each node. The estimation is regarded as the cornerstone for the later phases. According to the size of outputs generated at the previous phase, the optimizer selects the cheapest strategy such as Shipping Strategies or Local strategies. More specifically, in the case of two data sources - one is much large inputs and nonetheless another is tiny inputs, the optimizer may generate the plan: one with large inputs enforces local forward strategy (data transportation through in-memory channel), whereas another one with tiny inputs carries out broadcast strategy (data transportation through network channel). By optimizing shipping strategy, we could reduce network traffic and gain higher efficiency. Another optimization is built on output contracts through which key-based data partitioning is kept track of on each node. If there are a series of data partitioning in the Stratosphere program, developers can append several compiler hints in output contracts and guide the optimizer that some keys are not changed. By using output contracts, the optimizer will perform the data partitioning at the first time and successive partitioning operations can be ignored. In a word, the optimizer helps Stratosphere minimize data partitioning. Data partitioning usually leads to large number of data over network and the large number of data through network is one of main bottlenecks in parallel computing framework. Therefore, the optimizer may speed up the completion of Stratosphere programs.

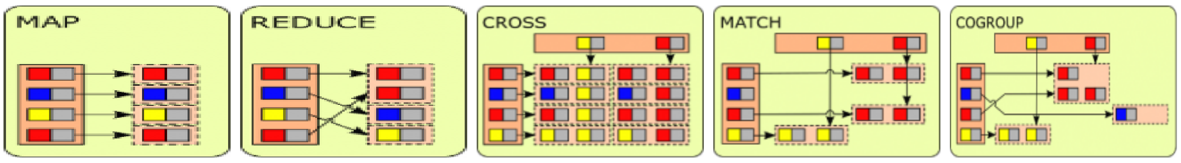


Figure 3.1. Typical parallel methods[5]. Left split in a record is KEY and right split in a record is VALUE.

In summary, Stratosphere allows users to manually provide optimized options whereby Stratosphere can be guided to achieve the best shipping strategy and reduce the amount of data through network. At this context, Stratosphere would shrink the execution time. Reversely, Spark has only fixed shipping strategies.

3.3.3 Parallel Methods

The Table 3.1 summarizes most parallel methods that present the same functionality at each row. Each functionality of parallel method at each row is described in Figure 3.1. Apart from the APIs in Table 3.1, Spark provides more parallel methods than Stratosphere, such as map, union, a series of methods based on keys. In Spark, map represents 1-to-1 mapping methods, which means one form of input can be transformed to another form of input. The difference of map and flatMap is that map is used to map 1-to-1 relationship but flatMap reflects 1-to-* relationship - 1 input is transformed to multiple pieces of output. The union can be understood as merge methods, for example, that {1} and {2} goes through union method is transformed to {1,2}. The groupByKey in Spark is similar to cogroup in Stratosphere but groupByKey is asked to have the same key of input data in advance. The reduceByKey is applied to the same principle as compared to reduce.

Spark	Stratosphere	MapReduce
flatMap	map	map
reduce	reduce	reduce
map		
union		
filter		
cartesian	cross	
join	match	
groupwith	cogroup	
groupByKey		
reduceByKey		
sortByKey		

Table 3.1. The comparison of Spark and Stratosphere in parallel API

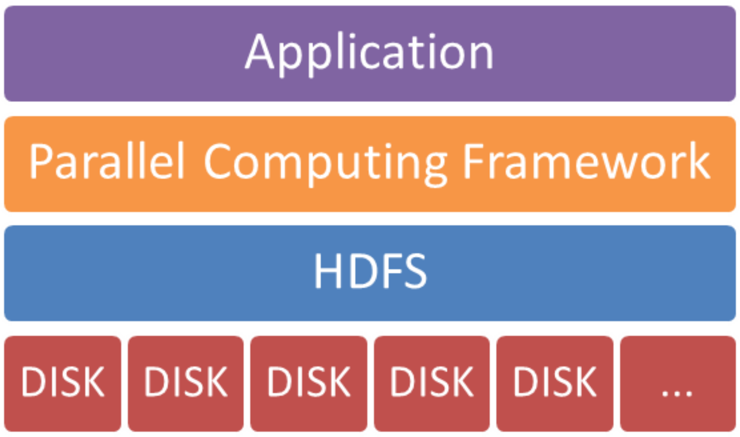


Figure 3.2. General parallel computing architecture

3.3.4 Fault Tolerance

Fault tolerance is one of main factors emphasized in distributed environment. What approaches are there in distributed environment for achieving stable enforcement (fault tolerance)? Before account for those approaches, it is good to supplement the knowledge about the architecture of distributed computing. The Figure 3.2 depicts the infrastructure. At the lowest level, each disk represents one node where there are certain computing resources as well as stable storage devices. On the upper layer of DISK, it is HDFS that is distributed file system, hiding complex low-level communication among nodes for synchronization, file indexing and so forth. To ascend from HDFS, that is the parallel computing framework, for example, MapReduce, Spark, and Stratosphere. Finally, all the parallel applications are built on the parallel computing framework.

In general, the components related to fault tolerance are the parallel computing framework and the HDFS in the hierarchy. The problem of fault tolerance in HDFS is solved by replicating multiple copies of data source across different nodes, which is not the concern in this thesis. Next, the mechanism of fault tolerance in the parallel computing framework will be discussed. The goal of fault tolerance in the parallel computing framework is to guarantee the application completion as well as distributed tasks scheduled by job scheduler as soon as possible even when some node accidentally fails during execution. Subsequently, we will explain the fault tolerance mechanism of parallel computing framework from two perspective views: fault tolerance in Spark and fault tolerance in Stratosphere.

In the respect of Spark, the default way of keeping fault tolerance is so called lineage that contains enough information about relations among RDDs. By means of lineage, the missing piece of RDDs can be recovered by re-computing or rebuilding from previous RDDs when some node fails in the process of execution. The approach of lineage can, in many cases, solve problems of fault tolerance and efficiently keep the ongoing execution. The RDD lineage is the default approach for fault tolerance. Besides lineage, Spark can develop another approach for fault tolerance: check pointing. Spark exposes an API for manually activate the checkpoint approach to replace the lineage approach. Therefore, it is of interest to elaborate respective strengths and weaknesses. Lineage like Meta information occupies little space for depositing relational records of RDDs. As relational records, they seldom get involved in modifying operations unless the relations between RDDs happen to change. As a result, there are no many update messages sent across networks. In another respect, the checkpoint approach deals with a great deal of update messages and needs to keep different versions of snapshots at different time point for rolling back, which naturally leads to the demand of relative massive storage space. Even though the checkpoint approach spends more storage space than lineage, it can achieve quicker data recovery from erroneous nodes over the lineage approach, especially in the case of the long lineage. In short, Spark makes use of the lineage approach or the alternative checkpoint approach to gain the fault tolerance.

Another parallel computing framework Stratosphere gains theory progress or experimental exceptional handling on fault tolerance when this thesis is being written. The way of handling fault tolerance in Stratosphere is dependent on the checkpoint approach. More deeply speaking, file channel - one of three types of shipping strategies and the rest of them are network channel and in-memory channel - is used as a channel to flush intermediate result to stable storage before passing the temporary result to next task in execution plan. Despite it may be a feasible approach for maintaining fault tolerance, the latest distribution of Stratosphere has no support for fault tolerance. Therefore, Stratosphere programs would throw an exception of data accessing unavailable if some slices of intermediate results are missing. What follows then, Stratosphere has to re-run the program from start.

Nowadays, the big data processing will consume more and more time, like a few hours, a few days or even one month. In the distributed environment with a large number of computing nodes, it is irritable that the applications have to start over merely because of small part of intermediate results missing. This reflects the importance of fault tolerance in the distributed world or an unstable computing environment.

3.3.5 Job Scheduler

Parallel programs are built on the basis of parallel computing framework, invoking APIs the parallel computing framework provides. And the job scheduler of parallel computing framework is in charge of how the programs runs across nodes and supervise the progress of the programs.

As mentioned above that RDDs are the foundation of all things in Spark, the Spark job scheduler is responsible for forming a series of DAG stages based on the RDDs lineage graph by grouping pipeline transformations and leaving shuffle operations as in-between boundaries. In short, a set of indexed stages come into being the execution plan by job scheduler. Here it is an interesting question why stages are proposed and consisted of the execution plan. The purpose of separating a job into a few stages is to ensure available usage of resource instances. In the context of parallel computing, the DAG is represented as the sequential execution workflow, which implies some latter operations will be suspended until the completion of early operations. After that, the intermediate results from the early operations are regarded as inputs to the latter operations. Therefore, the most optimized scheduling approach is to allocate partial computing resource to early tasks. And then, when the prerequisite of the latter operations about computing resources are satisfied, the job schedule launches the resource allocation for latter operations. The dynamic way of allocation brings in two aspects of benefits. Firstly, it pays attention to the cost-efficient factor, especially in the case of pay-per-usage paradigm. Secondly, it could reduce the risk of running out of computing resources and then lead to the exception of no available computing resources. Except for stages, in order to uniform names and exclude ambiguity, Spark defines two new terms that can be represented in DAG: the relation between pipeline transformations is called narrow dependency, and the relation between shuffle operations is called wide dependency. The distinction between wide dependency and narrow dependency embodies in the efficiency of fault tolerance. When a node fails, the missing partitions within narrow dependency can be recomputed through the parent RDD of the failed RDD. By contrast, if the missing partitions are placed within wide dependency, all the ancestors of the failed RDDs have to participate to recover the missing partitions. The job scheduler can be specified the parallel degree of tasks on each RDD in the Spark programs and allocate enough computing resources.

At the second place, what functionality of the job scheduler possesses in Stratosphere? The Stratosphere job scheduler also makes the execution stage as well as has the same purpose as Spark. However, the rules of formulating the execution stages in Stratosphere are different from in Spark. The stage separation in Spark is based on shuffle operations, whereas the stage separation in Stratosphere three types of channels: network channels that exchange data via a TCP connection, in-memory channels that exchange data via main memory, and file channels that exchange data via local file system. The Stratosphere job scheduler binds the subtasks that connect with network channels or in-memory channels within the same execution stages, whereas subtasks that connect with file channels must locate in the different level of stages. The last feature of Stratosphere job scheduler is that developers can manually specify the parallel degree of tasks on each node.

3.3.6 Resource Management and Execution Engine

Resource management and execution engine are used to manage the allocation of computing resource, distribute parallel programs across nodes, keep track of task scheduling, and parallelize subtasks in execution. The subsequent paragraphs in this chapter explore the internal structure of execution engine or resource management system of Spark and Stratosphere.

