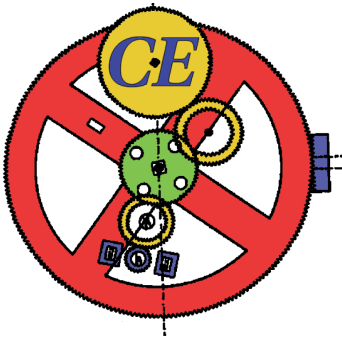




MSc THESIS

Evaluation of different storage systems for Apache Spark and Apache Hadoop

Thomas Samuel Hubregtsen

Abstract

CE-MS-2015-01

Big Data systems have been used for multiple years to solve problems that require scale. A framework takes care of scalability and resiliency issues, and allows the user to focus on relevant computation, in the form of map and reduce functions. In these Big Data systems, we currently see a shift from the traditional use of the Hard Disk Drive (HDD) towards in-memory computation. In this thesis, we have theoretically evaluated these two generations of Big Data systems, as well as two implementations, being Apache Hadoop and Apache Spark, in combination with Flash technology. We have also evaluated the possible use of Flash technology in these Big Data systems, by performing two experiments. Our first experiment examined the performance of Apache Spark versus Apache Hadoop for a representative iterative algorithm, and the performance degradation of Apache Spark under memory constraints. We have found that, for the chosen algorithm, Apache Spark performs equal-or-better compared to Apache Hadoop when data has to be loaded from the HDD, such as is the case of the initialization phase of a program. For the iterative part of our program, we have seen an overall speedup of 30, and a speedup of 100 for the map and reduce phases. In our second experiment, we evaluated two ways of using Flash, in particular using the IBM FlashSystem 840 connected to a Power8, in Apache Spark. We have first evaluated Flash technology with a mounted file system, and used this setup to replace the HDD as default spill location. We have found that this was not valuable, as the possible performance improvements were negligible compared to the overhead generated by data aggregation and system calls. We then shifted our focus to CAPI connected Flash, and modified Apache Spark to spill intermediate data directly to the FlashSystem using key_value pairs. In our experiment, while limiting the memory to a fixed amount to force spilling, we were able to remove 70% of the overhead caused by spilling. This was mainly overhead of Operating System (OS) involvement. In our future work, we will address the overhead caused by data aggregation, as we can write smaller amounts of data, because we are writing in key_value pairs. We believe that, once this overhead is removed, Big Data systems can benefit from Flash technology, and especially CAPI Flash technology, as one can use a system with a limited amount of expensive DRAM and a large Flash backend, to solve larger problem sets while maintaining a performance equal to in-memory computation.

Evaluation of different storage systems for Apache Spark and Apache Hadoop Comparing HDD, DRAM and (CAPI) FLASH on a Power8

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Thomas Samuel Hubregtsen
born in Wandsworth, United Kingdom

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Evaluation of different storage systems for Apache Spark and Apache Hadoop

by Thomas Samuel Hubregtsen

Abstract

Big Data systems have been used for multiple years to solve problems that require scale. A framework takes care of scalability and resiliency issues, and allows the user to focus on relevant computation, in the form of map and reduce functions. In these Big Data systems, we currently see a shift from the traditional use of the HDD towards in-memory computation. In this thesis, we have theoretically evaluated these two generations of Big Data systems, as well as two implementations, being Apache Hadoop and Apache Spark, in combination with Flash technology. We have also evaluated the possible use of Flash technology in these Big Data systems, by performing two experiments. Our first experiment examined the performance of Apache Spark versus Apache Hadoop for a representative iterative algorithm, and the performance degradation of Apache Spark under memory constraints. We have found that, for the chosen algorithm, Apache Spark performs equal-or-better compared to Apache Hadoop when data has to be loaded from the HDD, such as is the case of the initialization phase of a program. For the iterative part of our program, we have seen an overall speedup of 30, and a speedup of 100 for the map and reduce phases. In our second experiment, we evaluated two ways of using Flash, in particular using the IBM FlashSystem 840 connected to a Power8, in Apache Spark. We have first evaluated Flash technology with a mounted file system, and used this setup to replace the HDD as default spill location. We have found that this was not valuable, as the possible performance improvements were negligible compared to the overhead generated by data aggregation and system calls. We then shifted our focus to CAPI connected Flash, and modified Apache Spark to spill intermediate data directly to the FlashSystem using key_value pairs. In our experiment, while limiting the memory to a fixed amount to force spilling, we were able to remove 70% of the overhead caused by spilling. This was mainly overhead of OS involvement. In our future work, we will address the overhead caused by data aggregation, as we can write smaller amounts of data, because we are writing in key_value pairs. We believe that, once this overhead is removed, Big Data systems can benefit from Flash technology, and especially CAPI Flash technology, as one can use a system with a limited amount of expensive DRAM and a large Flash backend, to solve larger problem sets while maintaining a performance equal to in-memory computation.

Laboratory : Computer Engineering
Codenummer : CE-MS-2015-01

Committee Members :

Chairperson and advisor: Prof. dr. K.L.M. Bertels, CE, TU Delft

Member and advisor: Assistant prof. dr. ir. Z. Al-Ars, CE, TU Delft

Member and advisor: Prof. dr. ir. H. P. Hofstee, IBM Research and TU Delft

Member: Assistant prof. dr. ir. M. Zuniga, CS, TU Delft

Life is a journey, and the journey itself is home.

~

I dedicated this thesis to all the people I met along the road. Some of big influence, some small. Some with positive influence, some negative. But it is the sum of the experiences I had with all of you, that made me the person I am today. For this, I am grateful.

Contents

List of Figures	ix
List of Tables	xi
List of Acronyms	xiii
Acknowledgements	xv
1 Introduction	1
1.1 Evolution of Big Data Systems	1
1.2 Contribution of this thesis	2
1.3 Outline of this thesis	4
2 Background: Big Data systems	5
2.1 First generation: MapReduce	5
2.1.1 Programming model	5
2.1.2 Scalability	6
2.1.3 Resiliency	6
2.1.4 Ease of use	6
2.1.5 Bottleneck	7
2.1.6 Implementations	7
2.1.7 Example	7
2.1.8 Limitations of the first generation	10
2.2 Second generation	11
2.2.1 Resiliency	11
2.2.2 Bottleneck	11
2.2.3 Scalability	11
2.3 Flash	12
2.4 Research questions	13
3 First generation: Apache Hadoop	15
3.1 Program implementation	15
3.1.1 Program execution	15
3.1.2 Resiliency	16
3.1.3 Scalability	17
3.1.4 Ease of use	17
3.2 Example	17
3.3 Limitations	19

4	Second generation: Apache Spark	21
4.1	Resilient Distributed Datasets	21
4.1.1	Execution	23
4.1.2	Resiliency through lineage	23
4.1.3	Memory management	23
4.2	Impact on example	24
4.3	Data storage in Apache Spark	26
4.3.1	Spilling in the storage region	29
4.3.2	Spilling in the shuffle region	29
4.3.3	Performance	29
4.4	Research questions	30
5	Flash	31
5.1	Ways to store data	31
5.1.1	Serialized in the file system	31
5.1.2	In a key_value store via the Operating System	31
5.1.3	In a key_value store that is user accessible	32
5.1.4	As object store in memory	32
5.2	Ways to use Flash	32
5.2.1	Supporting a file system	33
5.2.2	In a key_value store via the OS	34
5.2.3	In a key_value store in the user address space	34
5.2.4	Objects in memory	34
5.3	Flash in Apache Spark	34
5.3.1	Replace the HDD with Flash	35
5.3.2	Refactor the spill system to spill key_value pairs to Flash (secondary object store)	35
5.3.3	Refactor the data store to write key_value pairs to Flash (primary object store)	35
5.3.4	Reconfigure the DRAM to be used together with Flash (Flash as memory)	36
5.4	Research Questions	36
6	Experiments	37
6.1	Experiment 1: How does Apache Spark perform	37
6.1.1	Hardware	37
6.1.2	Software	37
6.1.3	Algorithm	38
6.1.4	Input set	38
6.1.5	Limitations	38
6.1.6	Experiment 1a: How does Apache Spark perform relative to Apache Hadoop?	39
6.1.7	Experiment 1b: How does Apache Spark perform when we constrain available memory	42
6.2	Experiment 2: Can Flash technology contribute to Apache Spark	46

6.2.1	Hardware	46
6.2.2	Experiment 2a: Can we improve Apache Spark by using Flash technology connected as file storage	46
6.2.3	Experiment 2b: Can we improve Apache Spark by using Flash technology connected as secondary object store	49
7	Conclusion	53
	Bibliography	56
A	Code created	57
A.1	xDegreesOfSeparation for Apache Hadoop	57
A.2	xDegreesOfSeparation for Apache Spark	67
A.3	Altered Apache Spark code	79
A.3.1	main/scala/org/apache/spark/storage/BlockManager.scala	79
A.3.2	main/scala/org/apache/spark/storage/capi/FlashBlockManager.scala	80
A.3.3	main/scala/org/apache/spark/storage/capi/FlashBlockObjectWriter.scala	81
A.3.4	main/scala/org/apache/spark/storage/capi/FlashMapIterator.scala	83
A.3.5	main/scala/org/apache/spark/util/collection/ExternalAppendOnlyMap.scala	87
A.4	Code for the FlashSystem and Apache Spark	92
A.5	Code for the writing speed test	103
B	Spilling code for reference	107
B.1	MemoryStore.scala	107
B.2	ExternalAppendOnlyMap.scala	120
B.3	ExternalSorter.scala	137
B.4	ShuffleMemoryManager.scala	156
C	Code for reference	161
C.1	Ubuntu 14.04 Little Endian on Power	161
C.2	Apache Hadoop on Ubuntu with Power	168
C.3	Apache Spark on Ubuntu with Power	171

List of Figures

2.1	Movie information stored in a graph. Actor as a vertex, collaboration as an edge.	9
2.2	Movie information using IDs stored in a graph, two files and a single file respectively.	9
2.3	General data flow	10
3.1	Hadoop execution overview.	16
3.2	Execution flow in Hadoop during the initialization part of the example.	18
3.3	Execution flow in Hadoop during the iterative part of the example.	19
4.1	Different modules in Spark. Source: https://spark.apache.org/	21
4.2	Dependencies in Spark. Source: https://databricks.com/blog/2014/03/20/apache-spark-a-delight-for-developers.html	22
4.3	Execution flow in Spark during the initialization part of the example.	25
4.4	Execution flow in Spark during the iterative part of the example.	26
4.5	RDDs and dependencies in our implementation of the example in Spark.	27
4.6	Different ways to connect Flash in Apache Spark.	28
5.1	Different ways to connect Flash in Power8	32
5.2	Left: FlashSystem connected as a file system. Right: FlashSystem connected using CAPI Flash. Source: IBM internal communications	33
5.3	Different ways to connect Flash in Apache Spark. Dashed line represents data resides in the user address space.	35
6.1	Relative performance of Apache Hadoop versus Apache Spark for both the initialization step and the iterative step, split out were possible in map, sort and reduce phase timings.	41
6.2	Timing of Apache Spark per iterative step.	42
6.3	Performance degradation under constrained memory in Spark.	45
6.4	Hard disk versus ram-disk for Spark as default storage device.	49

List of Tables

1.1	Ways in which we can use the Flash storage system in different generations of Big Data systems.	2
2.1	Approximated performance characteristics and costs of DRAM, Flash and HDD.	12
6.1	Timing in seconds together with the Speedup for both the initialization step and iterative step in Apache Hadoop and Apache Spark.	40
6.2	Timing in seconds for the iterative steps in Apache Spark.	40
6.3	Time in seconds for various memory sizes in Gigabytes. Measured with no RDDs, all RDDs and the only iterative RDD cached.	44
6.4	Timing in seconds during the iterative step for both HDD and ram-disk, together with the associated speedup.	48
6.5	Writing speeds in microseconds for DRAM, Flash and HDD for various block sizes.	50

List of Acronyms

DFS	Distributed File Systems
HDFS	Hadoop Distributed File Systems
DRAM	Dynamic Random-Access Memory
IOPS	Input/Output Operations per Second
HDD	Hard Disk Drive
SSD	Solid State Disk
CAPI	Coherent Accelerator Processor Interface
RDD	Resilient Distributed Dataset
I/O	Input/Output
OS	Operating System
SAS	Serial attached SCSI
PEX	Power Systems Exchange
VSL	Virtual Storage Layer
SSD	Solid State Disk
MCS	Memory Channel Storage

Acknowledgements

The research for this thesis has been conducted at IBM Research in Austin, Texas. I would first like to thank Prof. Bertels, for recommending me for this position, and my previous internship, as well as his support during these placements. I would also like to thank Prof. Hofstee for his incredible daily support and supervision, as well as for his warm welcome to the United States and his continuing support after my placement in Austin. Further, I would like to thank Assistant Prof. Al-Ars for his supervision. I would also like to thank the people at IBM Research - Austin, and in specific J. Rellermeyer, PhD, J. Rubio, PhD and F Gebara, PhD, for their help and support during my stay in Texas. Last but not least, I would like to thank my friends and family for their support.

Thomas Samuel Hubregtsen
Delft, The Netherlands
January 20, 2015

Introduction

1.1 Evolution of Big Data Systems

At the start of the millennium a lot of developers were trying to solve similar problems. These problems typically involved processing massive numbers of raw (unstructured) text files, crawled from the Internet. Initially, these computations were performed on scalable High Performance Computing systems, leveraging openMP and MPI. The majority of the code was spent on providing scalability and resiliency, as the actual computation could generally fit into three relatively simple phases: a map phase, a sort phase and a reduce phase, closely resembling the structure of the functional programming language Lisp. Spending time on scalability and resiliency for every program was inherently inefficient, and a framework was introduced. This framework took care of the scale and resilient behavior needed to solve the problems, and the programmer only had to write two functions (implementing the map and the reduce phases), together with a job specification. This framework was popularized by Google in 2004, and goes under the name of MapReduce [1]. MapReduce can be seen as a framework for a first generation Big Data system.