Spark integrates with Apache Mesos[19]. The Apache Mesos can not only pro-

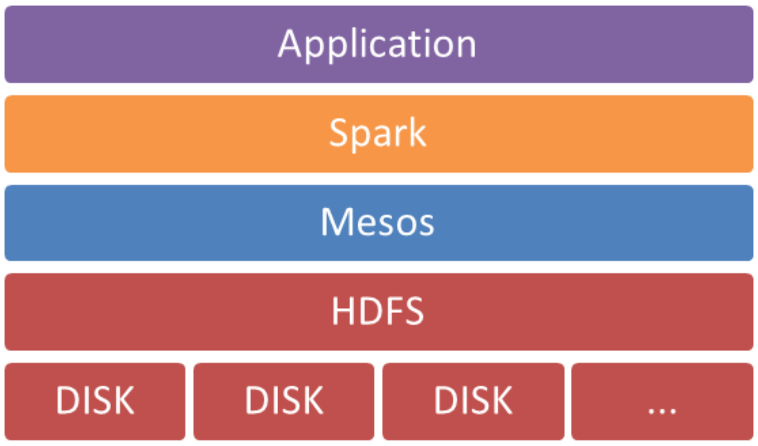


Figure 3.3. Spark architecture

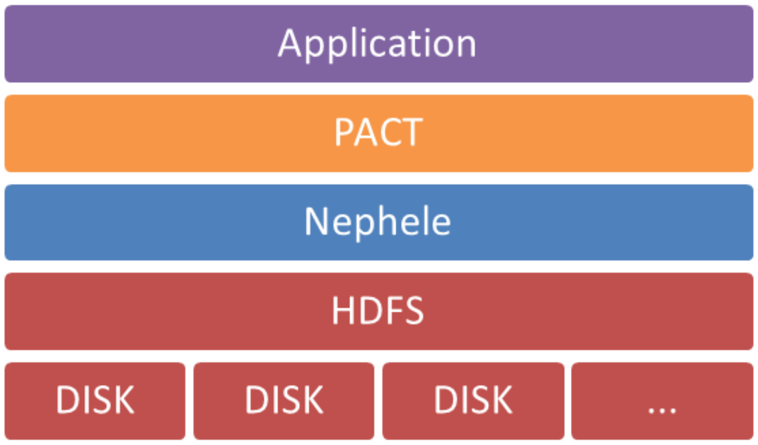


Figure 3.4. Stratosphere architecture

vide the resource management for Spark but also associate with other parallel computing frameworks such as Hadoop, MPI to take advantage of data from other parallel computing frameworks. Mesos is used for dynamic resource sharing among multiple frameworks, multiple frameworks isolations, and resource distribution. In a word, Mesos is a universe platform of cluster resource management. The Figure 3.3 demonstrates the parallel computing framework and decomposes the subcomponents of parallel computing framework.

Similar to the engine of resource management and execution in Spark, Stratosphere develops a dedicated engine, namely Nephele[20]. The infrastructure of Stratosphere is illustrated as below Figure 3.4.

Although Spark is built on Mesos, Mesos is an independent project from Spark. Spark splits the functional component about allocating/de-allocating resource in-

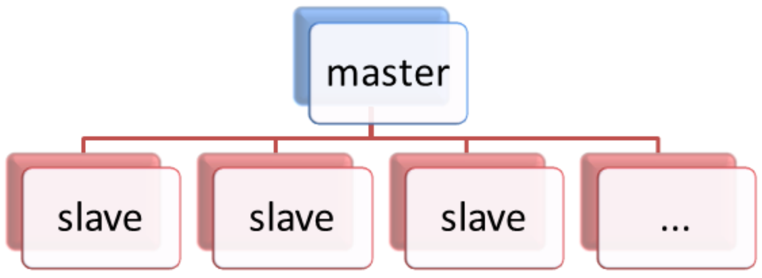


Figure 3.5. Master-slaves architecture

stances and takes advantage of Mesos for resource allocation in the cluster environment. The job plan generated from Spark is fed to Mesos, and then Mesos keeps track of the rest of the execution. Mesos and Nephele draw lessons from master-slaves structure where the single master node receives a job graph from the upper layer (Spark or PACT) as ground to apply for resources by communicating with resource administrative unit that is used to manage computing resource of slave nodes. When adequate resources are available for certain job, the master node distributes tasks to slave nodes and keeps track of their progress such as initialization, running, completion and so forth. In addition, distributed tasks are resided in different slave nodes which may have multiple tasks to execute at the same time. The Figure 3.5 can be a more intuitional way to clarify the connection between master node and slave nodes.

The master-slaves structure possesses the following features: simple relation among nodes, clear division of work, and a small quantity of communicative messages. On the contrary, the single master node is prone to being single point of failure. The solution to this problem is to provide redundancy masters. Once the active master has failed, one of passive masters can immediately replace the active master and then become the new active master. Periodically, all masters must synchronize their information and thus any master can be safely shifted from each other. The above-mentioned algorithm of solving the single point of failure has been implemented in ZooKeeper[21] through which all masters can coordinate with one another and make decision of electing the primary master (aka, consensus).

The chief difference in the aspect of resource management and execution engine between Spark and Stratosphere is that Spark takes advantage of external resource management component and internal execution engine, whereas Stratosphere encapsulates resource management and execution engine to Nephele. It results in a little difference in the following utility. As a universal resource, Mesos can be compatible with current multiple frameworks such as Hadoop, Spark, and MPI[22] and takes advantage of data generated by other parallel computing frameworks, whereas Nephele is dedicated for Stratosphere to execute the parallel programs and cannot work with other parallel computing frameworks . From another standpoint, Nephele can be seamless integrated within Stratosphere framework and seldom occur the compatibility issues when Stratosphere is installed. However, the installation of Mesos and Spark might happen to incompatible problems.

Chapter 4

Comparative Evaluation

After theoretical parallel programming models are compared, this chapter will draw our attention to experiments. First of all, we will briefly introduce our cluster environment, its hardware configuration and software setup. Then, we will present set of benchmark scripts we developed. At the last step, we will select three types of applications: classic MapReduce example, iterative machine learning example, and relational query example. By using our benchmark scripts, we will compare and evaluate the performance of these three applications running on Spark or Stratosphere.

4.1 Experiment Environment

Hardware

Our compute cluster consists of 7 servers that are interconnected. Each of them has 12 CPU and 32 GB RAM. One server is elected master and the rest are slaves. Our compute cluster is a shared user environment.

Software

Spark (version 0.3 for Scala 2.9), Stratosphere (version 0.1.2), Hadoop (version 0.2.0), the set of benchmark scripts (we developed). In order to evaluate Spark and Stratosphere fairly, the configuration of both frameworks must be unanimous in their respective configuration files. We allocate 12 CPU processors and 3 GB memory on each node for virtual experiment to avert the resource contention. All the software is installed in the identical hierarchy on each node (e.g. "/home/USERNAME/stratosphere").

4.2 Benchmark scripts

We developed a set of new independent benchmark scripts rather than directly use internal benchmark programs of Spark or Stratosphere. For our experiments, we observe the execution time of application running on Spark and Stratosphere and CPU usage on each node. But, their internal benchmark programs of Spark and Stratosphere cannot satisfy our requirement. Our benchmark scripts are based on Linux Bash and consist of exclude.sh, start_top.sh, stop_top.sh, extract.sh, collect.sh and plot.sh.

1. The exclude.sh is used to detect and record the ID of running Java programs before Spark and Stratosphere programs startup. It is implemented by invoking "top -u $USERNAME -n 1 -b | grep java | awk 'print $1' > $OUTPUT-FILE".
2. The start_top.sh is used to startup the monitor application. The monitor application is Linux TOP that would flush TOP information to a log file per 5 second at each slave. It is implemented by invoking "top -d 5 -b > $OUTPUT_TOP &" and "ps -ef | grep -v 'grep' | grep 'top -d 5'".
3. The stop_top.sh is used to shut down the monitor application by Linux Kill.
4. The extract.sh is used to extract relevant CPU metrics (%CPU in $OUTPUT_TOP) by Linux Grep.
5. The collect.sh is used to copy CPU metrics on all the slaves to the master by Linux Scp.
6. The mplot.sh is used to plot figures about CPU, execution time, number of iterations, and size of inputs by gnuplot.

4.2.1 Methodology

We developed another Bash script, namely integrate.sh, which schedules our benchmark scripts. The integrate.sh is also our methodology in our experiments.

1. Running exclude.sh on each slave.
2. Running start_top.sh for monitoring the status on each slave.
3. Running user application. In experiments, once monitor application starts up, user applications are activated immediately.
4. Switching off the monitor application on each slave by running stop_top.sh. By the completion of user application, it is triggered to shut down the monitor application and output stream to local logs on all slaves.
5. Performing Extraction by running extract.sh.
6. Coping CPU metrics from slaves to the master by running collect.sh.
7. Plotting CPU changes over time on each slave by running plot.sh.

The benchmark workflow is considered as the methodology of all our experiments. Therefore, we will not restate the same execution procedure in each experimental example.

4.3 Classic MapReduce example

Since both Spark and Stratosphere are inspired by MapReduce, it is good to implement one MapReduce algorithm. We selected WordCount as the MapReduce example, for it is a prime example from literature and provided by MapReduce, Spark and Stratosphere.

4.3.1 WordCount

The WordCount is an application that counts how often each distinct word appears in given text, file or document collection.

4.3.2 Data Source

Test data is generated by Linux random generator (/dev/urandom). We generate four diverse sizes of inputs: 2 GB, 4 GB, 6 GB and 8 GB and deploy them to HDFS with block size 64 MB.

4.3.3 Algorithm Description(in MapReduce)

1. **Map phrase:** each node performs data transformation, data cleansing, and data output as the specified form. More specially, set delimiters for each "word" in the input stream by interjecting a spacebar between words, transform upper case letters into lower case ones, split words with the spacebar delimiters, filter meaningless blank words and emit output as a form of (word, 1).

2. **Shuffle phrase:** outputs from the map phrase are shifted through network in terms of hash-based shipping strategy. The hashed value of "word" in (word, 1) is calculated by default hash function. According to the hashed value, (word, 1) is shifted to the specified a compute node.