A first generation Big Data system can be characterized as being scalable, resilient and easy to use. The input data typically consisted of unstructured text. This input data would be loaded into a distributed file system, and processed offline (batch processing). The MapReduce framework would then launch multiple workers, with reported clusters of up to thousands of nodes consisting of commodity hardware. When the input data is completely unstructured, the entire dataset needs to be read. This could be done no faster than the storage medium would allow, meaning a bottleneck of Hard Disk Drive (HDD) bandwidth. As 10 HDDs with around 1 Gigabit bandwidth were two orders of magnitude faster than storage connected over a commodity network with 100 Megabit bandwidth, an organization where the HDDs were locally attached was favored. Currently we see a trend in which more data is getting annotated and indexed. Big data systems no longer have to load the entire dataset in to memory, but only the relevant information. Because less data is loaded, the overall system is faster, but the actual read is slower, as the relevant information is scattered over the entire dataset. This incurs a lot of HDD seeks, each inducing millisecond time penalties, leading to a shift in bottleneck from HDD bandwidth to HDD Input/Output (I/O). The need to attach storage locally also becomes less, as network speeds have caught up to HDD speeds.

Another trend we currently see is the desire to run iterative algorithms and perform interactive analysis. This type of usage requires the input data to be loaded once,

after which it is used during a mix of multiple map and multiple reduce steps, before the answer is either written back to HDD or displayed on the screen. A first-generation Big Data system would save all results between MapReduce cycles to HDD, only to load them again in the next cycle. This is inefficient, as it incurs a lot of overhead in the form of serialization and HDD I/O and possibly duplication. Second-generation Big Data systems avoid this, by keeping the data in Dynamic Random-Access Memory (DRAM), as DRAM has more bandwidth and Input/Output Operations per Second (IOPS) than HDD. This is referred to as in-memory computation. Although DRAM is fast, it is significantly more expensive per byte and thus is limited by the available capacity.

1.2 Contribution of this thesis

As we explained in Section 1.1, we see two bottlenecks in Big Data systems. The first generation is limited by the number of IOPS that a HDD can handle. A second-generation system will use DRAM for in-memory computation, but is now limited by the available capacity. Flash technology has three orders of magnitude more IOPS than HDD, and multiple orders of magnitude more capacity than DRAM. Our aim is to investigate the possible use of Flash technology in Big Data systems. From this, we can formulate our initial question, which we will later redefine into our two research questions.

Initial question: Can Flash technology contribute to Big Data systems?

We can see in Table 1.1, that there are three¹ ways in which we can use Flash technology in Big Data systems. We can use Flash as a storage system, and replace HDD in both first- and second-generation Big Data systems. We can also use Flash as a memory system, and replace or supplement DRAM.

	Storage	Memory
1st generation	v	x
2nd generation	v	v

Table 1.1: Ways in which we can use the Flash storage system in different generations of Big Data systems.

As we explained in Section 1.1, the key difference between a first- and a second-generation Big Data system is the in-memory computation of the latter. When there is sufficient memory available to be leveraged by the system, we expect second-generation Big Data systems to always perform better for iterative algorithms and interactive analyses [2]. When we constrain the use of memory in a second-generation system, we expect the performance to gracefully degrade [2] [3] to a performance similar or better [4] than the performance of a first-generation Big Data System. In order to verify this assumption, we first question how second-generation Big Data systems perform relative to first-generation Big Data systems with no restrictions. We then question

¹First generation systems do make use of memory in the sort phase, but this data can not be re-used and is not relevant in this perspective.

the performance under memory constraints. We can also see from 1.1 that Flash can be connected both as a storage and as memory. With this information, we can now define our two main research questions.

Research questions

1. How does a second-generation Big Data system perform?
 - (a) How does a second-generation Big Data system perform relative to a first-generation Big Data system?
 - (b) How does a second-generation Big Data system perform when we constrain available memory?
2. Can Flash technology contribute to second-generation Big Data systems?
 - (a) Can we improve second-generation Big Data systems by using Flash technology connected as a file storage
 - (b) Can we improve second-generation Big Data systems by using Flash technology connected as an object store?

Questioning the relative performance should also give us insight into the actual need for improvement, as we expect Flash to be faster than HDD, but slower than DRAM. We would also like to note that although our focus lays on second-generation systems, we do expect Flash technology will become beneficial for first-generation systems now that input data becomes more structured, as Flash handles up to three orders of magnitude more IOPS than HDD.

In order to find an answer to our two research questions, this thesis contributes the following:

1. Exploration and evaluation of possible ways in which Flash can contribute to second-generation Big Data systems.
2. Empirical evaluation of performance degradation in Apache Spark for an iterative algorithm
3. Implementation and evaluation of a Coherent Accelerator Processor Interface (CAPI) enabled IBM Flash storage system configured as a secondary object store.

In order to perform the before mentioned evaluations, this thesis also contributes the following:

4. Guide to install little-endian Ubuntu 14.04 on an IBM Power8 and an IBM Power7 server
5. A port of both Apache Hadoop and Apache Spark to little-endian Ubuntu 14.04 on an IBM Power8 server
6. An implementation of the six degrees of separation from Kevin Bacon problem in both Apache Hadoop and Apache Spark

1.3 Outline of this thesis

In order to better understand the reasoning about possible implementations, we cover on an abstract level the background of both first- and second-generations of Big Data systems in Chapter 2. We then look at a popular implementation of a first generation Big Data system: Apache Hadoop, in Chapter 3. This is followed by an increasingly popular implementation for a second-generation Big Data system in Chapter 4, being Apache Spark. An exploration of different storage systems is given in Chapter 5. Chapter 6 will explain the experimental setup used. Chapter 7 presents and evaluates the results of these experiments. Chapter 8 draws conclusions, and we end with listing the suggested future work.

Background: Big Data systems

In this chapter we will take an abstract look at the differences between a first-generation Big Data system and a second-generation Big Data system and how Flash technology can contribute. We do this by first introducing the MapReduce framework. We will look at the programming model, the characteristics and the bottleneck. We will then discuss the limitations of a first-generation Big Data system, by the means of an example. Once we understand the functioning and limitations of a first-generation Big Data system, we will introduce the second-generation of Big Data systems, and how it overcomes these limitations. At the end of the chapter we will know the bottlenecks for Big Data systems, and see how Flash Technology compares to Hard Disk Drive (HDD) and Dynamic Random-Access Memory (DRAM) for these points.

2.1 First generation: MapReduce

The goal of the MapReduce framework is to provide a library that supports relative simple map and reduce functions, while automatically taking care of parallelism: the same code should easily be able to scale to thousands of commodity machines. Because of this scale, failure is to be expected, and has to be accounted for by the library. This leads to the key characteristics of a first generation Big Data system:

- Ease of use
- Resilient
- Scalable

Let us first take a look at the MapReduce programming model.

2.1.1 Programming model

The input consists of a set of raw data files stored into a Distributed File Systems (DFS), in the format of a key (filename), and value (raw data). This input will be taken by the map function, written by the user, which will generate a set of intermediate key_value pairs, saved in the memory. The MapReduce library will then group all values belonging to intermediate key I , and pass these in an iterator to the reduce function. This iterator is a special type of list that enables the user to efficiently handle lists of intermediate values that do not fit in one piece into memory. The reduce function is also written by the user, and accepts an intermediate key I and the set of values belonging to this key, stored in the Iterator. It will then merge the values belonging to key I to a result containing a possibly smaller set of values, with a typical number of zero or one elements. These results can then be written back to the DFS [2].

The keys and values from the input of the map function can be from a different domain than the keys and values of the output of this map function. However; the domain of both the keys and the values in both the input and the output of the reduce function must match the domain of the keys and values used in the output of the map function:

```
map(k1, v1) → list(k2, v2)
reduce (k2, list(v2)) → list(v2)
```

Next to this, the user also has to provide execution information, such as file locations and number of parallel processes to start in a so-called MapReduce specification object. Key in this all is that the framework has to be easy to use. Following this line of reasoning, we rely on the framework to take care of the scalability and the resiliency. Each implementation of the framework has its own details, but we can make some generalizations, as listed in the next subsection.

2.1.2 Scalability

So far, we have explained how every logical record has its own map job, and every key, together with its values, has its own reduce job. This reduce job will not be able to start before all results of the relevant map jobs are available. Because of this, most map and reduce jobs are not able to execute in parallel. However; there is no such dependency between the set of map jobs, nor between the set of reduce jobs. Because of this, we can highly parallelize calculation during these phases.

In order to scale a cluster it used to be the case that disks were locally attached, but as network speeds have grown faster than HDD speeds, we no longer have to cluster data for locality in first-generation systems.

2.1.3 Resiliency

MapReduce jobs are typically run on clusters consisting of thousands of commodity machines. As the mean time to failure of a single machine is relatively low, it is likely that one or more machines in the cluster will fail during job execution. In order to handle this, the MapReduce framework guarantees that any job can be recomputed upon failure, given that the input data is still available. To ensure this, both input and output data is stored with a user-chosen number of duplicates. Typically the total number of copies is three: one on the target worker, one on a machine in the same rack, and one on a machine in a different rack. Upon machine failure, the framework will mark all jobs of this machine as failed. It will then reschedule the jobs for execution, with a preference to move the computation to a worker containing a duplicate of the input set.

2.1.4 Ease of use

We can see that all of this is set up for the ease of use of the user. Both the scalability and the resiliency are taken care of by the library, and the distributed storing is generally

taken care of by the DFS. The only part the user has to write is a map and reduce function, together with a job specification.

2.1.5 Bottleneck

First-generation systems started out operating on mostly unstructured text. To run a query on this set, the system had to read the entire dataset and examine every record. The read will go as fast as the HDDs will allow. Because of this, first-generation systems are bandwidth bound for unstructured input. Currently we see a trend in which data gets broadly classified. An example of this is a simple tag to all fields labeling the data as being from a certain sender, or generated at a certain time-stamp. This can be done at the moment of generation, or during pre-processing. This broad tagging results in semi-structured data. Big Data systems no longer have to load the entire dataset, but only subsets of the results that carry a particular tag. This results in an overall faster performance, but the read of the particular input set for a query (which is now a subset of the entire dataset), will be slower. This is the case because the relevant data is scattered along the entire input set, and since we are working with HDDs, every new block read incurs a penalty of milliseconds to move the disk reader. As we can no longer work on the maximal bandwidth of the HDDs, we now see a shift in bottleneck from bandwidth towards Input/Output Operations per Second (IOPS).

2.1.6 Implementations

Many implementations exist, and the choice of one is dependent on both the problem set and the environment on which the system will run. Most systems in use nowadays are meant for batch problem sets, and run on clusters consisting of thousands of commodity machines, such as the x86. Since it is widely adopted, the focus of this thesis for a first generation Big Data system will be the Apache Hadoop implementation. This is elaborated upon in Chapter 3. But first we are going to look at the limitations of the first generation by introducing an example that we will use throughout the thesis.

2.1.7 Example

During the rise of the first generation of Big Data systems, companies such as Google, Yahoo and Facebook used the MapReduce framework, and specifically the Apache Hadoop implementation, to solve problems for which the answers could be found within a single iteration, and the program was allowed to run for hours or days to find it. At the current age in which the user expects to be fed with new statistics upon undetermined events, for instance right after a goal during an important soccer match, we can no longer do with reading and writing from HDD for every query. Real-time information needs to be combined with historic trends, and an answer is expected on the spot. Another group of algorithms that is also performing poorly in first generation's implantations is that of iterative algorithms, such as the ones used in machine learning. These algorithms re-use their data multiple times, but need to access the DFS to retrieve and write their data during every iteration. This data does not only consist of new results, but also contains information that does not change during computation.

These two problems, interactive analysis and iterative algorithms, and their nature to frequently access the HDD, were key to the start of a new generation of Big Data systems, in which HDD access times are avoided as much as possible. To explain this, we will use the algorithm discussed below as a leading example throughout the thesis. We will introduce this with a personal anecdote, about something that happened on several occasions during the past year.

Personal anecdote:

During my previous internship in Hong Kong, I was exploring the city and stumbled upon a fellow Dutch person. We talked for a bit, and then connected through social media, in order to stay in touch. After looking at each other's profile, we noticed that we had one friend in common. It turned out that she used to be a neighbor of a good friend of mine, who had moved to Delft, and with whom I used to share a student house with. After this occurrence, I became interested to see how often this was the case, and indeed, with almost every Dutch person of my age that I met, we were able to create a path between the two of us, consisting usually of only a few people, that could connect us. But there was one occasion that stood out even more: at some point I found myself on one of the small islands located in the Indonesian archipelago. While I was walking down the street, talking to somebody I met the other day, I saw a boat coming in, landing on the beach. For some reason I've found myself looking at the passengers jumping of into the sand. To my amazement, I noticed a particular individual in this group: an old friend of mine! I could not help but think: it sure is a small world!

This principle can of course be translated in an algorithm, and is commonly known under the name of "the small world principle" [5], or "six degrees of separation" [6]. A popular experiment along these lines that can be easily conducted is to calculate the number of degrees of separation (six, to be precise) from a well-connected movie star: Kevin Bacon [7]. On the Internet, there is a website called omdb.org [8], which stands for open movie database. The license is Creative Commons [9], and allows the use of data on the website, as long as you reference the source. One can easily write a script, or use for instance Apache Solr [10], to crawl the set of pages listing the collaborating actors in a movie. This set, containing around 80,000 movies, can then be used by a MapReduce program to calculate the number of degrees between every actor in the list, relative to Kevin Bacon.

The crawled input data can be represented as a graph, in which the vertices are actors. The edges represent a collaboration between the two actors. A reduced example is shown in Figure 2.1, containing the main actors from the movies Apollo 13, Forest Gump and the television series House of Cards. The edges are dashed according to the movie they represent as shown by the boxes on the left of the image. If we now look at the link between Kevin Spacey, we can use his shortest path in our example. We do so, by taking a more abstract look at the problem, by giving each actor an ID. We also assign every vertex a relative distance, initialized to infinity, except for the distance of Kevin Bacon, linked to ID D. His relative distance will be initialized at 0. This graph, depicted in Figure 2.2 can be represented by a single file. This file contains a

line per vertex. For every vertex, we list the current relative distance, and the actors it collaborates with in a bi-directional way. This file will be the input for our calculation, as shown in Figure 2.3.

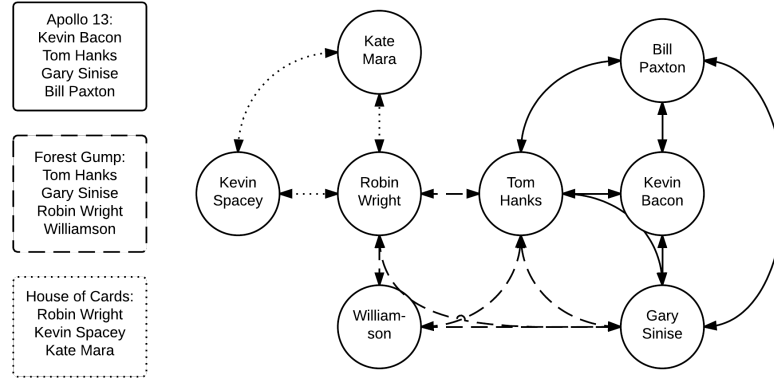


Figure 2.1: Movie information stored in a graph. Actor as a vertex, collaboration as an edge.

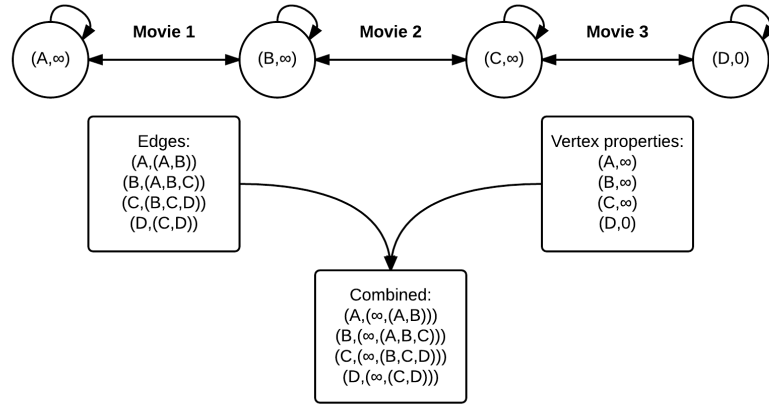


Figure 2.2: Movie information using IDs stored in a graph, two files and a single file respectively.

Every line from the previously mentioned input file will be sent to a map function, written by the user. In this map function we look at the key actor as the center of the universe, and all his neighbors are one hop away from him. Based on this, we create new nested key_value pairs, in which we assign a new distance of old distance + 1 to all neighboring actors. We also retain state, by creating a pair of key actor to key actor with new distance being equal to the old distance. The key_value pairs are in the form

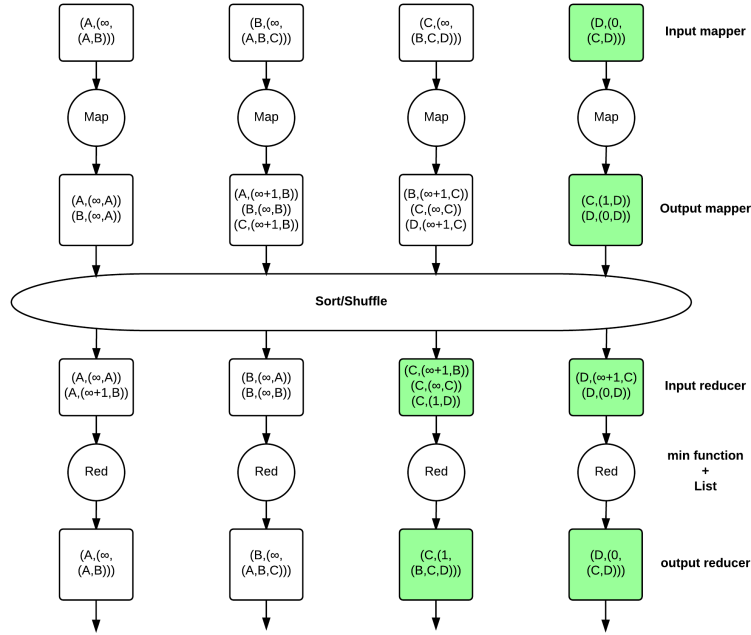


Figure 2.3: General data flow

of $\langle collaboratingActor, \langle newDistance, currentActor \rangle \rangle$. These key-value pairs will then be sorted based on their key, and used as input for the reduce function.

The reduce function receives the set of values belonging to a particular key stored in an iterator, and will iterate through this list looking for the shortest distance, while also rebuilding the list of collaborating actors. The final output values differs from the input in that all actors that are within one hop distance from Kevin Bacon are now assigned a distance of one. In our case, this is only Tom Hanks, with ID C. All other actors are still at infinity. The output is saved into DFS. Since we are looking for six degrees of separation, we will have to run this program five more times.

2.1.8 Limitations of the first generation

As we can see in this example, there are a lot of intermediate results from a single MapReduce iteration ($\langle actor, distance \rangle$) saved in and loaded from DFS. As the DFS is typically stored on HDD, this results in a significant overhead in the form of serialization, duplication, and HDD Input/Output (I/O). Another disadvantages is that a lot of static data ($\langle baseActor, List(collaboratingActors) \rangle$) has to be carried throughout the map and reduce phase in order to be able to save it together with the intermediate results, as a join of two separate input sets (one set containing the previous intermediate results, a second set containing the static data generated only once during the initialization phase) would require an extra MapReduce step. The preservation of this data also generates overhead, as it will need to be transferred over the network and written to the DFS,

often with a certain amount of duplication. This overhead can all be avoided, by mixing map and reduce phases, performing more efficient and in-memory computation.

2.2 Second generation

Due to the recent shift in needs towards iterative algorithms and interactive analysis tools, we now have a need for programs that will perform multiple MapReduce steps. As we saw in the example, every MapReduce step is traditionally followed by a store to the DFS, to secure the results. If we perform multiple of these MapReduce steps, we will need to load and store all data to DFS, taking huge penalties for the replication, HDD I/O and serialization. This same data is then loaded again. To avoid this, second-generation systems try to keep all intermediate data, both between a map and reduce task and between the reduce task and the following map task, in memory. This provides faster program execution, as DRAM is orders of magnitude faster as HDD, while still complying with our previously defined key characteristics: Resiliency, Scalability and easy of use.

2.2.1 Resiliency

Resiliency in second-generation systems is still guaranteed by rescheduling computation for lost partitions based on data in stable storage. But as we mentioned, second-generation systems will typically only hold input and output data in stable storage. To create resiliency with a minimal time penalty, one has to consider a tradeoff between time needed to guarantee resiliency during runtime versus time needed for recovery times the change of failure. Different implementations take a different position in this. ScaleOut's hServer will micro-checkpoint every step with a certain replica number [11]. Upon failure, one can reschedule computation on a different node containing a copy of the lost intermediate input set. Apache Spark allows user-defined check pointing, but mostly relies on fast re-computation of lost fractions of the dataset, by keeping track of the line of computation from the last known dataset, possibly being the original input set [2].

2.2.2 Bottleneck

Because we now mainly use DRAM as opposed to HDD, we do not have a problem with a mechanically moving disk reader, as we had in first-generation systems. DRAM can handle huge amounts of IOPS and has a very large bandwidth, but comes with limited capacity. This is currently handled by relying on the programmer to know which data will be re-used, and providing him with the means to select which data to cache. When the available space runs out, the system will then spill data to HDD, with expected performance gracefully degrading [2] [3] towards timings similar to a first-generation system.

2.2.3 Scalability

We see the same characteristics for scalability in second-generation Big Data systems that we saw in first-generation Big Data systems. There is not a lot of overlap between the

map, sort and reduce phase, as they are mostly sequential, but functions in every separate phase are highly concurrent. The need for data locality that is disappearing for first generation Big Data systems as network speed increases relative to HDD read bandwidths is still present in second-generation Big Data systems. This is the case because most of the computation for second-generation Big Data systems is done in-memory, which is still orders of magnitude faster than current commodity network speeds. We do see the effect of inefficient network performance, as our map, sort and reduce are orders of magnitude faster than before and do not mask bad network performance anymore. The tradeoff in network performance appears to have similarities to those in High performance Computing. When we have an unstructured input, a network such as fat-tree can be beneficial. This network is fast and scales easily, but expensive. A semi-structured input will benefit more from networks supporting locality, such as Torus, given that the application is not only moving the data to the computation, but is also aware of the data present on neighboring nodes. A Torus network is less expensive, but harder to maintain and to scale. It handles locality well.

2.3 Flash

In order to see how Flash can improve Big Data systems, we need to see how it relates in the areas that we marked as bottlenecks in Section 2.1.5 and 2.2.2, being IOPS and capacity. We can see in Table 2.1 that HDD has a great capacity, but not enough IOPS. DRAM can handle a great number of IOPS, but does not have a large capacity. Flash has both a great capacity as well as a good number of IOPS. Our goal is to see if the number of IOPS in combination with the larger capacity of a Flash system can provide a similar performance delivered by DRAM, for a tenth of the cost.

	DRAM	Flash	HDD
Type	DDR3 1600	SATA	SATA
Bandwidth	12.8 Gb/s	12 Gb/s	6 Gb/s
IOPS	100,000,000	100,000	100
Capacity	8GB	240Gb	4Tb
Costs	\$70	\$130	\$140

Table 2.1: Approximated performance characteristics and costs of DRAM, Flash and HDD.

2.4 Research questions

The following three chapters will introduce specific implementations of a first and second-generation Big Data system, and a specific type of Flash system. These systems will provide a platform for a deeper evaluation of our research questions, introduced in Section 1.2 and here repeated for clarity.

Research questions:

1. How does a second-generation Big Data system perform?
 - (a) How does a second-generation Big Data system perform relative to a first-generation Big Data system?
 - (b) How does a second-generation Big Data system perform when we constrain available memory?
2. Can Flash technology contribute to second-generation Big Data systems?
 - (a) Can we improve second-generation Big Data systems by using Flash technology connected as a file storage
 - (b) Can we improve second-generation Big Data systems by using Flash technology connected as an object store?

For a first-generation Big Data system, we have chosen Apache Hadoop [12] because of its wide adoption in industry. The second-generation Big Data system that we will evaluate is Apache Spark [13], because of its promising outlook based rapid developers growth, interest from industry and results in benchmarks tests such as the Gray Sort competition [4]. For our Flash system, we have chosen the IBM Flash System 840, connected via Coherent Accelerator Processor Interface (CAPI), as CAPI allows us to interact with Flash without making system calls to the operating system.

First generation: Apache Hadoop

3

For our first-generation Big Data system we have chosen Apache Hadoop. Apache Hadoop is widely accepted in industry, being used in companies such as IBM, Google and Facebook [14]. We will first explain how Apache Hadoop implemented the MapReduce framework, after which we will see the effects on our example. Using this example, we will highlight the limitations of Apache Hadoop and first-generation Big Data system in general.

3.1 Program implementation

In this section we will describe how a typical program is executed. We will then explain how Apache Hadoop complies with the three characteristics of a Big Data systems, being scalable, resilient and easy to use.

3.1.1 Program execution

A MapReduce task is defined by the user according to the programming model as defined in 2.1.1. The input data is loaded into Distributed File Systems (DFS), according to a default or user specific partitioning. Every node in the cluster will start a copy of the program, and one node will become the master node. All other nodes with the exception of the secondary backup master will be workers. The master node consists of a Jobtracker and a Namenode. The Jobtracker keeps track of the status of all map and reduce task to be executed, by maintaining a list of tasks together with the status "idle," "in progress," "completed" or "failed." The Namenode maintains a list of locations of intermediate and final results. The workers consist of a Tasktracker, to execute tasks, and a Datanode, to manage their partitions of the distributed file system [15]. This is depicted in Image 3.1.

On program start, the Jobtracker will select map tasks and try to assign them to workers that have the primary data file, or a backup copy, to move the computation to the data. A worker will first process the contents of the corresponding input split. For these splits, it will generate key_value pairs, and pass these to the user defined map function. This function will process the pairs, and store the results sorted in local memory. It will then provide the Namenode with the location of the results and ping the Jobtracker that it is idle. The Jobtracker will then again issue a new task, based on available splits at the machine.

A reduce task will be scheduled as soon as its input data is available. Backup tasks of reduce tasks in the final stage of the program are run, to avoid overhead

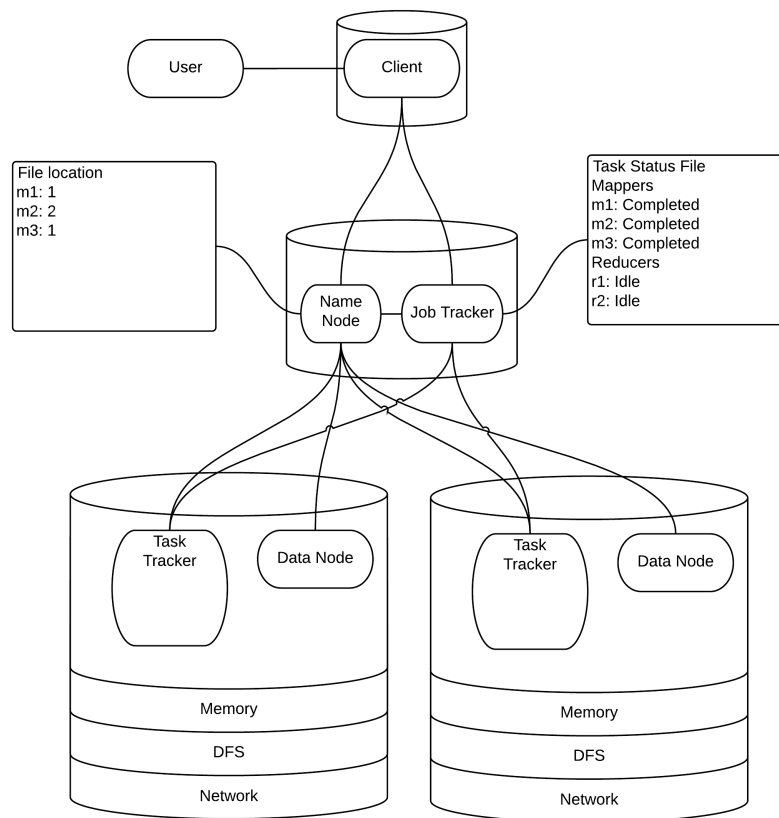


Figure 3.1: Hadoop execution overview.

incurred by machines with partial failure, that make slow progress. A reduce task reads the distributed intermediate results by remote procedure calls, and merges them with the local results. It then iteratively passes an intermediate key_value pair consisting of a unique key and it's corresponding set of values to the user defined reduce function. Results are appended to a private temporary file in the Hadoop Distributed File Systems (HDFS). This filename is atomically changed to the final name upon task completion. No further (backup) tasks are able to commit to this completed file, in order to remain correctness. The files in HDFS are kept with a certain replica number for both resiliency and high data availability when moving computation to the data. This number is typically three: one on the target machine, one on a node in the same rack for data locality reasons, and one outside of the rack to counter rack failure. This is called a checkpoint.