3. **Reduce phrase:** each node aggregates the amount of data that share the same key. For example, node 1 receives (word, 1), (goodbye, 1), (word, 1), (goodbye, 1), (hi, 1). The outputs of node 1 are (goodbye, 2), (hi, 1), (word, 2)

Since Spark and Stratosphere are a superset of MapReduce, they can implement the above WordCount algorithm by invoking their respective map and reduce operations. In the WordCount program running on Spark, output data in Spark is not arranged by alphabetical order. For our experiment, we developed two versions

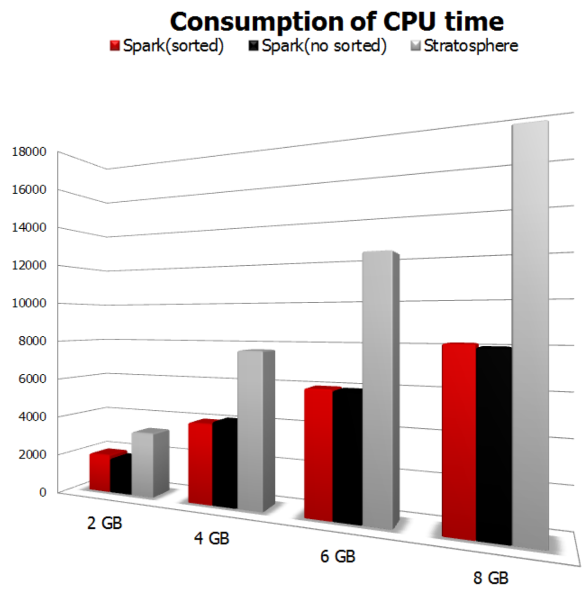


Figure 4.1. Comparison between Spark and Stratosphere in consumption of CPU time. X-coordinate represents input size(unit is GB) and y-coordinate shows used CPU time(unit is second)

of Spark WordCount programs. One Spark WordCount program (Spark (sorted)), including additional alphabetic order mechanism to achieve the identical output as Stratosphere. Another Spark WordCount program (Spark (no sorted)), is to complete the WordCount algorithm as fast as possible (no additional alphabetic order mechanism).

4.3.4 Results and Discussion

	2G input CPU time(s)	4G input CPU time(s)	6G input CPU time(s)	8G input CPU time(s)
Spark(sorted)	1941 s	3997 s	5970 s	8110 s
Spark(no sorted)	1786 s	4134 s	5919 s	7971 s
Stratosphere	3280 s	7650 s	12149 s	17008 s

Table 4.1. Consumption of CPU time running the Spark and Stratosphere Word-Count programs

Table 4.1 reflects the correspondence between the size of input and CPU usage and Figure 4.1 is a graphic demonstration based on Table 4.1. Both Spark and Stratosphere WordCount programs scale the linear growth in CPU usage. The two

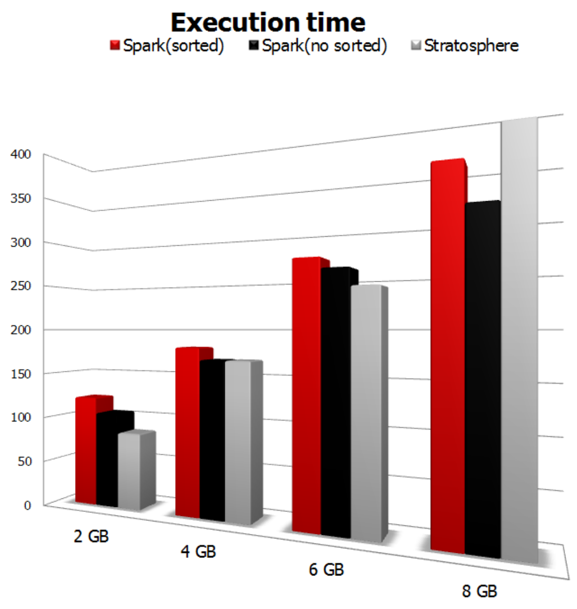


Figure 4.2. Comparison between Spark and Stratosphere in execution time. X-coordinate represents input size(unit is GB) and y-coordinate shows execution time(unit is second)

versions of Spark WordCount programs have the similar increasing degree, but the Stratosphere WordCount program consumes roughly 2 times CPU usage more than the Spark WordCount program for the same size of input.

	2G	4G	6G	8G
Spark(sorted)	122 s	181 s	272 s	354 s
Spark(no sorted)	105 s	168 s	261 s	314 s
Stratosphere	84 s	168 s	243 s	383 s

Table 4.2. Execution time running Spark and Stratosphere WordCount programs

As observed in Table 4.2, to begin with 2 GB as the amount of inputs, the Stratosphere WordCount program is superior to any of Spark WordCount programs in the aspect of consumption of execution time. As the size of input data rises, the Stratosphere WordCount program bumps execution time consumption over these two Spark WordCount programs until 8 GB data text as the input. According to Figure 4.2 that is a graphic demonstration of Table 4.2, both Spark and Stratosphere WordCount programs cannot meet the linear growth. This may result from various processing time at various slave nodes. As Figure 4.3 4.4 show, cloud1 and cloud7 accomplished their allocated tasks around 50 seconds slower than the rest of slave

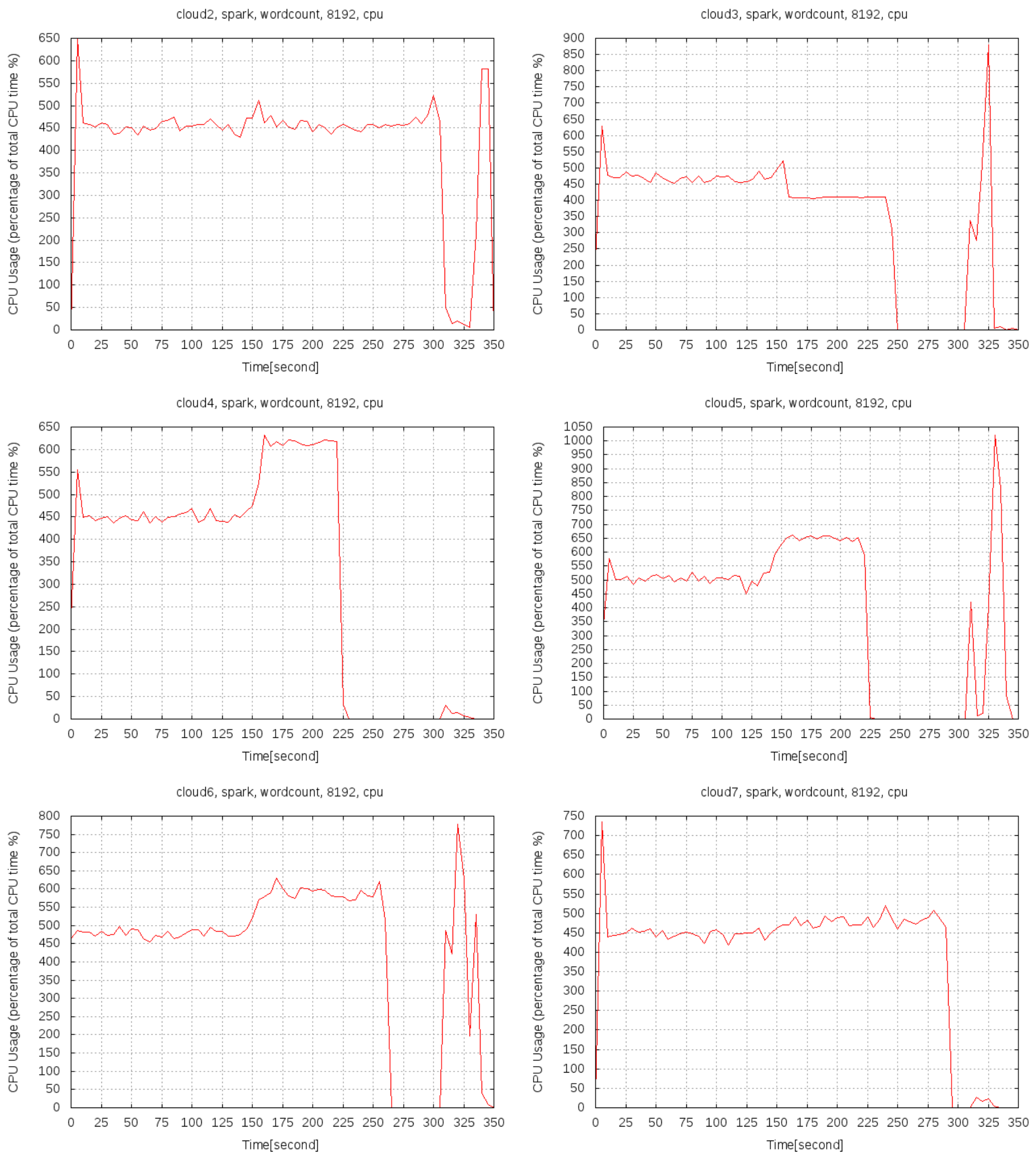


Figure 4.3. Consumption of CPU time on each slave running the Spark WordCount program excluding the master

nodes.

Although Stratosphere exceeds Spark two times in CPU consumption, their WordCount programs end up within the approximate execution time. On the analogy of this, if Spark and Stratosphere take advantage of the same CPU resources, the Spark WordCount would take 2 times less execution time than the Stratosphere WordCount. Therefore, it is conclusion that Spark is a more suitable framework for handling WordCount-like problems than Stratosphere.