3.1.2 Resiliency

As clusters contain up to thousands of commodity hardware, failure is to be expected. In order to cope with failure, Apache Hadoop implemented a heartbeat signal and a checkpoint system. The checkpoint system is explained in the last paragraph of the previous

subsection. The heartbeat periodically tells the master node that the worker is alive and running. If the worker fails to send this signal for longer than a pre-determined period of time, the master assumes failure. In the status file, tracked by the job tracker, all map tasks executed or currently executing on the failed machine are marked as "failed," as are the currently executing reduce tasks. These "failed" tasks will be rescheduled for execution, as the results are lost. The rescheduled task will be scheduled on a node containing a backup copy. These are created any time the system writes results after the reduce phase, as otherwise node failure would incur failure of the data associated with it. The only tasks that do not have to be recomputed, are reduce tasks that are marked as "completed," as the results of these tasks are stored with replica in the HDFS. Master failure is dealt with by running a passive secondary master that keeps in sync with the primary, and can take over upon failure [16].

3.1.3 Scalability

As we already explained in Section 2.1.2, the MapReduce program allows for massive parallelism between map tasks and reduce tasks. A worker can be added to the cluster by running a copy of the program and adding it to the list of available machines. It will then query the master node for a task, and start executing it. A cluster can also scale down, as it can simply disconnect the worker. Consistency will be handled by the fault tolerance system.

3.1.4 Ease of use

The user only has to write a piece of code for the map phase and the reduce phase, together with a job specification. For our experiments, we have developed multiple pieces of code, introduced in Chapter 6: Experiments. Code for the following example is also introduced here.

3.2 Example

We will now take a look at our example implementation, introduced in Section 2.1.7. In order to implement the algorithm, we need to have at least two steps. An initialization step and one or more iterative steps. The initialization step takes the input data and generates a text file representing the graph in Figure 2.2. This graph is used as input for the iterative phase. The iterative phase consists of 6 map and reduce tasks, with an implicit sort in between. Let's first take a look at the input data.

As explained in Section 2.1.7, we have input files crawled from omdb.org [8]. While crawling, we have formatted this data into a single file. In this file, every line represents a set of edges between nodes, corresponding to a connection based on the set of actors that played together in this movie. This file is uploaded into HDFS, and partitioned into chunks based on the default partitioning scheme.

Once a task tracker receives a map task, it will retrieve a record. This record is

in our case a line in the file, containing a set of collaborating actors. As we can see in Image 3.2, we will first apply a map function to this input set, to create key_value pairs. If we have three actors working in the same movie, we will create $3^3 = 9$ key_value pairs, stored sorted for this output set in local memory. For every actor, we create pairs linking this actor as the key to all collaborating actors, and to itself. The link to itself is in order to remain state, as we want the new distance to remain the same if no better distance is found. Once this is performed for every line, we will have created a bi-directional graph representing all collaborations between actors. This graph still contains a lot of duplicates, so we perform a reduce step. In this step, we filter the duplicates, and create a list of collaborations per actor to save space. We do so by grouping all intermediate keys I, together with it's corresponding values. When merging, we check for duplicates, and ignore these. All results of the reduce tasks are saved back into HDFS. This file now consists of lines containing a unique actor, together with a list of collaborating actors. We also stored the actor's initial distance to our target actor: Kevin Bacon. This initial distance will be infinity by default, except for Kevin Bacon, who will be initialized to 0.

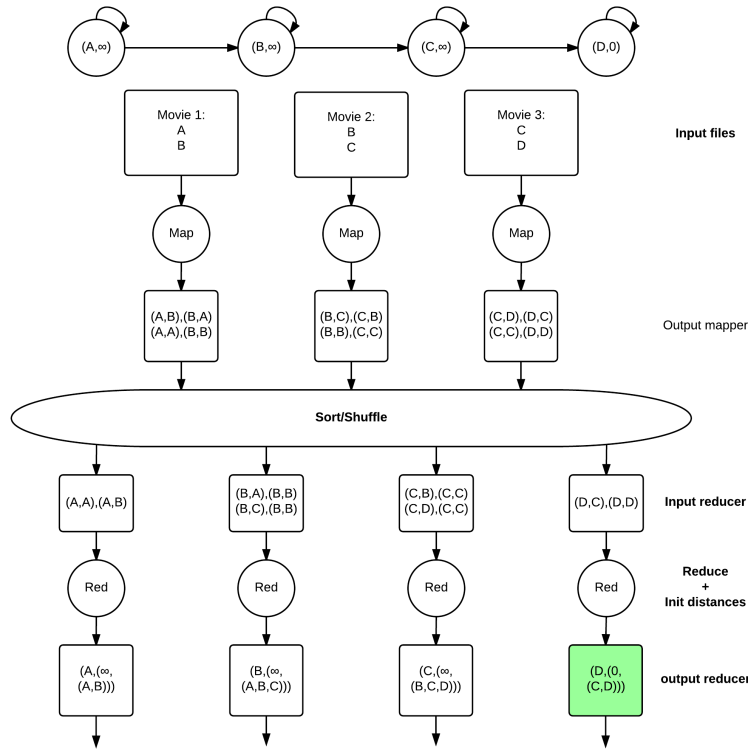


Figure 3.2: Execution flow in Hadoop during the initialization part of the example.

Now for the iterative part, as shown in Figure 3.3, we load this newly created file from HDFS again. Every map function again receives 1 line from this file, containing the list

of collaborations for one actor, which we will refer to as our key actor in this paragraph. We now assume that every collaborating actor of this key actor is at a maximum of (key actor's distance) + 1 hops away from Kevin Bacon. To indicate this, we create nested key_value pairs. In these pairs we store a collaborating actor as the key, and in the value we store another key_value pair with the (key actors distance) + 1 as the key, together with the ID of the key actor. This last part is static data, but needed to rebuild the collaboration graph. The results are again sorted and stored locally. The reducers will retrieve and merge the records based on the unique key, together with their corresponding set of values. These values represent a set containing all maximum distances for one actor, based on the distances of its neighbors. We can now reduce this set by looking for the minimal value, while rebuilding the list of collaborating actors for this current actor. These results are saved into HDFS again. It will take six iterations for all actors that are within six hops distance to Kevin Bacon, to retrieve their actual distance. All other actors will remain at infinity in this implementation.

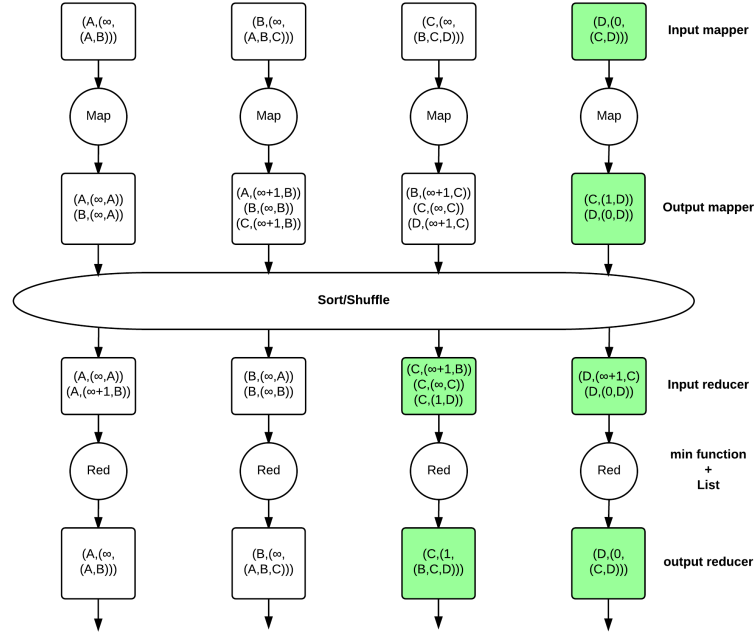


Figure 3.3: Execution flow in Hadoop during the iterative part of the example.

3.3 Limitations

In this example it becomes clear that a first-generation system will always write it's data back to HDFS. This induces overhead in the form of Hard Disk Drive (HDD) I/O, serialization and replication overhead. Apache Hadoop is also not able to make an efficient distinction between static (the list of collaborating actors) and dynamic (the intermediate distances). As we will see, Apache Spark is able to mix multiple map and

reduce phases, allowing for a join before a map phase, after which this data is stored in cache without flowing through the data in the form of key-value pairs. Apache Spark also does not have to write to HDFS after every reduce phase, as it is able to re-compute based on lineage. All this is explained in more detail in Chapter 4.

Second generation: Apache Spark

4

Apache Spark is created by the same team that also designed the Resilient Distributed Dataset (RDD) abstraction, located at the AmpLab, as part of the University of California: Berkeley. Apache Spark targets the iterative algorithms and interactive analysis tools that fall short in the first generation [2]. The core is written in Scala, but it also provides a wrapper for both Java and Python. The command-line interface is only available in Scala. Apache Spark is more than just a MapReduce platform. The program comes with libraries for graph computation, machine learning, SQL and streaming. The focus of this thesis lays on Apache Spark's implementation and extension (such as join operations) of MapReduce library, but all ideas can be extended to the others libraries as well, as all rely on this implementation for their computation. This is depicted in Image 4.1. We will start this chapter by introducing the RDD abstraction on which Apache Spark relies for its MapReduce computation. We will then look at the change in the dataflow of our example, followed by an evaluation of the spill behavior of Apache Spark to see where Flash could fit in. We will then refine our two Research Questions, by specifying these locations in the questions, and also formulate a Hypothesis.

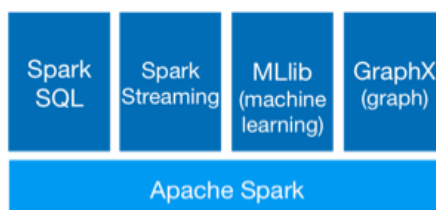


Figure 4.1: Different modules in Spark. Source: <https://spark.apache.org/>

4.1 Resilient Distributed Datasets

Resilient distributed datasets are at the very core of Apache Spark. RDDs are fault-tolerant data structures that provide an interface based on coarse-grained transformations on the partitioned data it contains. They are immutable, and can only be created through deterministic operations on either data in stable storage, or through transformations on other RDDs. Examples of such transformations are map, filter and join. An RDD consists of five pieces of information [2]:

- A set of partitions containing the data stored in the RDD
- A set of dependencies on partitions in parent RDDs
- Information about the transformation to perform on the partitions it depends on

- Metadata about its partitioning scheme
- Metadata about its data placements

The input data can be loaded into a Distributed File Systems (DFS), such as the Hadoop Distributed File Systems (HDFS). The HDFS system uses either the default or a user defined partitioning scheme, to distribute the data across the nodes. This input data can then be loaded as an RDD in Spark. We will refer to this RDD as the base RDD. From this base RDD, we can create new RDDs by performing coarse-grained transformations, such as map, filter and join. These RDDs will follow the same partitioning unless otherwise specified. On these new RDDs we can either perform more transformations, or perform an action. An action will either return a number, for instance "count" returns the number of elements in the dataset, or allows to save the dataset to for instance HDFS. The reduce function can be both a transformation and an action, as we can perform a "reduceByKey" to create more key value pairs, or a "reduce" in order to implement a function similar to count and create a single answer. A list of all transformations and actions can be found at [17].

All partitions in a new RDD depend on the partitions of their parent RDD. The new RDD will also follow the same partitioning as the parent RDD. This dependency can be a one-to-one, one-to-many, many-to-one or many-to-many dependency, depending on the type of transformation or action. A one-to-one dependency is called a "narrow" dependency; all others are called "wide" dependencies. Examples of both "narrow" and "wide" dependencies can be seen in Figure 4.2. As we always aim to move the computation to the data, we can see that "narrow" dependencies are relative fast, as both the parent's and child's partition will reside on the node, and no network communication is required. "Wide" dependencies also require data from other RDDs, and might require network communication. These types of dependencies incur more overhead, as we now have to move the computation to the place where we expect most data to reside. All other data has to be gathered, possibly over the network, involving larger delays.

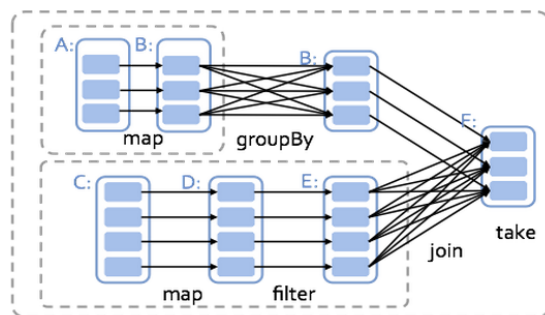


Figure 4.2: Dependencies in Spark. Source: <https://databricks.com/blog/2014/03/20/apache-spark-a-delight-for-developers.html>

Because we know that any RDD is immutable and new RDDs can only be created by performing logged transformations on already existing RDDs that always trace back to the base RDD, we know for every partition in any RDD which data is required to perform the required calculation. This trace is called the RDDs lineage and can be modeled as a Directed Acyclic Graph (DAG). This DAG is the basis for program execution. The lineage brings two big advantages: resiliency and lazy evaluation, as explained in the next subsections.

4.1.1 Execution

The execution closely resembles the execution in Apache Hadoop, with a difference in the formulation of jobs, since we no longer have a strict map-sort-reduce flow. For this we use the before mentioned DAG. This DAG contains all RDDs, structured according to their dependencies. This DAG can be divided into chunks of calculation of which we now can be computed on one node, without network communication. One chunk consists of a list of zero or more consecutive RDDs with narrow dependencies and possibly one wide dependency or an action (save) at the end. These chunks are called stages, and all narrow dependencies in a stage are pipelined. Transformations that do not lead to an action are disregarded. The process of pipelining instruction until an action is taken, and disregarding transformations that do not lead to actions is called lazy-evaluation. The results of all transformations in one stage only require one RDD. In Image 4.2, we see that task *A* and *B* can be merged, as well as task *C*, *D* and *E*. This leaves us with a total of four stages: $\{A, B\}$, $\{C, D, E\}$, $\{B'\}$ and $\{F\}$. On program start, we no longer have a list of map and reduce jobs the scheduler chooses from, but a list of Stages and their dependencies. We schedule the stages which dependencies are resolved first, being the stage that depends upon the base RDD. In our case there are two base RDDs and two stages, being $\{A, B\}$ and $\{C, D, E\}$, that are ready to be scheduled. One is picked arbitrarily, and scheduled. This process continues until all required stages for the output are computed, after which the program terminates.

4.1.2 Resiliency through lineage

Another great advantage of the lineage is the ability to recover lost partitions by rescheduling just the computation necessary to recalculate only the lost partition, as opposed to checkpointing entire output RDDs from stages with a certain replica number. This saves both memory capacity, which is scarce, and time penalty for extra I/O overhead for writing to persistent storage such as Hard Disk Drive (HDD). Another advantage is that because we do not require an action to checkpoint data after a reduce step, we can now pipeline more calculations, as we do not have to wait between a reduce and a map phase. This is demonstrated in the example in Section 4.2.

4.1.3 Memory management

Input data is typically stored in stable storage, usually being HDFS. On program execution, the relevant data is loaded into memory, and a number of pipelined transformations are performed, in order to create a new RDD. The amount of available memory for Spark

is defined at the start of the program¹. This amount is then divided into 3 regions and assigned a default fraction of the total available memory [18].

- General: 20
- Shuffle: 20
- Storage: 60

Every RDD at the end of a stage will reside in the Shuffle region, as the end of a stage is depicted by a sort. All other transformations are pipelined, and will not instantiate any other RDDs. When one chooses to explicitly cache an RDD into memory, the RDD moves from, and frees up space in, the shuffle region and moves into the third memory region: Storage. This does guarantee availability², but does not guarantee that the RDD is stored in memory, as it can be spilled to HDD when the Storage region runs out of memory.

4.2 Impact on example

The example now changes in multiple important ways, of which the effects will be discussed in the next subsection. Firstly, when sufficient memory is available, and the user does not explicitly persist to HDD, all data except for the input and output is kept in memory. Because of the resiliency through re-computation, we do no longer require checkpointing to a DFS saved on HDD, having only a stop-all like scenario in cases with many dependencies, such as sort. All others transformations can be pipelined. Secondly, we can separate the static and intermediate data, as a hash partitioned join only has a narrow dependency, and does not require an extra map-reduce step, as it would in Hadoop MapReduce. Because this is no longer required, we save on the amount of data that flows through the computation. To show how we can split the static from the intermediate data, let us first take a look at the initialization step in Spark, as show in Image 4.3.

As we can see, there is no difference with Apache Hadoop in the map and sort phases. There is however a difference after the sort phase. Apache Spark does not require a strict set of map, sort and reduce steps. Because of this, we are able to perform two steps after the sort. The first step creates a list containing the collaborating actors for one actor. We will refer to this RDD as being static, as the same data will be read multiple times. The next RDD will be referred to as being dynamic, as it contains intermediate results. These intermediate results will be read once, after which this RDD can be forgotten. It cannot be re-used for further storage, as RDDs are immutable. Because we will re-use our data, we will explicitly cache it. These two separate RDDs will now go into the iterative step, as shown in Image 4.4.

¹Available memory is set in `[SPARK_HOME_DIR]/conf/spark-env.sh` with `"SPARK_DRIVER_MEMORY=[x]g"`

²Assuming default settings: spilling to disk enabled and persisting with storage level `"MEMORY_AND_DISK"` [19]

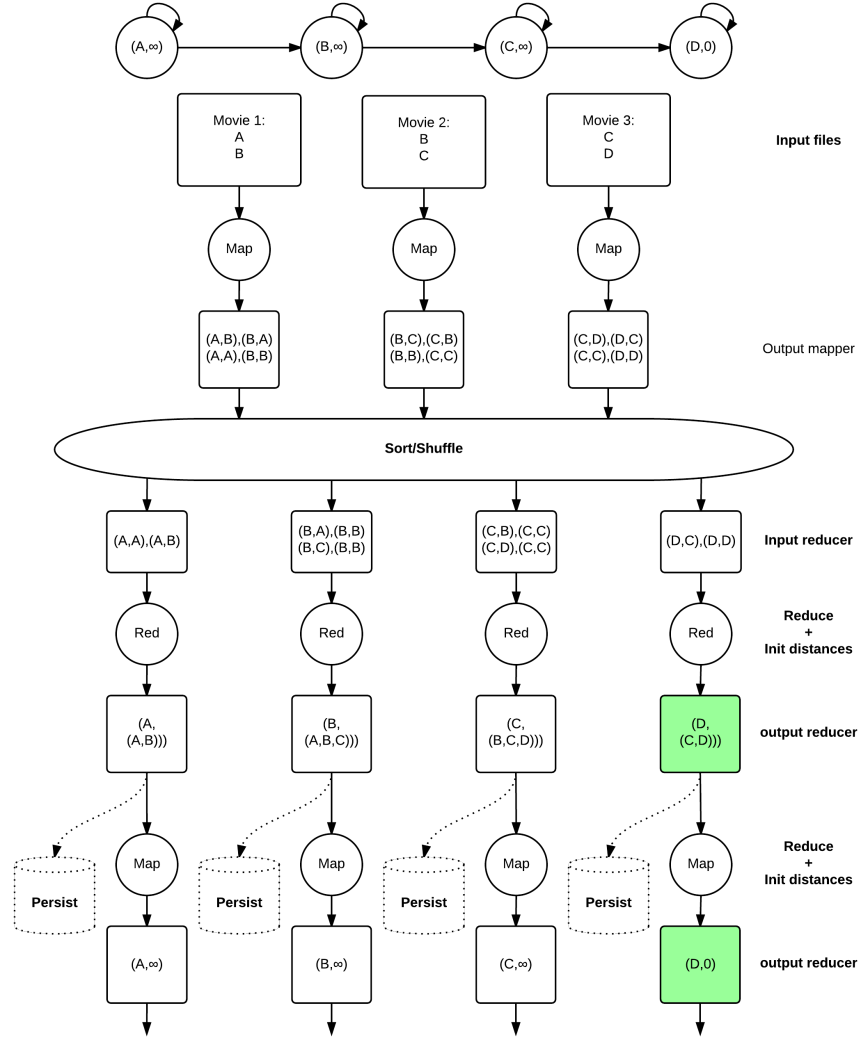


Figure 4.3: Execution flow in Spark during the initialization part of the example.

Now that we understand program execution, we can take a closer look at the generated RDDs, as shown in Image 4.5. Since both input RDDs for the iterative steps are partitioned similarly, we can perform a join with a "narrow" dependency that has little impact on the performance, especially since the result of the initialization is kept in memory as opposed to DFS stored on HDD. With this join, we create the required data to generate the same key_value pairs as we did before, assigning a distance + 1 to all collaborating actors. The difference here is that we can immediately forget about the current actor, as we do not have to rebuild the graph, because we cached it in memory. As the limiting factor in second-generation big data systems is the memory capacity, and the impact of this save is large because it concerns a lot of data, this makes a big impact. The intermediate results are then again sorted based on key, and the shortest distance is

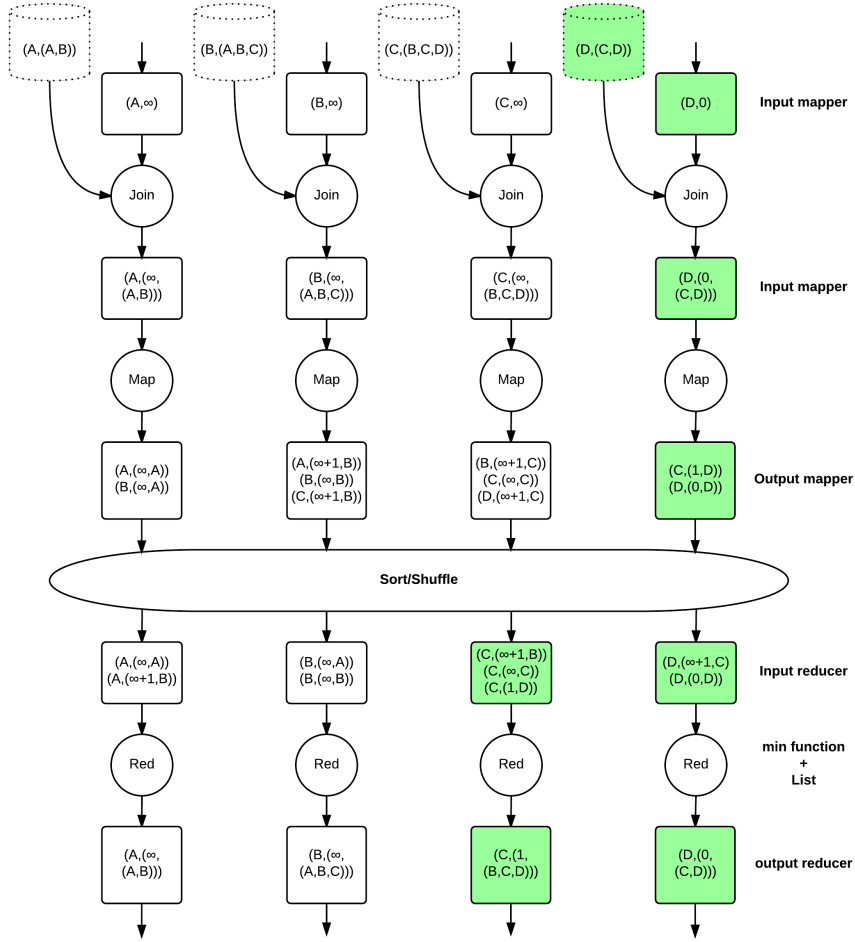


Figure 4.4: Execution flow in Spark during the iterative part of the example.

selected from the available options in the set of values corresponding to the unique actor. This information is stored in-memory in a new intermediate RDD, after which we can go directly into the second iteration. This is depicted in Image 4.5. We can see here, that the reduce, join, map and sort can all be pipelined, as there is no global checkpoint anymore.

4.3 Data storage in Apache Spark

Apache Spark currently uses two types of media to write data to: HDD and Dynamic Random-Access Memory (DRAM), as shown in Image 4.6. Input and output data are generally written to the DFS, and stored on a HDD. Intermediate data will have a preferred location of DRAM, but will be managed by a user-level memory controller. There are three memory regions on a user-level in Apache Spark, and each region has a default percentage from the total available memory. This is listed next:

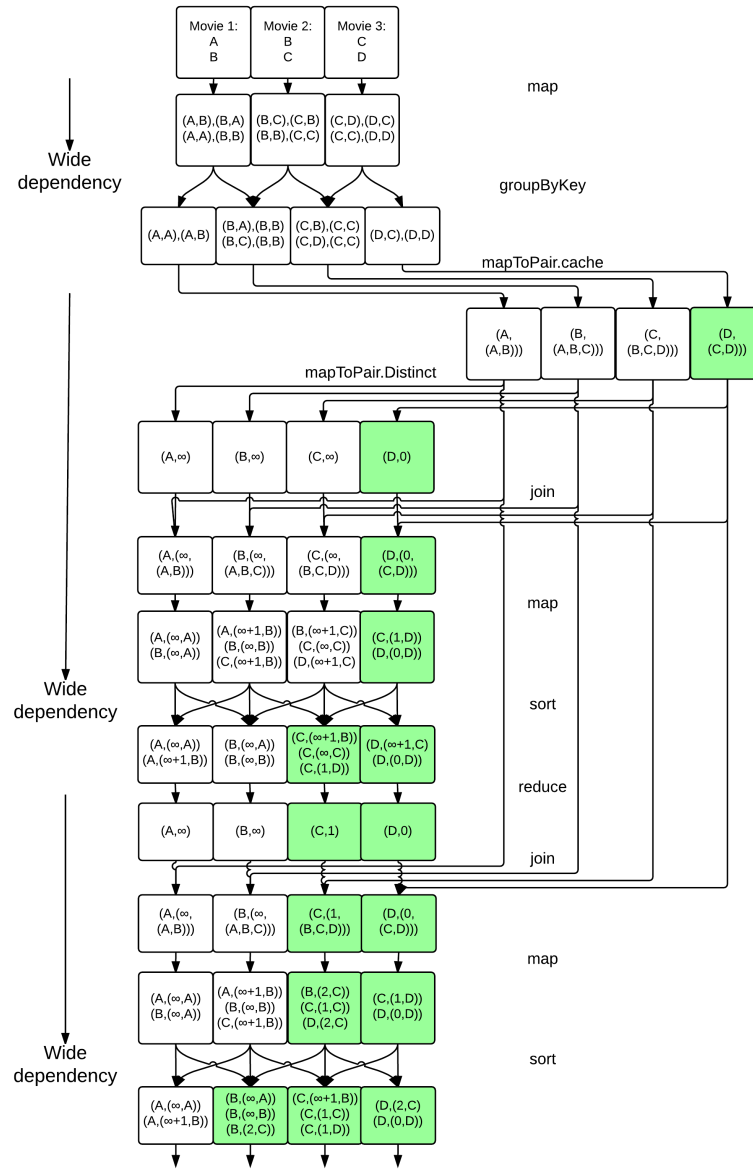


Figure 4.5: RDDs and dependencies in our implementation of the example in Spark.

- General: 20%
- Shuffle: 20%
- Storage: 60%

The general region will be used mostly by the Apache Spark core. The other two regions are used to store intermediate data. As we explained in Subsection 4.1.1, computation is combined into stages. As all stages either end with a sort or an action (such as save to DFS), and all other data is pipelined to the end of the stage, we see that RDDs will typically reside in the Shuffle region. This can be altered, when the user decides that it wants to cache a particular RDD. This RDD will then be moved towards the Storage region. This does guarantee availability, but does not guarantee that the RDD will be stored in memory, as this is up to the memory controller. The memory controller will decide per-thread and per-region upon a capacity budget. When a block does fit within this budget, it is stored into DRAM. When it does not fit, it will be spilled to HDD. This will be explained in the next two subsections.