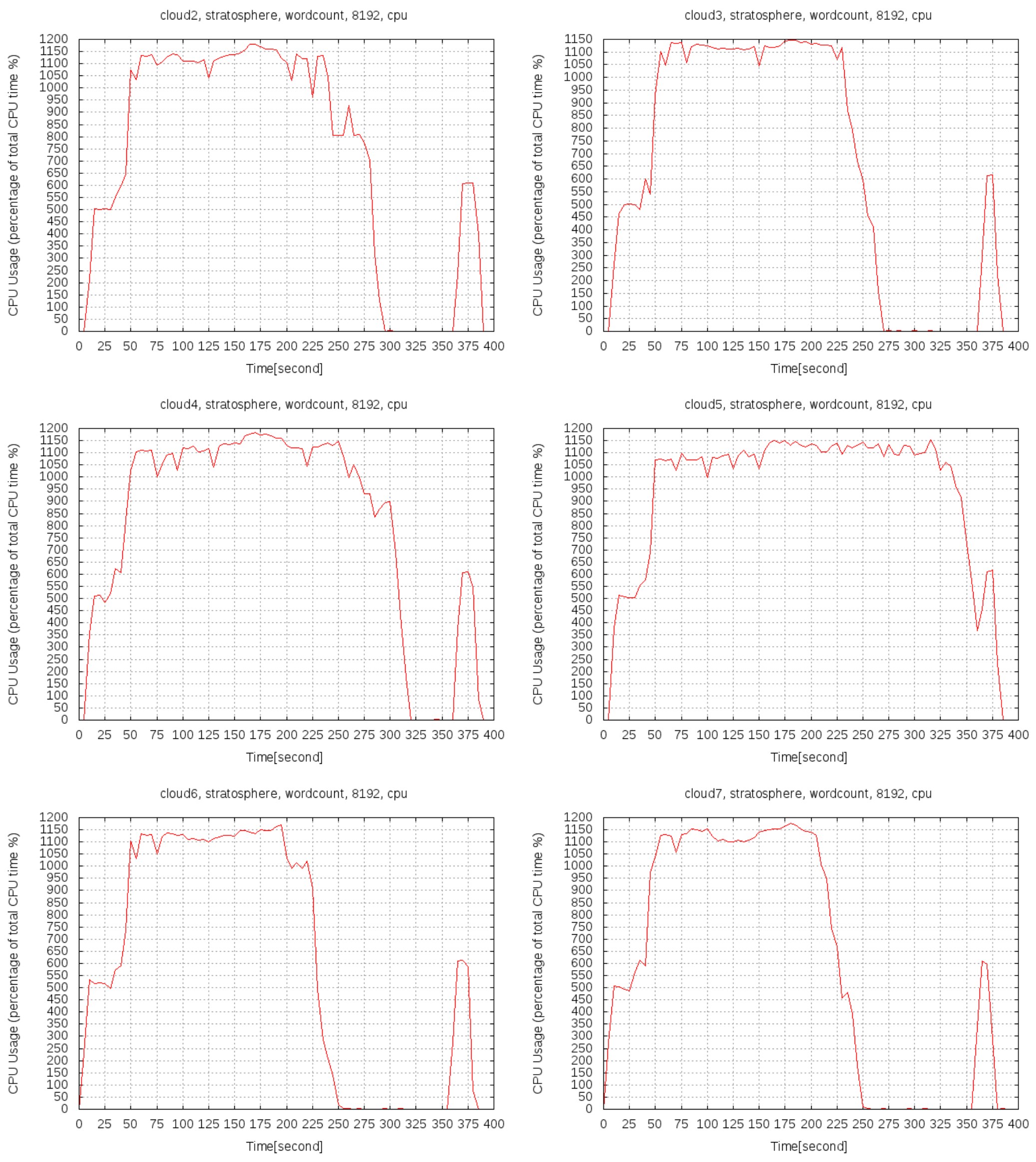


Figure 4.4. Consumption of CPU time on each slave running the Stratosphere WordCount program excluding the master

4.4 Iterative algorithm example

4.4.1 K-Means

We choose K-Means as the iterative algorithm example. K-means is one of clustering analysis methods that aim to distribute a group of data samples to k clusters where data objects have the most similarity each other. In this experiment, the similarity between data objects is evaluated by their Euclidean distance. In our experiment of K-Means, we narrow down the complexity of K-Means, in the premise of specifying a fixed number of clusters (k=3) and generating a specified size of input data.

4.4.2 Data Source

The data points are generated by a data generator[23] for K-Means provided by Stratosphere. We generated four different size of input data are generated ranging from 10,000,000 data points to 40,000,000 data points and are equally scaled at 10,000,000 data points.

4.4.3 Algorithm Description

The K-Means algorithm is described in the way of Stratosphere K-Means program[24]. This is one iterative implementation. We also developed an outer driver program to control the loop control for Stratosphere.

1. **Data source:** The program has two data sources: Data points and cluster centers. The program is shifted to associated data sources.
2. **Cross phase:** The CROSS operation computes distances between each data point and cluster center. User Function implements how to compute distances. Each output is formatted as the tuple (the ID of data point, (data point, cluster center, computed distance)).
3. **Reduce phase:** The REDUCE operation finds nearest cluster center for each data point. It partially groups the same key of records and then finds the minimal distance in this group by UF. The key of outputs is the ID of the closest cluster center and the value of outputs is data point.
4. **Reduce phase:** Another REDUCE operation re-computes the position of the cluster centers. It firstly groups the same key of records and then computes the average of these records in a group. The key of outputs is the ID of the closest cluster center and the value of outputs is new cluster center.
5. **Data sink:** DataSink operation writes records back to HDFS.

The Stratosphere outer driver program mainly includes a for loop statement where the exit condition is the specified number of iterations and the loop body is the above one iterative implementation.

The Spark K-Means implementation has two different points from Stratosphere. One is that Spark K-Means does not require an outer driver program to control the loop. Another one is that Spark allows developers to cache repeated data sets in the memory rather than repeatedly load data sets through HDFS.

4.4.4 Methodology

As K-Means is an iterative algorithm, both Spark and Stratosphere K-Means programs are executed as three various iterative approaches: 1 iteration, 2 iterations, and 3 iterations. In this experiment, two things are attractive to compare. One is the relation between the size of input and CPU usage. Another is the relation between the number of iterations and CPU usage.

4.4.5 Results and Discussion

Iteration	Framework	10 m	20 m	30 m	40 m
1	Spark	102 s	179 s	228 s	282 s
	Stratosphere	796 s	1636 s	2289 s	3327 s
2	Spark	155 s	285 s	447 s	707 s
	Stratosphere	1531 s	3014 s	4628 s	6704 s
3	Spark	164 s	312 s	563 s	1081 s
	Stratosphere	2335 s	4654 s	7092 s	9655 s

Table 4.3. Consumption of CPU time running Spark and Stratosphere K-Means programs

The Table4.3 reflects the statistics among the size of input, CPU usage and the number of iterations. Firstly, in Figure 4.5 4.6 4.7, the size of input and CPU usage correspond to the linear growth. In these three iterative instances, the Stratosphere K-Means program consumes roughly 10 times more CPU resources than the Spark K-Means program. Secondly, for the same size of input, the number of iterations and CPU usage also conform to the linear growth. But the Stratosphere K-Means program has much bigger increasing degree than the Spark K-Means program.

Table4.4 reflects the relation among the size of input, CPU usage and the number of iterations. We also view these statistics from two perspectives. Firstly, for any same iteration in Figure 4.8 4.9 4.10, the Spark K-Means program cannot be drawn as a linear graph, whereas the Stratosphere program roughly conforms to a linear graph. Secondly, for the same size of input, both Spark and Stratosphere K-Means programs conform to a linear graph. When the size of input data is entered ranging from 10,000,000 data points to 30,000,000 data points, the Stratosphere program clearly surpasses the Spark program in the respect of consuming execution time. When the size of input data reaches 40,000,000 data points, however, the growth tendency of the Spark program turns to be abnormal that the Spark program

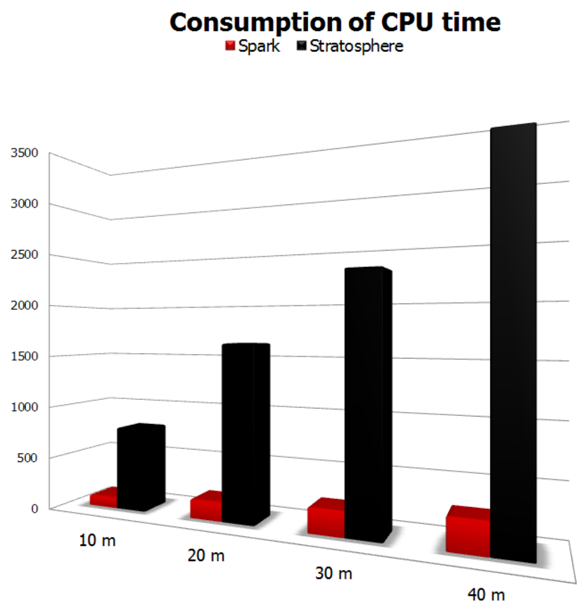


Figure 4.5. Comparison between Spark and Stratosphere in consumption of CPU time for 1 iteration. X-coordinate represents input size(unit is million data points) and y-coordinate shows used CPU time(unit is second)

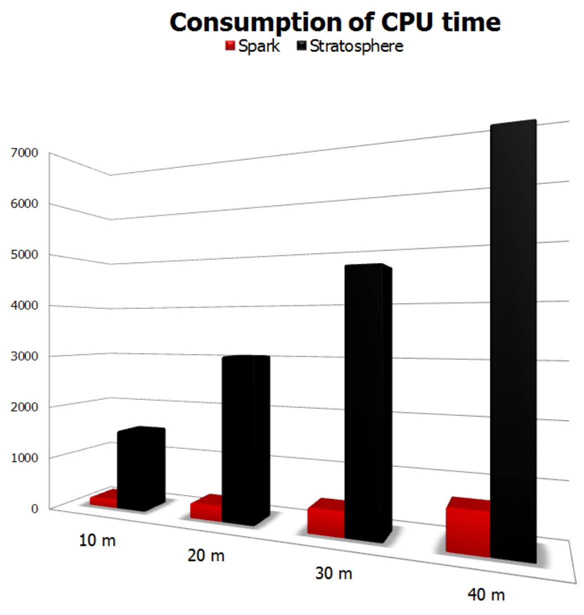


Figure 4.6. Comparison between Spark and Stratosphere in consumption of CPU time for 2 iterations. X-coordinate represents input size(unit is million data points) and y-coordinate shows used CPU time(unit is second)

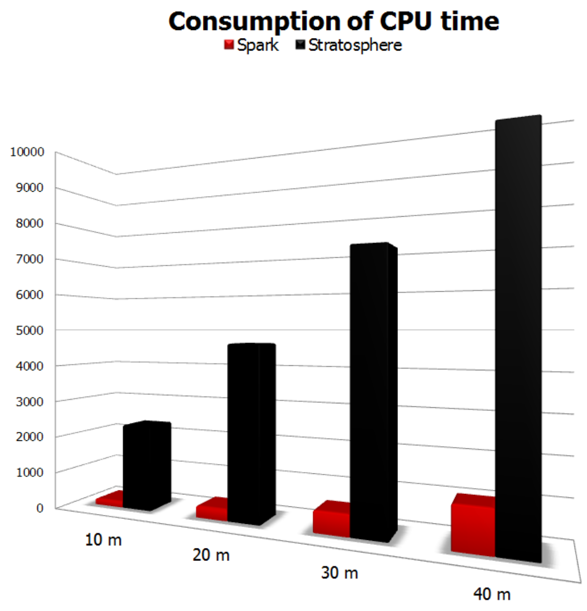


Figure 4.7. Comparison between Spark and Stratosphere in consumption of CPU time for 3 iterations. X-coordinate represents input size(unit is million data points) and y-coordinate shows used CPU time(unit is second)

Iteration	Framework	10 m	20 m	30 m	40 m
1	Spark	28 s	23 s	26 s	23 s
	Stratosphere	40 s	70 s	80 s	105 s
2	Spark	45 s	45 s	57 s	209 s
	Stratosphere	80 s	115 s	165 s	200 s
3	Spark	57 s	66 s	75 s	360 s
	Stratosphere	125 s	175 s	244 s	325 s

Table 4.4. Execution time running Spark and Stratosphere K-Means programs

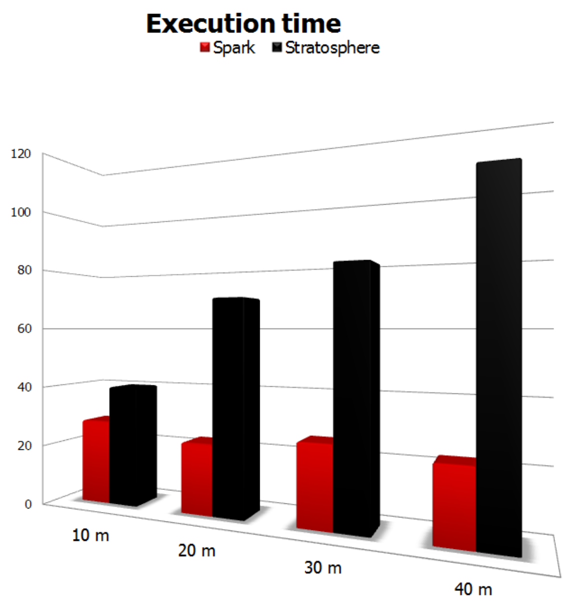


Figure 4.8. Comparison between Spark and Stratosphere in execution time for 1 iteration. X-coordinate represents input size(unit is million data points) and y-coordinate shows execution time(unit is second)

spends more execution time over Stratosphere. As considered that Spark has an additional caching procedure over Stratosphere, we hypothesize that Spark K-Means occupies too much memory for caching operations and little memory for computing operations. To verify the correctness of our hypothesis, we run the same program of K-Means of Spark with same size of input data, 40,000,000 data points, but remove the caching operation. If the K-Means program of Spark performs gracefully and has no quicker growth, we would prove the correctness of our hypothesis. Table 4.5 shows the data statistics of the Spark K-Means without caching operations. The execution time of Spark K-Means is gracefully increased as the number of iterations increases. Therefore, our assumption is corrected that the execution time would be dramatically increased if Spark occupies too much memory for caching operations.

	1 iteration	2 iterations	3 iterations
Spark(No cached)	24 s	38 s	44 s

Table 4.5. Execution time running the Spark K-Means program with no caching operation. The size of input is 8 GB.

Why does the Spark K-Means program suddenly slow down? As considered that Spark has an additional caching procedure over Stratosphere, we hypothesize

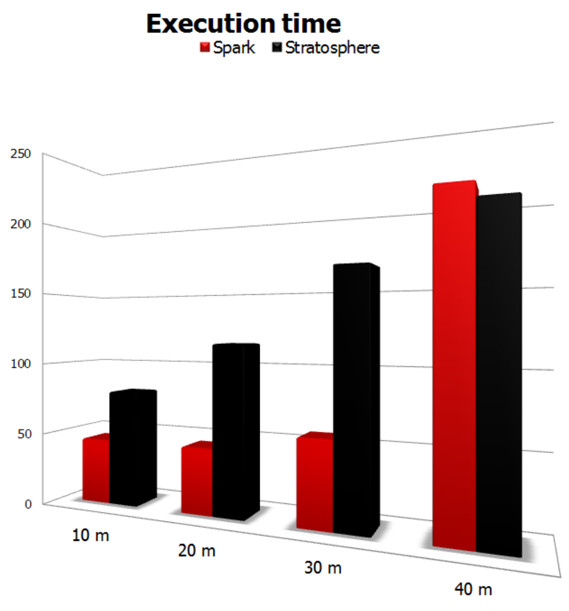


Figure 4.9. Comparison between Spark and Stratosphere in execution time for 2 iterations. X-coordinate represents input size(unit is million data points) and y-coordinate shows execution time(unit is second)

that the caching operations in Spark K-Means lead to the slowdown. To verify the correctness of our hypothesis, we run the same program of K-Means of Spark with same size of input data, 40,000,000 data points, but remove the caching operation. If the K-Means program of Spark performs gracefully and has no quicker growth, we would prove the correctness of our hypothesis. Table 4.5 shows the data statistics of the Spark K-Means without caching operations. The execution time of Spark K-Means is gracefully increased as the number of iterations increases. Therefore, the caching operations in Spark K-Means program leads to the slowdown.

Why does the caching operation in Spark K-Means program slow down the execution? We have two assumptions. Firstly, by observing the logs in the Spark K-Means program, Spark spends more time (10 more seconds) for searching cached blocks in the 40,000,000 data points' instance. We hypothesize that partial cached blocks may be kept in the distributed file system rather than in memory. Therefore, this may lead to the slowdown of the caching operations. Secondly, caching operations are used to occupy memory and the amount of memory for caching is dependent on the size of input data in this K-Means experiment. Larger amount of input data are loaded, more memory are asked to cache. In this experiment, 40,000,000 data points are equal to 753 MB and the memory is configured to 3 GB for Spark on each slave server. Although each slave server has available memory space to cache 40,000,000 data points, Spark has less memory for computational

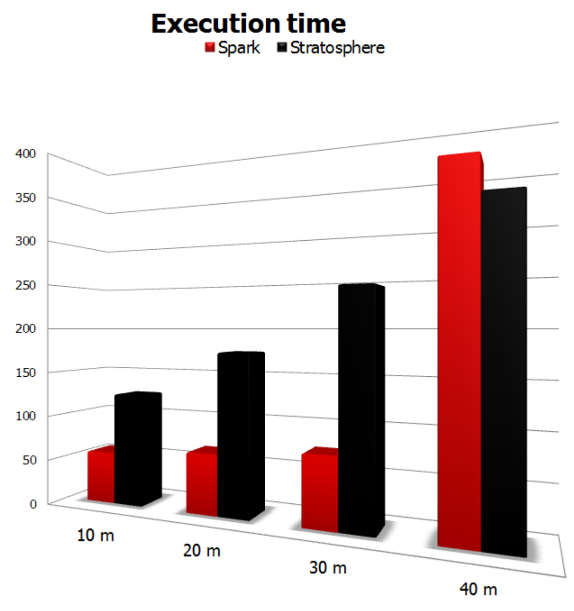


Figure 4.10. Comparison between Spark and Stratosphere in execution time for 3 iterations. X-coordinate represents input size(unit is million data points) and y-coordinate shows execution time(unit is second)

procedure in contrast of previous experiment with 10,000,000 data points. These two assumptions may be still further proved in the future experiments.

In summary, caching operations in Spark indeed are tempting by means of observing the results of Column 10 m and Column 20 m in Table 4.4 where consumption of execution time in 2 iterations is less than twice consumption of execution time in 1 iteration and so is the case of 3 iterations. However, for Column 30 m in Table 4.4, we gain no benefits from caching operations. Even when the size of input data is promoted to 40,000,000 data points, poor performance occurs when caching operations are triggered in the Spark K-Means program. Hence, it is concluded that there is a balance of memory allocation to computation and caching operations. But it is hardly to say how to allocate memory, since it depends on the actual algorithms and the amount of inputs.

4.5 Relational Query example

Query operations are widely applied in the field of analysis of big data. Therefore, we conduct a query operation on relational data.

```
SELECT l_orderkey, o_shippriority, sum(l_extendedprice)
FROM orders, lineitem
WHERE l_orderkey = o_orderkey
AND YEAR(o_orderdate) > 1990
```

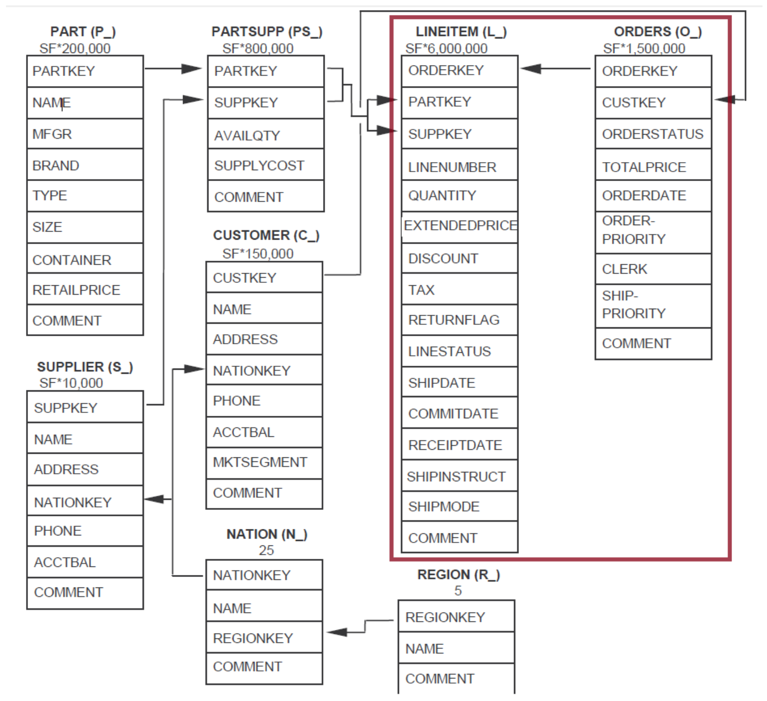


Figure 4.11. Overview of TPC-H database system (Marked tables are used in our experiment).

```
GROUP BY l_orderkey , o_shippriority
```

The above SQL statement is exemplified, which refers to only two tables, ORDERS and LINEITEM. This statement is also regards as a relational query example and implemented on top of Spark and Stratosphere.

4.5.1 Data Source

Tested data source is generated by the TPC-H Benchmark program[25], which is a decision support benchmark, including a set of data operations on data examination, data query, performance evaluation, and OLTP transactions. In the design and model of TPC-H database, it consists of eight tables depicted as the Chart 4.11. However, in our experiment, we are only interested in LINEITEM and ORDERS. We generated 2 GB, 4 GB, 6 GB and 8 GB input data that are stored as the form of LINEITEM and ORDERS.

4.5.2 Algorithm Description

The query algorithm is described in the way of Stratosphere query program.

1. **Data source:** The program has two data sources: ORDERS and LINEITEM.

The first attribute of these two tables is considered as the key of outputs. The value of outputs is the record itself.

2. **Map Phrase:** Each data sources go through MAP operation. The records from ORDERS are filtered in terms of "o_orderdata > 1990", and then attribute o_oderkey and o_shippriority in ORDERS are extracted. Attribute l_orderkey and l_extendedprice in LINEITEM are extracted.

3. **Match phrase:** MATCH operation performs join function in terms of "l_orderkey = o_orderkey". The new key is a combination of orderkey and shippriority. The value is the record itself.

4. **Reduce phrase:** REDUCE operation groups the records and sums up the extendedprice.

5. **Data sink:** The result is written back to HDFS.

The Spark query program has a similar algorithm as the Stratosphere query algorithm.