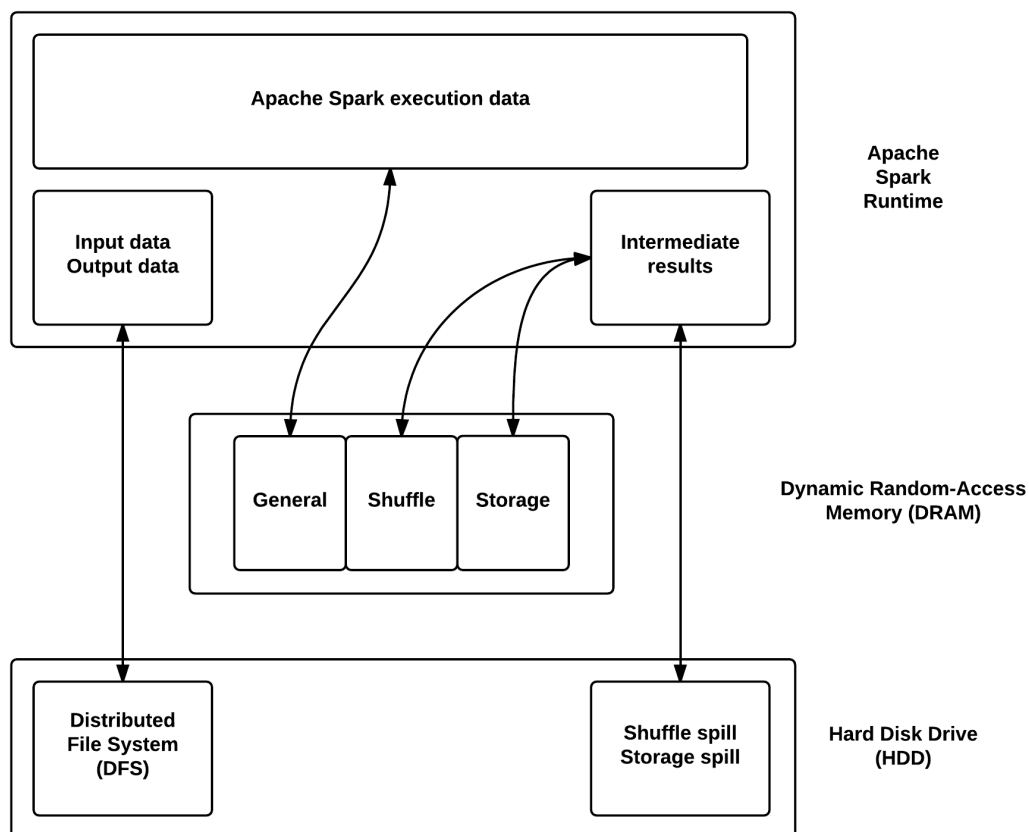


Figure 4.6: Different ways to connect Flash in Apache Spark.

4.3.1 Spilling in the storage region

Whenever a block is trying to be placed in memory, it will call `tryToPut()` in file `MemoryStore.scala`³. This will call `ensureFreeSpace()` in the same file, which will calculate the available space. `ensureFreeSpace()` will then return with `true` to indicate there is enough free space available and reserved, or with `false` in order to indicate that the block needs to be spilled. `tryToPut()` will act accordingly by either storing the blocks in memory, or dropping the blocks to disk.

4.3.2 Spilling in the shuffle region

Spilling in the shuffle region happens both in the `ExternalAppendOnlyMap.scala`⁴ and `ExternalSorter.scala`⁵. Both will try to request using `tryToAcquire()` the necessary amount of memory to continue sorting from `ShuffleMemoryManager.scala`⁶. `ShuffleMemoryManager` will evaluate this request, and based on fair division of available memory between threads either grant or deny. It will use the iterator, to retrieve every key-value pair, and place it in an object, managed by the block manager. When this buffer reaches the user's defined threshold number of elements and spilling is enabled⁷, Apache Spark will start spilling to disk.

4.3.3 Performance

It is claimed [20] that Apache Spark is expected to always perform better or similar to Apache Hadoop. Empirical results reported in [2] and [3] show that the performance of Spark is to degrade gracefully when memory is gradually constrained.

³Code of file `MemoryStore.scala` is available for reference in Attachment B.1

⁴Code of file `ExternalAppendOnlyMap.scala` is available for reference in Attachment B.2

⁵Code of file `ExternalSorter.scala` is available for reference in Attachment B.3

⁶Code of file `ShuffleMemoryManager.scala` is available for reference in Attachment B.4

⁷Spilling is enable by setting `spark.shuffle.spill` to `true` in `spark-defaults.conf`

4.4 Research questions

Now that we decided to use Apache Spark and Apache Hadoop, we can further refine our research questions. We can also formulate our first hypothesis, based on the sources presented in Section 4.3.3.

Research questions:

1. How does Apache Spark perform?
 - (a) How does Apache Spark perform relative to Apache Hadoop?
 - (b) How does Apache Spark perform when we constrain the available memory?
2. Can Flash technology contribute to Apache Spark?
 - (a) Can we improve Apache Spark by using Flash technology connected as file storage?
 - (b) Can we improve Apache Spark by using Flash technology connected as object store?
 - i. In the shuffle region
 - ii. In the RDD storage region

Hypothesis for the research questions:

1. (a) Apache Spark is expected to always perform equal or better than Apache Hadoop.
- (b) The performance of Apache Spark is expected to degrade gracefully when gradually constraining available memory.

Flash

Now that we have a deeper understanding of Big Data systems, and specifically have seen in Section 4.3 how Apache Spark stores its data, we are going to take a look at the potentials for using Flash technology. We will first take a general look at the ways we can store data in a system, and the amount of overhead involved. Based on this, we will look how Flash Technology can be used in a Power8 system. We will then look which of these methods can be applied to improve the performance of Apache Spark.

5.1 Ways to store data

In general there are four ways to store data:

1. Serialized in the file system
2. In a key_value store via the Operating System
3. In a key_value store that is user accessible
4. As object store in memory

In Section 4.3, we have seen that Apache Spark currently uses two of these: Serialized in the file system and as objects in memory. In the next subsection we will examine the overhead incurred by all four techniques.

5.1.1 Serialized in the file system

Apache Spark uses the Hard Disk Drive (HDD) both for storing the Distributed File Systems (DFS) and as a location for spilling data, as explained in Section 4.3. When data is about to be stored on HDD, Apache Spark will first aggregate data, in order to transfer more data to minimize overhead. Once the data is aggregated, Apache Spark will make system calls to the Operating System (OS) and involves the files system and device drivers. Involving the OS incurs a lot of overhead, and can also be suspended due to context switches and interrupts.

5.1.2 In a key_value store via the Operating System

Storing data in a key_value store is typically used to store a lot of small objects, whereas the file system typically stores less, but larger blocks. This means less time is spent aggregating data. Most systems still have to involve the OS, and make use of the device drivers. In general, overall overhead has to be lower, as there is less data being transferred in one copy.

5.1.3 In a key_value store that is user accessible

Data can also be stored in key_value pairs that are user accessible. This has as an advantage that data now resides in the user addressable memory space, and data transfers no longer require involvement of the OS, greatly reducing the amount of clock cycles but also removing the extra overhead in the form of waiting on context switches and handling interrupts. The types of overhead and the management of the data is similar to that of virtual memory.

5.1.4 As object store in memory

Data stored in memory can be accessed directly from the processor, as it resides in the user address space. It can be accessed with a high bandwidth and large number of IOPS, but memory generally does not have a large capacity.

5.2 Ways to use Flash

Flash technology can store data in all ways described in Section 5.1. In this section, we will look at specific implementations of Flash in the Power8. We will use Figure 5.1 (ToDo: update figure, (is DIMM still correct? I changed this? + number verwijderen)) to illustrate this.

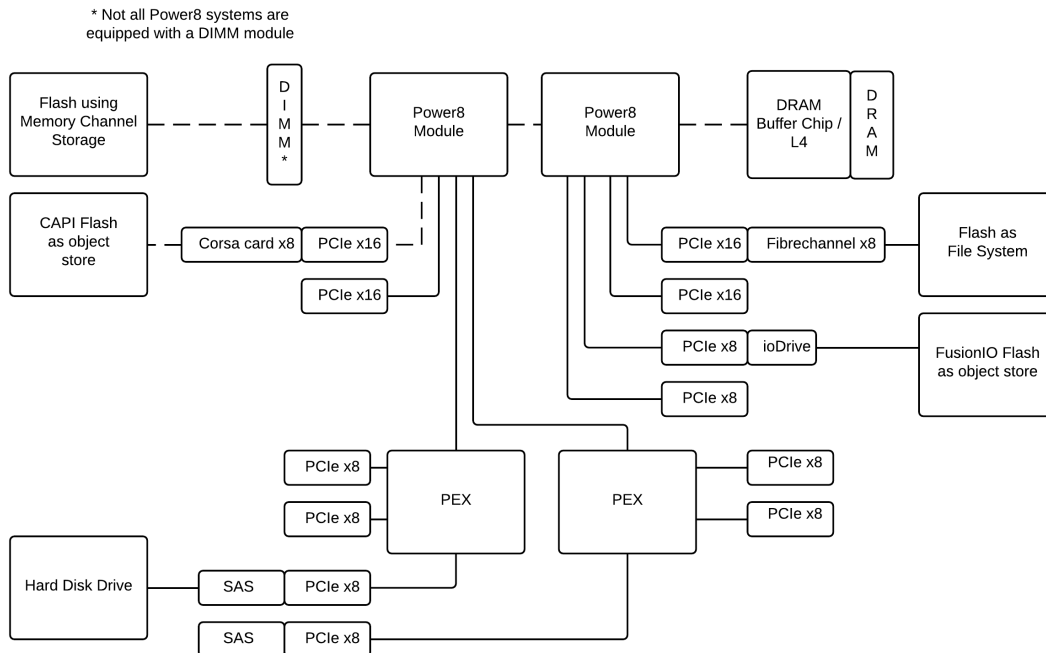


Figure 5.1: Different ways to connect Flash in Power8

5.2.1 Supporting a file system

A file system can be installed on a Flash system, so that it can be used to store large blocks on. This can be done in multiple configurations, as explained in the next subsections.

5.2.1.1 Flash as SSD

Storing blocks on a Solid State Disk (SSD) will undergo the same overhead as storing blocks on a HDD, but a SSD typically has more bandwidth and Input/Output Operations per Second (IOPS), as seen in Table 2.1.

5.2.1.2 Flash using FibreChannel

IBM's FlashSystem 840 can be connected via a FibreChannel and have a file system mounted on it. As the FibreChannel is connected directly into the PCIe slot, it will be closer to the processor as it would be using SAS connected to the PCIe switch (PEX). Data transfers still have to use the OS. This is depicted in Image 5.2 on the left side.

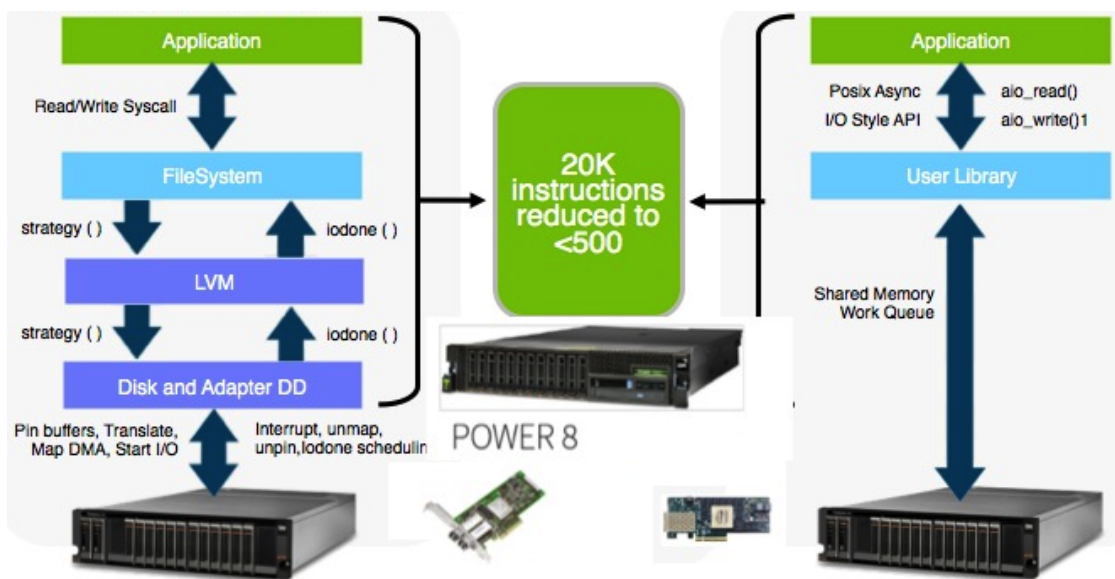


Figure 5.2: Left: FlashSystem connected as a file system. Right: FlashSystem connected using CAPI Flash. Source: IBM internal communications

5.2.1.3 Flash using VSL

Fusion's Virtual Storage Layer (VSL) virtualizes NAND Flash arrays, combining the advantages of virtual memory architecture with a transactional file system. This leads to a system that is closer to the processor with a faster access path, but still suffers from the involvement of the OS.

5.2.2 In a key_value store via the OS

Flash can be configured to store key_value pairs in the name space, for instance connected via the PCIe slot. An example of a system using key_value store using the OS is Dynamo, designed for the Amazon Shopping Card [21].

5.2.3 In a key_value store in the user address space

Different implementations of Flash in the user address space exists. We touch upon Diablo's Memory Channel Storage (MCS), using DIMM, and IBM's CAPI Flash, using PCIe.

5.2.3.1 Flash using MCS

Diablo's MCS allows one to connect a Flash into a DIMM slot. MCS is fully compatible with the DDR3 protocol. The data remains local to the CPU in the user address space, but the data needs to be retrieved asynchronously. In order to do this, the application has to implement a polling mechanism, or use interrupts. Synchronous access is not supported.

5.2.3.2 Flash using CAPI

A device connected over CAPI shares memory cache coherently with other processors in the system. Other than by its execution mechanism, a computational thread on a CAPI attached accelerator can not be distinguished from a thread running on a SMP. As a consequence, user threads can communicate with a device through shared memory without intervening the OS and without a need for a device driver. This is shown on the right side in Image 5.2. On power8 the coherent communication is tunneled over a standard PCIe Gen3 bus, but only lower protocol layers are used, so there is still no need for a device driver [22].

5.2.4 Objects in memory

Flash can also be used as system memory. The main advantage is its ability to scale to a large capacity, but Flash technology does not work well for small block sizes. A way to overcome this problem is by using the limited amount of DRAM available in the system as a write-in cache.