4.5.3 Results and Discussion

	2G input CPU time(s)	4G input CPU time(s)	6G input CPU time(s)	8G input CPU time(s)
Spark	473	807	1163	1523
Stratosphere	621	3285	5478	7163

Table 4.6. Consumption of CPU time running Spark and Stratosphere Query programs

The statistics about used CPU time and assorted input data are demonstrated in Table 4.6. The Stratosphere query program spends 5 times CPU resources than that the Spark one. In addition, both programs conform to the linear growth in Figure 4.12. Therefore, we would estimate that the query program running on Stratosphere needs more CPU resource than the one running on Spark in the case of larger amount of input data.

	2G	4G	6G	8G
Spark	30 s	39 s	42 s	66 s
Stratosphere	25 s	60 s	95 s	120 s

Table 4.7. Execution time running Spark and Stratosphere Query programs

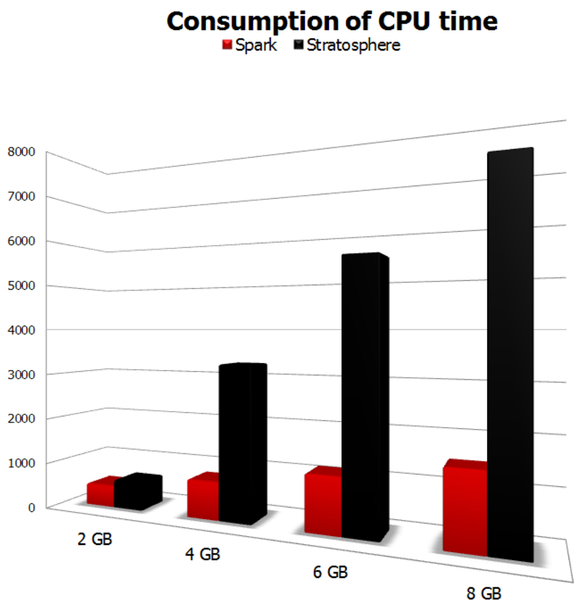


Figure 4.12. Comparison between Spark and Stratosphere in consumption of CPU time. X-coordinate represents input size(unit is GB) and y-coordinate shows used CPU time(unit is second)

According to the various amount of inputs, execution time of the Spark query program and the Stratosphere program are shown in Table 4.7. The Stratosphere query program has a linear growth in Figure 4.13, whereas the Spark query program does not satisfy a linear growth pattern. In the case of 2 GB input data, the Stratosphere query program is a little faster than the Spark one. After that, as the size of input is increasing, the Stratosphere query program consumes more execution time than the Spark one. When the size of input is 8 GB, the Stratosphere query program lags Spark twice execution time.

By comparing with the statistics of CPU usage and execution time, the Stratosphere program spends more CPU resources and execution time than Spark. Therefore, it is certain that we should choose Spark for handling query operation rather than Stratosphere.

At last, we discuss why Stratosphere programs consume much more used CPU time than Spark programs. We take the Spark and Stratosphere query programs with 8 GB inputs as an instance to explain. Figure 4.14 4.15 show variation trend of CPU usage on all the slaves. As map phase ends up (Spark at 25th second, Stratosphere unknown), Spark has obviously a falling trend, whereas Stratosphere has a flat trend and seems to keep CPU used all the time. Here, we have two assumptions. Firstly, Stratosphere uses a synchronous (push) communication model. Stratosphere does not wait for shipping and sorting until all Map tasks have been finished. In

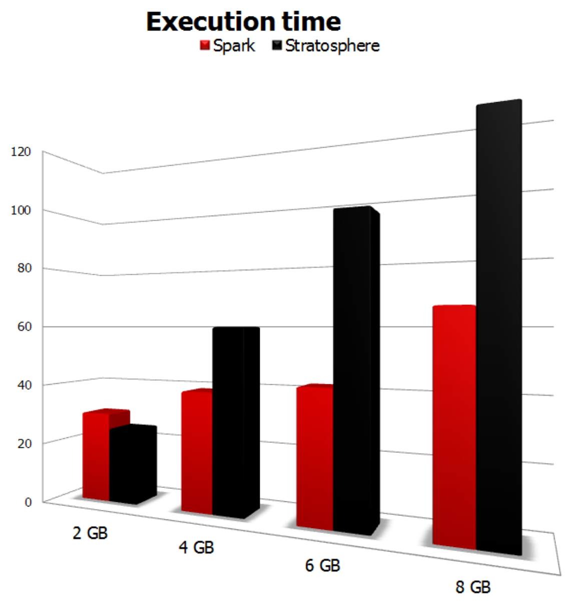


Figure 4.13. Comparison between Spark and Stratosphere in execution time. X-coordinate represents input size(unit is GB) and y-coordinate shows execution time(unit is second)

turn, Hadoop-like framework (e.g Spark) operates with asynchronous (pull) data shipping. All the map tasks must be done before shuffle operations start. Stratosphere synchronous communication model need consume additional communication overhead over Spark asynchronous communication model. Secondly, Stratosphere may invoke more threads for sorting than Spark.

In summary, Spark takes advantage of less execution time than Stratosphere in handling the query algorithm. And Stratosphere needs more CPU resources than Spark and even fully loads CPU resource. Therefore, as the size of input data are increased, Stratosphere may require more execution time than Spark.

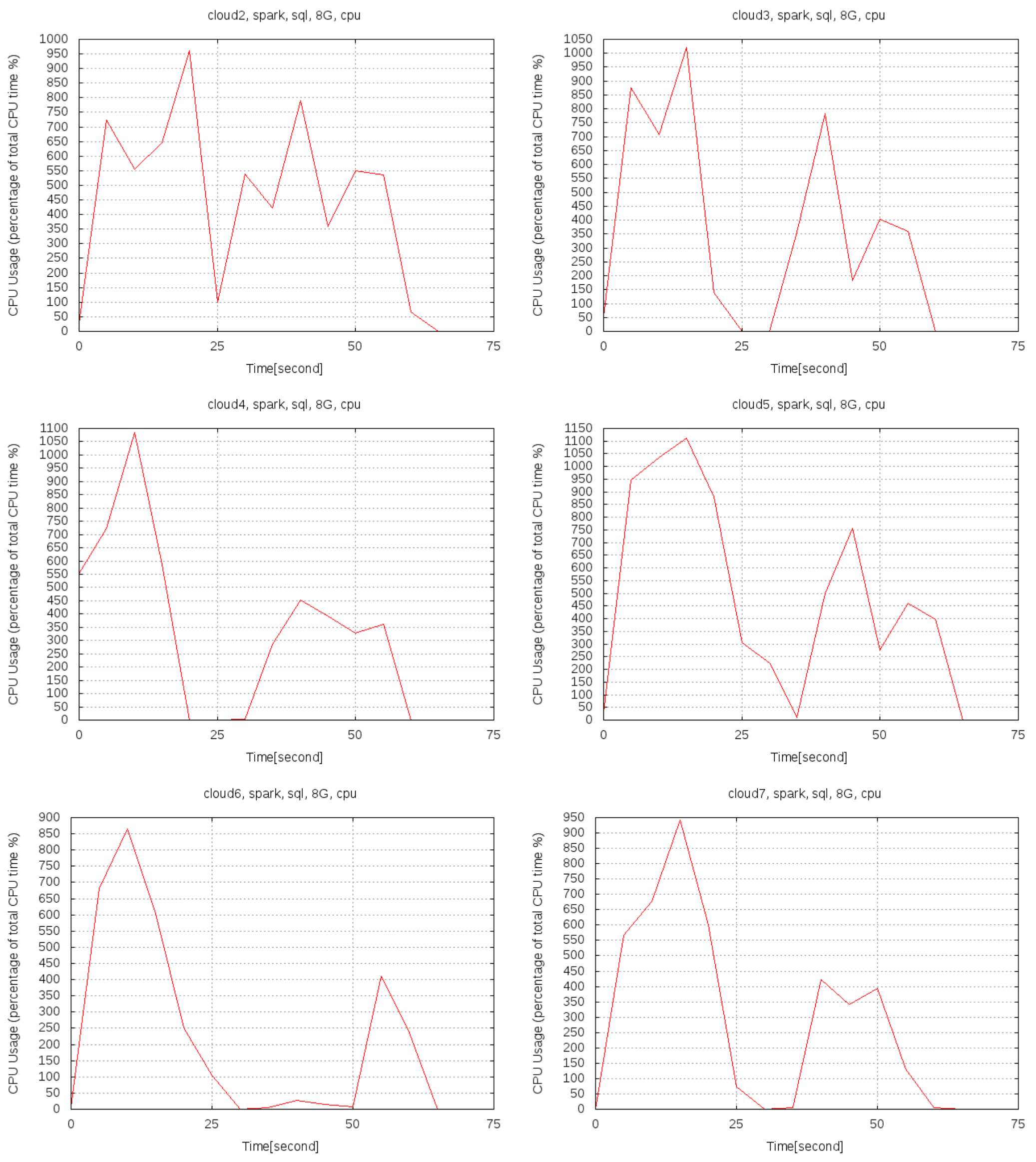


Figure 4.14. Consumption of CPU time on each slave from Query program in Spark excluding the master

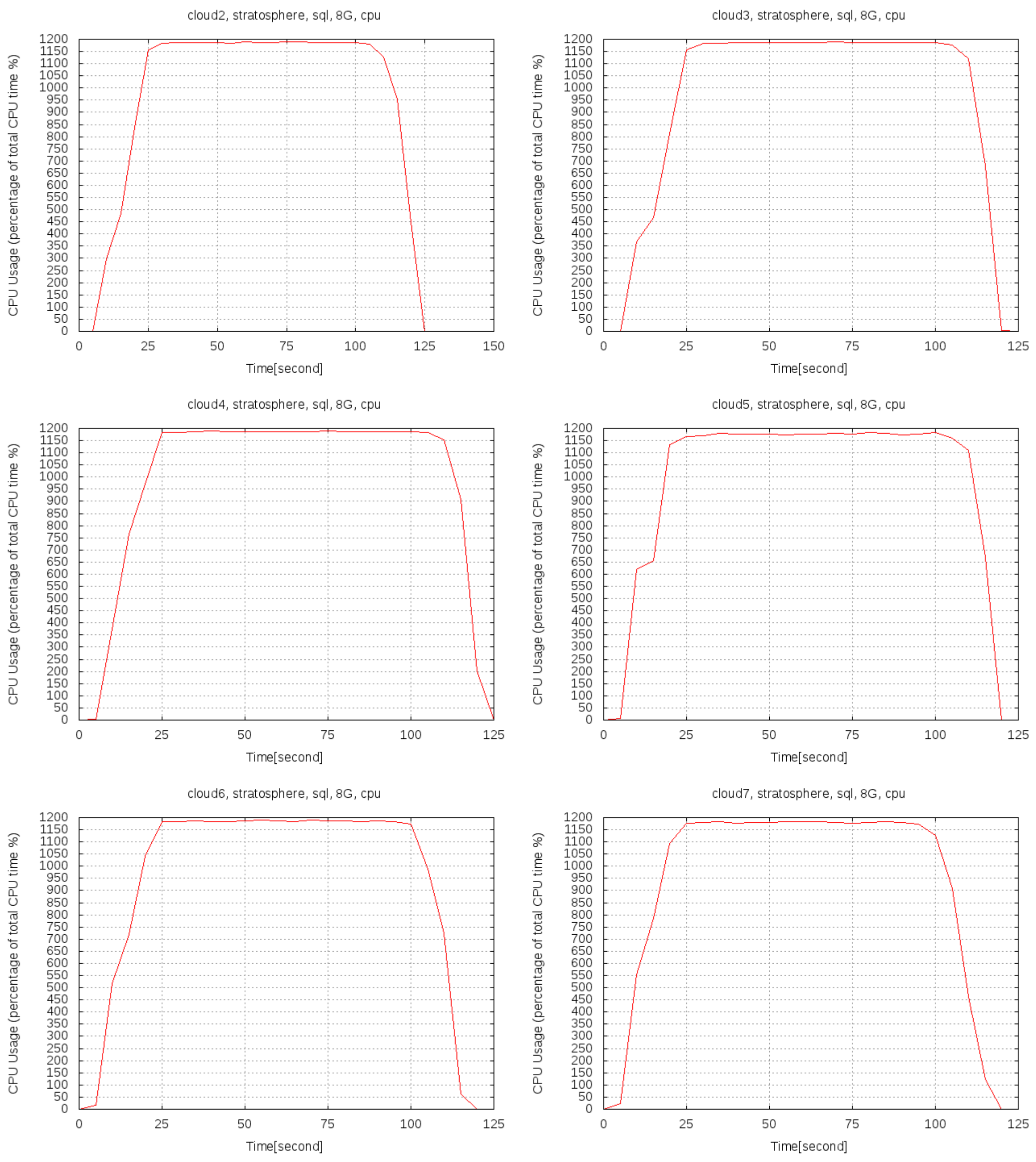


Figure 4.15. Consumption of CPU time on each slave from Query program in Stratosphere excluding the master

Chapter 5

Ongoing Work

When this thesis is being written, new features and additional theory programming models related to Spark and Stratosphere are designed and published. They aim to extend Spark and Stratosphere and ease them to develop a wide range of algorithms. For example, Shark enables Spark users to write simple SQL-like scripts and manipulate Spark system without having deep understanding of Scala. Likewise, for Stratosphere users, Stratosphere also designs similar SQL-like scripts to ease the development of Stratosphere programs by using Meteor and Sopremo. Besides, Stratosphere teams endeavor to extend existing programming model and propose new approaches to support iterative algorithms. In this chapter, we introduce Shark, Meteor and Sopremo, and Stratosphere spinning fast iterative data flow.

5.1 Shark

Shark[26] represents Hive[27] on Spark. As the name of Shark implies, Shark provides similar features as Apache Hive, like reporting, ad hoc analysis, ETL for machine learning, and so forth. More specifically, Shark supports Hive's query language, metastore, serialization formats, and user-defined functions. Since Shark is built on Spark, Shark can take advantage of the innovative features of Spark (e.g. the caching mechanism). Besides the caching mechanism, Shark includes a set of performance optimizations. These optimizations materialize in two aspects: hash-based shuffle rather than sort-based shuffle to speed up the performance of group-by operations; restriction of push-down in order-by operations, for example, "select * from LINEITEM order by ORDERKEY limit 20".

5.2 Meteor and Sopremo

Meteor and Sopremo[28] are applied on top of PACTs and Nephele. The Figure 5.1 shows the anatomy of the Stratosphere system that includes Meteor and Sopremo. Meteor and Sopremo can simplify the parallel programming by writing

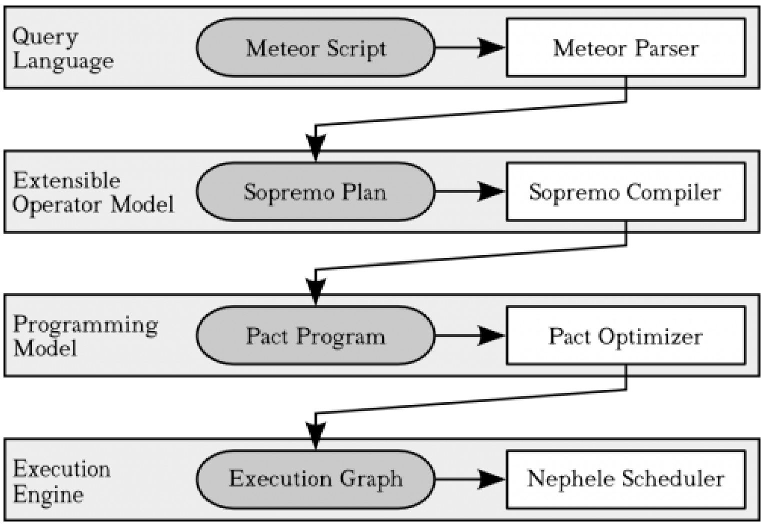


Figure 5.1. Architecture of Stratosphere integreted with Meteor and Sopremo[6]

SQL-like statements without being aware of underlying implementation of PACTs and Nephele.

Meteor is a high-level query language and reflects the similar features as Shark scripts. Sopremo is used to handle query scripts and generate an execution plan for underlying components (PACTs and Nephele).

5.3 Spinning Fast Iterative Data Flow

The novel spinning fast iterative data flow[29] is contrived in order to solve problems that PACTs programming model is incapable of efficiently handling iterative algorithms. The Stratosphere spinning fast iterative data flow is based on PACTs and provides support for bulk iterations and incremental iterations.

Bulk iterations: At the new iteration, the program loads the previous iteration's result and computes a complete new result (e.g. K-Means).

Incremental iterations: Based on the previous iteration's result, the program changes or adds some data at the new iteration (e.g. connected component algorithm[30]).

Currently, Stratosphere supports the new iterative model in theory, but the latest version of Stratosphere has not implemented the new iterative functionality. Unlike the Stratosphere, Spark has provided the support for bulk iterations but restricts the working sets in the memory as the read-only form. The Stratosphere extended iterative model allows more freedom than Spark for memory reads and writes, and supports both bulk iteration and incremental iterations. Hence, Stratosphere requires more concerns in the aspects of data consistency and fault tolerance than Spark.

Chapter 6

Conclusions

Both Spark and Stratosphere programming framework have attractive merits and it is hard to say which one is better in general. Therefore, I will illustrate several key criterion that may help users choose one from Spark and Stratosphere in terms of users' requirements. These key items include richly functional parallel methods, fault tolerance, optimization, execution time, expense of computing resource, support for iterative algorithms, deployment, and application development.

As mentioned in Section 3.3.3, both frameworks provide almost the same quantity of parallel methods and have identical functions (e.g. map, reduce, cross, match and cogroup). Therefore, as compared with the richly functional parallel methods, there is no decisive winner. Secondly, (at least, currently) Spark surpasses Stratosphere in the aspect of fault tolerance. In theory, Spark allows users to choose the tradeoff between lineage and checkpoint approach in terms of specified use cases, whereas Stratosphere only supports checkpoint. In the distributed version of Spark 0.9 and Stratosphere 0.1.2 used in the evaluation, Spark implements fault tolerance but Stratosphere does not. Thirdly, we consider the optimization mechanisms. Stratosphere specially provides output contracts as well as a set of compiler hints to optimize shipping strategies, but Spark has no similar optimization mechanisms.

Next, we draw conclusion about execution time and the expense of computing resources. The reason that I consider both items together is because execution time is associated with expense of computing resources. In the ideal experimental environment, by tapping more computing resources, applications can spend less execution time to carry them out. In practical environment, however, the computing resources are limited. When requested computing resources cannot be met, applications have to spend more execution time. In our experiments, when we enter the same amount of input, Stratosphere spends more computing resources than Spark, but Stratosphere can complete its job ahead of Spark. However, when the requested resources cannot be met, Stratosphere spends more execution time than Spark at a certain point. This case may frequently occur in the multi-user cluster environment.

The following criterion is related to whether both Spark and Stratosphere have support for iterative algorithms. The programming model of Spark is characterized by having designed for handling iterative algorithms, leaving data objects in memory. These data objects can be located and used on ongoing iterations. Therefore, it avoids applications reloading data from distributed file systems repeatedly. However, in our experiments, for iterative algorithms, the Spark programs may show the poor results in performance in the environment of limited memory resources. For example, applications with cached operations spend more execution time than applications without cached operations. As compared to Spark, Stratosphere does not support such feature.

For the deployment in the distributed environment of Spark and Stratosphere, Stratosphere exceeds Spark. Usually, we meet compatibility issues when we deploy Spark and Mesos. The compatibility issues result from incompatible versions of Spark and Mesos, because Spark and Mesos are two independent projects and new changes in each one may potentially threaten each other. Stratosphere is an integrated system and therefore Stratosphere is never trapped in the same plight as Spark.

At last, application development based on both Spark and Stratosphere will be covered. Spark applications are implemented by writing Scala code, whereas Stratosphere applications are developed by using Java code. This distinction may also influence the decision of choosing Spark or Stratosphere. In the process of writing code, targeted the same algorithm, the number of code lines in Scala is much less than in Java.

In summary, how to select from Spark and Stratosphere for ongoing projects is, in fact, depend on the requirement of actual projects. By making comprehensive analysis on projects and illustrating the key criterion, we may select the suitable one.

6.1 Open Issues

There is one problem in K-Means program based Spark framework that when the size of input data is 40,000,000 data points, in the instances of 2 iterations and 3 iterations, the execution time escalate dramatically. The reason is because there is no enough memory for computational operations. In our experimental setting, memory is a quite limited resource on each slave server (3 GB memory) and we spend most of memory on each slave server for caching inputs, when the size of input data is 40,000,000 data points. In turn, Spark has less memory for performing computation. Hence, the Spark K-Means program with 40,000,000 data points as inputs spends strange execution time. In my opinion, prior to run Spark programs, we could run pre-phase experiment with small proportion of input data and cluster resources like memory, CPU. If the tested results conform to the linear growth, we could safely perform caching operations. Otherwise, we may cache partial input data in memory, invoke file paging or do not use caching operations at all.

6.2 Future Work

6.2.1 Decision Support System

Following this thesis, one can develop a decision support system that is used to select the favorable parallel compute framework. In this system, one can predefine a set of interesting items. For example, fault tolerance, iteration support, optimization support, and so forth. Then, by comparing Spark and Stratosphere with these items, one can score each item (denoted as I) for Spark and Stratosphere. When customers try to select the favorable parallel compute framework, they need to decide the weighted value of each item (denoted as W). And then, the decision support system computes the final score of each parallel compute framework by summing up each item's value(I*w). The highest score one is regarded as the favorable framework.