5.3 Flash in Apache Spark

Now that we know the different ways Apache Spark stores data and Flash can be connected, we can look for opportunities to optimize. When we look again at Image 5.3, we see that there are four areas highlighted. We will discuss those in the next subsections.

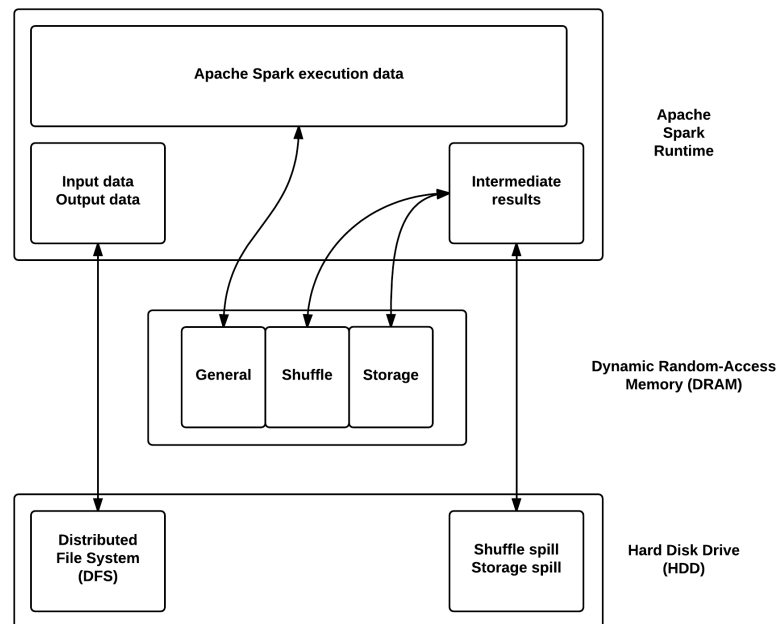


Figure 5.3: Different ways to connect Flash in Apache Spark. Dashed line represents data resides in the user address space.

5.3.1 Replace the HDD with Flash

Both the DFS and the object spill mechanism make use of the file system. This normally is installed on a HDD, but can be replaced with Flash. As we saw in Subsection 5.2.1, this will not impact the amount of overhead, but Table 2.1 does show that Flash has a better bandwidth and a higher number of IOPS.

5.3.2 Refactor the spill system to spill `key_value` pairs to Flash (secondary object store)

Currently, Apache Spark writes large blocks of spill data to the file system. This system can be refactored to spill to a Flash system, configured as `key_value` store, in order to spill smaller blocks (with less overhead from aggregation) to a location closer to the processor.

5.3.3 Refactor the data store to write `key_value` pairs to Flash (primary object store)

Instead of using memory, with the risk of running out, we can also refactor Apache Spark to use Flash, together with a limited amount of Dynamic Random-Access Memory (DRAM) as write-in cache, as it's primary object store. We can then remove the Apache Spark memory manager, as we can expect Flash to have enough capacity.

5.3.4 Reconfigure the DRAM to be used together with Flash (Flash as memory)

Currently DRAM has good writing speeds, but the operating system will decide to page data to HDD when it is close to reaching the memory capacity limit. We can use Flash, in combination with DRAM connected as write-in cache, to fully replace the memory, as Flash has a large capacity.

5.4 Research Questions

Research questions:

1. How does Apache Spark perform?
 - (a) How does Apache Spark perform relative to Apache Hadoop?
 - (b) How does Apache Spark perform when we constrain the available memory?
2. Can Flash technology contribute to Apache Spark?
 - (a) Can we improve Apache Spark by using Flash technology connected as file storage?
 - (b) Can we improve Apache Spark by using Flash technology connected as secondary object store?
 - i. In the shuffle region
 - ii. In the RDD storage region

Hypothesis for the research questions:

1.
 - (a) Apache Spark is expected to always perform equal or better than Apache Hadoop.
 - (b) The performance of Apache Spark is expected to degrade gracefully when gradually constraining available memory.
2.
 - (a) We expect to see a small improvement, as the only change in overhead is the characteristics of HDD versus Flash.
 - (b) We expect to see a larger improvement, as we do not only have better performance characteristics because of the use of Flash versus HDD, but are also share memory cache coherently with the processors over a fast PCIe connection.

Experiments

In order to answer our research questions, of which the final form is presented in Section 5.4, we will perform two experiments. The first experiment will give us insight in the behavior of a second-generation Big Data system. We would like to see how a second-generation relates to a first-generation Big Data system, in order to justify our focus on second-generation Big Data systems. We would also like to see how a second-generation Big Data system behaves when constraining the available memory, in order to see if there is room for improvement. The second experiment will try two methods presented in Section 5.3, to see if Flash technology can provide this improvement.

6.1 Experiment 1: How does Apache Spark perform

In this section, we will first list the used hardware and software, together with its configurations. We will then explain our two sub experiments, present the results and give an evaluation. A final conclusion is drawn in Chapter 7.

6.1.1 Hardware

The first experiment is conducted on an IBM Power System S882L. The S882L is equipped with two 12-core 3.02 Ghz Power8 processor cards, and has 512 Gigabyte 1600 MHz DDR3 memory. Only one hard disk is in use.

6.1.2 Software

We installed Ubuntu server 14.04 Little Endian [23]. Instructions how to do this are available in Appendix C.1. In our experiment we use Apache Hadoop and Apache Spark. Both are build from source for the PowerPC64 platform. We used Java version 7.1 developed by IBM for the ppc64le, as this version was designed for our platform. It can be downloaded from the IBM website, when logged in as a registered user.

6.1.2.1 Apache Hadoop

We have downloaded the source files for Apache Hadoop 2.2.0 from [24]. We then build Apache Hadoop using the commands available in Appendix C.2.

6.1.2.2 Apache Spark

We have downloaded the source files for Apache Spark 1.1.0 from [25]. We then build Apache Spark using the commands available in Appendix C.3.

6.1.3 Algorithm

We believe that the lead example, the six-degrees of separation algorithm explained in Section 2.1.7, is a representable iterative algorithm. We implemented this algorithm in both Apache Hadoop and Apache Spark. Code for our implementation in Apache Hadoop can be found in Appendix A.1, code for our implementation in Apache Spark can be found in Appendix A.2. Data flow for our implementation in Apache Hadoop can be found in 3.2, dataflow for our implementation can be found in 4.2. We would like to point out that we are not looking for the most effective way to implement this algorithm, but for an equal implementation of a representable iterative algorithm in both Apache Hadoop and Apache Spark to provide a fair comparison.

6.1.4 Input set

As an input set, we originally used files from omdb.org. To better match our needs, we created an artificial input set based on the files available at omdb.org. This input set contains of one file with 10,000 lines. Every line contains 1 to 101 actors with an actor ID range of 10,000. The total file size is around 40 Megabytes.

6.1.5 Limitations

In order to get timing results for the different phases in Apache Spark, we had to combine certain information, as it is not possible to get accurate timing results from the main function. When you try to time the separate phases, you run into an overlap. In one stage of our program we perform a reduce, a sort and a map phase (in this specific order). A map tasks will only be scheduled for execution when all relevant info is available. This will only occur after the relevant partitions of the reduce tasks have completed and are sorted. However, when you place timers around these phases in the main function, you will see that all timers start at the same time and end at the same time, as all tasks for all phases in a stage are being evaluated for execution and only depend on a limited portion of the total number of partitions. Because of this, we have timed inside the specific functions. This is only possible for the reduce and the map phase, as the sort phase is implicit. This brings two limitations. The sort phase is calculated by $time(Sortphase) = time(stage) - time(reduce) - time(map)$. Because of this, all overhead is added to the timing of the sort phase.

We also have trouble with the first map and sort, as these are pipelined into a stage of which the execution belongs to the Initialization step. As we expect the time of an iterative step to be orders of magnitude smaller than time of the initialization step, we believe that this map and sort can be neglected in the total time of the initialization, and do not spend any time on this.

Apache Hadoop has a clear map, sort and reduce phase, and is therefor timed in the main function.

6.1.6 Experiment 1a: How does Apache Spark perform relative to Apache Hadoop?

In our first sub experiment, we are going to look at the relative performance of Apache Spark and Apache Hadoop. We will run both implementations multiple times on the same platform, and compare the average of the timing results.

6.1.6.1 Goal

The goal of this experiment is to verify our assumption that second-generation Big Data systems have an equal-or-better performance relative to first-generation Big Data systems. We will use this information to justify our focus on second-generation Big Data systems. As we believe Apache Spark has better performance for iterative algorithms, we will split the timing results into the initialization step and iterative step.

6.1.6.2 Settings

We run this experiment with sufficient memory available. This is set by adding "SPARK_DRIVER_MEMORY=50g" in SPARK_HOME/conf/spark-env.sh.

6.1.6.3 Expectations

Initialization step:

The limitation in our system is currently the amount of Hard Disk Drive (HDD) accesses. Apache Hadoop will have to load the input data and store the output data to Distributed File Systems (DFS), located on a HDD. Apache Spark will only have to load input data, as it can keep the results in memory. Even though loading should take more time, as writing can use a write buffer, we expect Apache Spark to be twice as fast, as it only needs to perform half the HDD accesses.

Iterative step:

Apache Hadoop will again have to load and store data into the DFS, located on the HDD. Apache Spark will have both the static and dynamic input data in memory. It does have to perform an extra join, but this join does not require a sort phase, as it is partitioned correctly. As Apache Spark has all the input data in memory, and will keep the output data in memory as well, we expect to see the promised 20x to 100x speedups for the map, sort and reduce phases.

6.1.6.4 Results

The timing results together with the associated speedups for both the initialization phase and the iterative phase are presented in Table 6.1. The timing results per iteration for Apache Spark are presented in Table 6.2.

		Hadoop	Spark	Speedup (Told/Tnew)
Initialization step	Maps phase	151	x	x
	Sort phase	86	x	x
	Reduce phase	80	x	x
	Total initialization time	317	320	1
Iterative step	Maps phase	180	2.0	90
	Sort phase + overhead	117	11.7	10
	Reduce phase	137	1.3	105
	Total iterative time	435	15.0	29

Table 6.1: Timing in seconds together with the Speedup for both the initialization step and iterative step in Apache Hadoop and Apache Spark.

	Map	Reduce	Sort + overhead	Total
Iteration 2	2.5	1.3	14.0	17.8
Iteration 3	2.3	1.3	13.1	16.6
Iteration 4	2.4	1.3	12.8	16.6
Iteration 5	1.5	1.2	9.6	12.3
Iteration 6	1.5	1.3	9.1	11.9

Table 6.2: Timing in seconds for the iterative steps in Apache Spark.

6.1.6.5 Evaluation

Graph 6.1 visualizes the data presented in Table 6.1. We can see that the time required in the initialization step is about equal for Apache Hadoop and Apache Spark. This is surprising, as we expected $time(\text{ApacheSpark}) = 0.5 * time(\text{ApacheHadoop})$. Upon further analysis, we found that only half the time in the initialization is spend on loading from HDD and creating actors pairs. The other half is spend on deleting duplicate pairs before going into the iterative phase. We have implemented this different for Apache Hadoop and Apache Spark.

We have used the default function, called "distinct," in Apache Spark. This function receives one big input set, and has to filter for duplicates. We determined the size of the input set as follows: The application input set contains 10,000 movies, with 1-100 actors, uniformly distributed. The minimal amount of pairs that can be created for one movie is zero, as 1 person does not make a pair. The maximum amount of pairs that can be created for one movie, is $100 * 100 = 10,000$, as we want (a,a), (a,b) and (b,a). On average we create 4000 pairs, as the number of pairs in a movie is not distributed evenly. This leads to a total number of pairs of $10,000 * 4000 = 4 * 10^7$. In worst case, every single pairs is compared to every other pair, creating $(4 * 10^7)^2 = 1.6 * 10^{15}$ comparisons in 150 seconds. DRAM can handle 10^8 IOPS, so some smart algorithm has been implement in the default "distinct" function in Apache Spark, but this cannot compete with the method we used in Apache Hadoop, that used knowledge about the set that the default function did not have.

In Apache Hadoop, we let the data flow through the sort phase, after which everything was sorted per key. When building this list, our reduce function checked for duplicates while building the list. For this, it only had to compare one value to the (growing) set of values belonging to one key. This sort was already required, and the comparison is made against a very limited subset, yielding orders of magnitude faster calculation.

Providing the default "distinct" function in Apache Spark with the entire dataset was a bad implementation from our side, but when we disregard this error, we see that our expectation of $time(\text{ApacheSpark}) = 0.5 * time(\text{ApacheHadoop})$ is correct. The total number of pairs after removing duplicates is about

The timing results for the iterative step are also as expected. We see an overall speedup of 29, and a per phase speedup of 90 and 105 for the map and reduce step respectively. The sort phase only has a speedup of ten, but this is also expected, as our measurements collect all the overhead in the sort phase, as explained in Section 6.1.5. This overhead becomes less as the program advances. This is clear In Graph 6.2, where we see the timing results of Apache Spark per iterative step. In the beginning of every stage, Apache Spark decides which Stage has all it's dependencies resolved, and is ready to be executed. As we continue to have fewer stages left after every stage, we see the required time go down. Based on the distribution between time in the map, sort and reduce phase in Apache Hadoop, we expect that the actual speedup of the sort phase in Apache Spark will be similar to the map and reduce phase in Apache Spark. We do see ways to verify this assumption, for instance by varying sizes of data going into the sort function, but found our results promising enough to not spend time on this and move on to the next experiment.

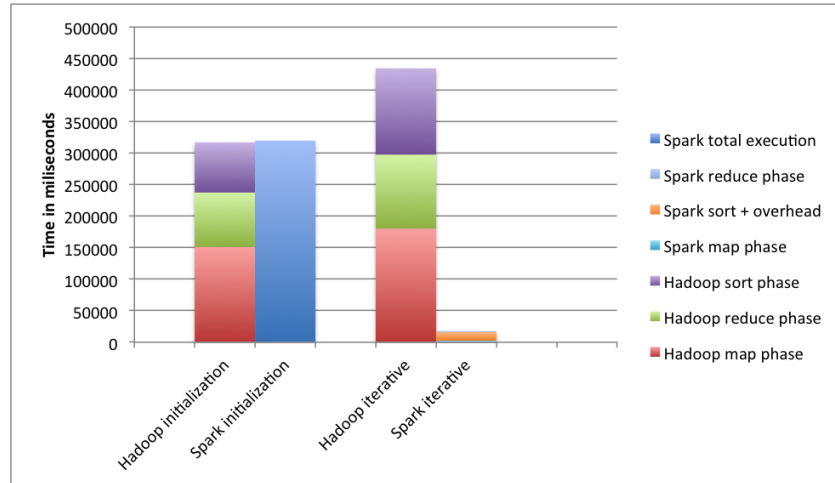


Figure 6.1: Relative performance of Apache Hadoop versus Apache Spark for both the initialization step and the iterative step, split out were possible in map, sort and reduce phase timings.