6.2.2 The comparison of Spark programming model and new extended programming model of Stratosphere

As we have discussed in Section 5.3, the new extended programming model of Stratosphere proposes the spinning fast iterative data flow. Until now (Stratosphere version 0.1.2), the distribution of Stratosphere does not support the new iterative approach. But, it will be interesting to evaluate the Spark and Stratosphere iterative applications when Stratosphere appends the new iterative feature in the ongoing distribution.

6.2.3 Evaluation between Shark and Sopremo

The Spark and Stratosphere programs are developed at relatively low level, which means developers still need to spend plenty time and write complex codes. For this reason, Spark authors have proposed Shark and Stratosphere authors have proposed Sopremo. Based on Shark and Sopremo, developers can keep away from writing complex codes and instead write simple SQL-like statements. If the transition from Shark/Sopremo scripts to Spark/Stratosphere executable codes does not lead to the performance loss, it will be also attractive to evaluate the performance of Shark and Sopremo programs.

Appendix A

WordCount

A.1 Spark WordCount Source Code

```
package main.scala
import spark.SparkContext
import SparkContext._

object WordCount {

  def main(args: Array[String]) = {
    if (args.length < 5) {
      System.err.println("Usage: SparkWordCount <master> <Spark_home> <input> <output> <noReducer> <isSort> <Jars>")
      System.exit(1)
    }
    val noReducer = args(4)
    System.setProperty("spark.default.parallelism", noReducer)
    val spark = new SparkContext(args(0), "SparkWordCount", args(1), List(args(6)))
    val file = spark.textFile(args(2))

    val isSort = args(5)
    if (isSort.equals("true")) {
      val counts = file.flatMap(line => line.replaceAll("\\W", " ").toLowerCase().split(" "))
        .filter(word => word.compareTo("") != 0)
        .map(word => (word, 1)).reduceByKey(_ + _).sortByKey(true)
      counts.saveAsTextFile(args(3))
    } else {
      val counts = file.flatMap(line => line.replaceAll("\\W", " ").toLowerCase().split(" "))
        .filter(word => word.compareTo("") != 0)
        .map(word => (word, 1)).reduceByKey(_ + _)
      counts.saveAsTextFile(args(3))
    }

    spark.stop()
  }
}
```

A.2 Stratosphere WordCount Source Code

Original Stratosphere WordCount source code can be found here . To gain the same result, we modify line 73, 74 in the following codes:

```
String line = value.toString();
```

```
line = line.replaceAll("\\W", " ");
line = line.toLowerCase();
```

Appendix B

K-Means

B.1 Spark K-Means Source Code

```
package main.scala
import spark.SparkContext
import spark.SparkContext._
import java.text.DecimalFormat
import java.text.DecimalFormatSymbols
import scala.collection.mutable.HashMap

object SparkKMeans {
  def parseVector(line: String): Vector = {
    val index = line.indexOf('|')+1
    val subline = line.substring(index)
    return new Vector(subline.split('|').map(_.toDouble))
  }

  def closestCenter(p: Vector, centers: HashMap[Int, Vector]): Int = {
    var index = 0
    var bestIndex = 0
    var closest = Double.PositiveInfinity

    for (i <- 1 to centers.size) {
      val vCurr = centers.get(i).get
      val tempDist = p.squaredDist(vCurr)
      if (tempDist < closest) {
        closest = tempDist
        bestIndex = i
      }
    }

    return bestIndex
  }
   def aggre_func(pair1:(Vector,Int),pair2:(Vector,Int)):(Vector,Int)={
    return (pair1._1 + pair2._1, pair1._2 + pair2._2)
  }
  def main(args: Array[String]) {
    if (args.length < 8) {
      System.err.println("Usage: SparkKMeans <master> <Spark_home> <file> <centerfile> <output> <iters> <noReducer> <
      System.exit(1)
    }
    val noReducer = args(6)
    System.setProperty("spark.default.parallelism", noReducer)
    val sc = new SparkContext(args(0), "SparkKMeans", args(1), List(args(7)))
    val lines = sc.textFile(args(2))
    val data = lines.map(parseVector).cache()
    var points = sc.textFile(args(3)).toArray().map(parseVector)
    var kPoints = new HashMap[Int, Vector]
    val iterations = args(5).toInt
    val output = args(4)

    for (i <- 1 to points.size) {
      kPoints.put(i, points(i-1))
    }
    for (i <- 1 to iterations) {
      var  closest = data.map ( p => (closestCenter(p, kPoints), (p, 1)) )

      var pointStats = closest.reduceByKey(aggre_func(_,_))

      var newPoints = pointStats.map {pair => (pair._1, pair._2._1 / pair._2._2)}.collect()

      for (newP <- newPoints) {
        kPoints.put(newP._1, newP._2)
      }
    }
    val result =  sc.parallelize(kPoints.toSeq,1)
    result.saveAsTextFile(output)
    sc.stop()
  }
}
```

B.2 Stratosphere K-Means Source Code

Original Stratosphere WordCount source code can be found here

Appendix C

TPC-H Query

C.1 Spark Query Source Code

```
package main.scala
import spark.SparkContext
import SparkContext._

object SQL {

  def main(args: Array[String]) = {
        if (args.length < 7) {
          System.err.println("Usage: SparkWordCount <master> <Spark_home> <ordersPath> <lineitemsPath> <output> <noRe
          System.exit(1)
        }
    val spark = new SparkContext(args(0), "SparkSQL" , args(1) , List(args(6)) )
    val noReducer = Integer.parseInt(args(5))

    val orders = spark.textFile(args(2))
    val filter = orders.filter(filter_func).map(extract_func)

    val lineitem = spark.textFile(args(3))
    val project = lineitem.map(project_func)

    val join = filter.join(project, noReducer).sortByKey(true)

    val result = join.reduceByKey(aggre_func(_,_), noReducer)
    result.saveAsTextFile(args(4))
    spark.stop()
  }

  def filter_func(line:String):Boolean={
    val YEAR_FILTER:Int=1990
val items=line.split('|')
if(Integer.parseInt(items(4).substring(0,4))>YEAR_FILTER
    )
return true
else
return false
  }

  def extract_func(line:String):(Integer, String)={
    val items = line.split('|')
    return (items(0).toInt, items(7))
  }
  def project_func(line:String):(Integer, Long)={
    val items = line.split('|')
    return (items(0).toInt, (items(5).toDouble.toLong))
  }

  def aggre_func(pair1:(String,Long),pair2:(String,Long)):(String,Long)={
    return (pair1._1,pair1._2+pair2._2)
  }
}
```

C.2 Stratosphere Query Source Code

Original Stratosphere WordCount source code can be found here . To gain the same result, line 76 is modified in the following codes:

```
this.prioFilter = parameters.getString("PRIO_FILTER", "5");
```

And line 114-130 are modified as follows:

```
orderDate = record.getField(3, PactString.class);
if (!(Integer.parseInt(orderDate.getValue().substring(0, 4)) > this.yearFilter))
return;
record.setNull(2);
record.setNull(3);
record.setNull(4);

out.collect(record);
```

Appendix D

Benchmark Scripts

This set of benchmark scripts can be found here .

Bibliography

[1] Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/. [Online; Last accessed 25-December-2012].

[2] Computer cluster. http://en.wikipedia.org/wiki/File:Nec-cluster.jpg. [Online; Last accessed 24-September-2012].

[3] Quotation of ec2 in the region of eu. http://aws.amazon.com/ec2/pricing/. [Online; Last accessed 24-September-2012].

[4] Hdfs architecture. http://hadoop.apache.org/docs/r0.20.2/hdfs_design.html. [Online; Last accessed 24-September-2012].

[5] Typical parallel methods. http://stratosphere.eu/wiki/doku.php/wiki:pactpm. [Online; Last accessed 24-September-2012].

[6] Stratosphere integreates with meteor and sopremo. https://stratosphere.eu/wiki/doku.php/wiki:systemarchitecture. [Online; Last accessed 24-September-2012].

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[8] Stratosphere. https://stratosphere.eu/. [Online; Last accessed 23-September-2012].

[9] Amazon ec2. http://aws.amazon.com/ec2/. [Online; Last accessed 23-September-2012].

[10] Hdfs architecture guide. http://hadoop.apache.org/docs/r1.0.3/hdfs_design.html. [Online; Last accessed 23-September-2012].

[11] Apache hadoop. http://hadoop.apache.org/. [Online; Last accessed 23-September-2012].

[12] Openmp. http://openmp.org/wp/. [Online; Last accessed 23-September-2012].

[13] Open mpi. http://www.open-mpi.org/. [Online; Last accessed 23-September-2012].

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[16] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[17] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively parallel data analysis with PACTs on Nephele. *Proceedings of the VLDB Endowment*, 3:1625–1628, September 2010.

[18] Pacts wiki. `http://stratosphere.eu/wiki/doku.php/wiki:pactpm`. [Online; Last accessed 23-September-2012].

[19] mesos. `http://incubator.apache.org/mesos/`. [Online; Last accessed 23-September-2012].

[20] Daniel Warneke and Odej Kao. Nephele: Efficient parallel data processing in the cloud. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[21] Zookeeper wiki. `https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index`. [Online; Last accessed 23-September-2012].

[22] Mpi. `http://www.mcs.anl.gov/research/projects/mpi/`. [Online; Last accessed 23-September-2012].

[23] Stratosphere k-means data source generator. `https://github.com/stratosphere-eu/stratosphere/blob/master/pact/pact-examples/src/main/java/eu/stratosphere/pact/example/datamining/generator/KMeansGenerator.java`. [Online; Last accessed 16-November-2012].

[24] Stratosphere k-means algorithm. `http://stratosphere.eu/wiki/doku.php/wiki:kmeansexample`. [Online; Last accessed 16-November-2012].

[25] The tpc benchmark h (tpc-h). `http://www.tpc.org/tpch/`. [Online; Last accessed 16-November-2012].

[26] Hive on spark. `http://shark.cs.berkeley.edu/`. [Online; Last accessed 23-September-2012].

[27] Apache hive wiki. `http://shark.cs.berkeley.edu/`. [Online; Last accessed 23-September-2012].

[28] Marcus Leich Ulf Leser Arvid Heise, Astrid RheinlÃ¤nder and Felix Naumann. Meteor/sopremo: An extensible query language and operator model. In *Int. Workshop on End-to-end Management of Big Data*, 2012.

[29] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, July 2012.

[30] Connected component algorithm. `http://en.wikipedia.org/wiki/Connected_component_(graph_theory)`. [Online; Last accessed 12-November-2012].

TRITA-ICT-EX-2013:21