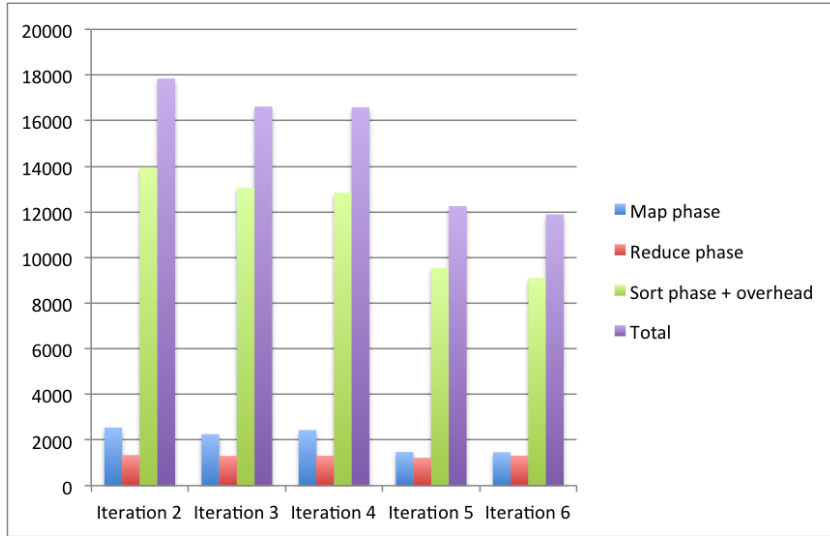


Figure 6.2: Timing of Apache Spark per iterative step.

6.1.6.6 Conclusion

The research question associated with this experiment is:

- How does Apache Spark perform relative to Apache Hadoop?

For our representative algorithm, and under sufficient memory availability, we have seen that Apache Spark performs in the same order of magnitude when input has to be loaded from HDD. For our algorithm, it achieves an overall speedup of 30 for the iterative part, and a per-phase speedup of 100 by leveraging DRAM over HDD. From this, we can conclude that Apache Spark performs equal-or-better to Apache Hadoop for iterative algorithms, when sufficient memory is available. Therefore, we believe our focus on second-generation Big Data systems is justified.

6.1.7 Experiment 1b: How does Apache Spark perform when we constrain available memory

In this experiment, we are going to look at the performance when we constrain available memory. We do this by examining the performance for a representative iterative example, the six-degrees of separation example introduced in Section 2.1.7.

6.1.7.1 Goal

The goal of this experiment is to verify our assumption that there is room for improvement in second-generation Big Data systems. If the outcome of this experiment shows that there is indeed room for improvement, we will use this information to create a proof-of-concept implementation in which we use Flash technology as medium to spill data to.

6.1.7.2 Settings

One can also choose to explicitly cache RDDs on a user level, as explained in 4.3. We have chosen to run experiments for three levels of caching:

1. No explicit caching
2. Only explicitly cache the iterative RDD
3. Cache all RDDs

For all levels, we will gradually constrain the available memory, by adding "SPARK_DRIVER_MEMORY=[x]g" in SPARK_HOME/conf/spark-env.sh, where [x] is the amount of memory granted to the program.

6.1.7.3 Expectations

We expect Apache Spark to degrade gracefully, to the point where it will perform equal to a first-generation big-data system. This expectation is based on the assumption that second-generation Big Data systems are different from first-generation Big Data systems in the way they use in-memory computation and on empirical results as discussed in Section 4.3.3. When there is no available memory, there remains no structural difference, only implementation specific differences.

We expect to see a difference for level of caching only in the experiments where limiting the memory actually causes Apache Spark to spill. The number of experiments for which this will happen, will vary per cache level. Level 1 will only use the memory budget for the storage region, which is 20%, as explained in Section 4.3. When we choose to cache RDDs, they will move to the storage region, which has a memory budget of 60%, freeing up data in the shuffle region. As this leads to less data to spill, we expect level 2 and 3 to perform faster than level 1. We also expect level 2 to perform faster than level 3, as there is only one RDD that needs to be cached. Caching all other RDDs will create one stage for every RDD. This will allow for less pipelining, as every stage is a stop-for-all synchronization.

6.1.7.4 Results

The results are presented in Table 6.3.

6.1.7.5 Evaluation

Graph 6.3 visualizes Table 6.3. Our first observation is that with no explicit RDD caching on a user level, we see a lot of variations in execution times. When using any replacement policy, some variation can be expected, but the variations seen here are larger than expected. We believe that the cause for this is that the decision to keep data in memory or spill to hard disk will be made for entire RDDs. Our intermediate RDDs are one to three Gigabytes in size. When you constrain memory to sizes less than 15 Gigabytes, placing an entire RDD in Dynamic Random-Access Memory (DRAM)

Memory size in Gigabytes	No RDDs cached	Only iterative RDD cached	All RDDs cached
1	807.93	826.8	852.93
2	952.72	416.56	
3	654.66	260.79	291.83
4	541.36	260.09	
5	670.6	406.51	427.63
6	1053.66	256.77	
7	532.13	133.31	158.23
8	506.72	112.34	
9	799.14	111.42	163.9
10	404.59	112.23	
11	486.78	97.88	122.28
12	411.75	101.06	
13	407.07	103.36	125.16
14	276.67	102.18	
15	268.71	112.34	133.71
16	268.16	108.66	
17	276.97	98.22	142.85
18	290.7	111.02	
19	252.18	110.31	151.82
20	263.8	102.57	
150	263.53	114.39	141.98

Table 6.3: Time in seconds for various memory sizes in Gigabytes. Measured with no RDDs, all RDDs and the only iterative RDD cached.

or HDD has a significant impact, due to the orders of magnitude difference in writing speeds between DRAM and HDD.

Caching also has a significant impact on performance. The difference between the three cache levels are as expected, but the number of experiments that are affected are not. Even for 150 GB, sufficient to hold all RDDs, there is a 2x performance difference. Upon further examination of the literature, we found that Spark can choose to either cache or re-compute an RDD. It seems that in our case, the iterative RDD is simply forgotten and re-computed in every iterative step, even when it has sufficient memory. Caching also has an influence on the variation in execution times for different amounts of available memory. This is logically, because we know that the iterative RDD is the best choice to keep in memory. As we have provided this information to the runtime, it will use this information to keep this RDD in memory, having to spill less data to HDD, resulting in less variation.

The optimal amount of memory when caching the iterative RDD is 7 Gigabytes for our algorithm with an input set of 40 Megabytes. This is because we create a lot

of actor pairs from a limited input set. When we start the program, we have 10,000 movies with an average of 50 actors per movie, accumulating to $10,000 \times 50 = 500,000$ objects. As we create pairs from every actor in a line to every actor in a line, we create $50 \times 50 \times 10,000 = 25,000,000$ pairs. As these are pairs instead of single objects, we have an approximate scaling factor of $(25,000,000 / 500,000) \times 2 = 100$. This makes our intermediate RDD sizes of around 3 Gigabytes understandable, as well as a optimal memory size of 7 Gigabytes.

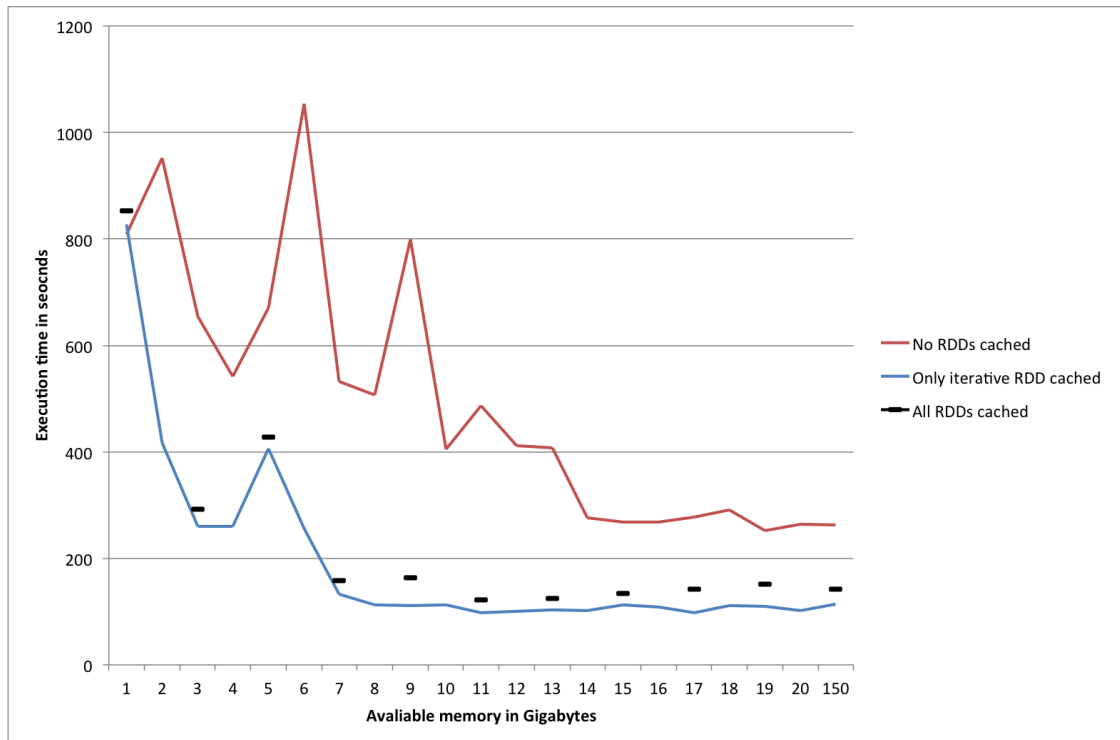


Figure 6.3: Performance degradation under constrained memory in Spark.

6.1.7.6 Conclusion

The research question associated with this experiment is:

- How does Apache Spark perform when we constrain the available memory?

Explicitly caching the iterative RDDs is crucial. When this is done, performance of Apache Spark degrades gracefully for our program. When memory is constrained, we see a performance decrease of 2x to 8x. This leaves room for improvement.

6.1.7.7 Future work

If it turns out that Flash can provide this improvement, it would be interesting to see this experiment with larger input sizes. This would show the correlation between input

size and optimal memory amount. This would also show if the exponential decrease in performance would continue until it reaches the performance of a first-generation Big Data system, and show how the optimal amount of memory increases with the input size.

6.2 Experiment 2: Can Flash technology contribute to Apache Spark

In Chapter 5, we have presented multiple possible ways in which Flash technology could contribute to Big Data systems. In experiment 2, we will implement and evaluate two of these: flash connected as file storage, as explained in Section 5.3.1, and flash connected as secondary object store, as explained in Section 5.3.2, but with the memory aggregation still present. To examine this, we will use the same software environment as we used in experiment 1, including both our implementations of the six-degrees of separation algorithm. We did use a slightly different hardware setup that we will present first, after which we will present our sub-experiments, together with the results, evaluation and conclusion.

6.2.1 Hardware

For this experiment, we have a different Power8 server with a slightly higher clock frequency, and connected the IBM FlashSystem 840.

6.2.1.1 IBM Power System S824L

The second experiment is conducted on an IBM Power System S824L [26]. The S824L consists of two 12-core 3.52 Ghz Power8 processor cards. These are 0.5 Ghz faster than the processors used in experiment 1. The system is equipped with 1600 MHz DDR3 memory with a total capacity of 256 Gigabytes. We also use only one hard disk in this system.

6.2.1.2 IBM FlashSystem 840

The IBM FlashSystem 840 [27] writes with a minimum latency of 90 us and with a maximum number of 600,000 IOPS and maximum bandwidth of 4GB/s. It can read with a minimum latency of 135 us and with a maximum number of 1,100,000 IOPS and maximum bandwidth of 8GB/s.

6.2.2 Experiment 2a: Can we improve Apache Spark by using Flash technology connected as file storage

As we have seen in Table 2.1, DRAM has a higher number of Input/Output Operations per Second (IOPS) and a higher bandwidth than Flash. Before connecting the Flash system as a file storage, we performed a sanity check. We did this by installing a file system on DRAM and created a so-called ram-disk. We will first evaluate this, as this altered our expectations for this experiment.

6.2.2.1 Goal

The goal of this experiment is to evaluate the possible use of Flash technology connected as file storage to spill intermediate data to.

6.2.2.2 Settings

We created a ram-disk by following [28]. We limited the memory to force spilling of intermediate data, by adding "SPARK_DRIVER_MEMORY=[x]g" in SPARK_HOME/conf/spark-env.sh, where [x] is the amount of memory granted to the program.

6.2.2.3 Expectations

When memory is not constrained, we expect to see a speedup in the initialization phase, where data needs to be loaded from HDD in DRAM. We also expect to see a speedup when we constrain available memory, and Apache Spark will start spilling to disk.

6.2.2.4 Results

Our results are presented in Table 6.4.

Memory size	HDD	Ram-disk	difference	Speedup (Told/Tnew)
1	762.7	764.51	-1.81	0.99763247
2	1277.16	1291.71	-14.55	0.988735862
3	2241.2	2075.58	165.62	1.079794563
4	2229.3	2179.05	50.25	1.023060508
5	690.86	691.12	-0.26	0.999623799
6	669.38	653.01	16.37	1.025068529
7	399.53	405.56	-6.03	0.98513167
8	1014.74	974.58	40.16	1.041207495
9	792.21	806.09	-13.88	0.982781079
10	388.16	368.58	19.58	1.053122796
11	2059.73	2018.58	41.15	1.020385618
12	2069.96	2012.86	57.1	1.028367596
13	1612.71	1636.89	-24.18	0.985228085
14	950.31	962.86	-12.55	0.986965914
15	1231.24	1007.2	224.04	1.222438443
16	705.21	707.04	-1.83	0.997411745
17	711.88	780.8	-68.92	0.911731557
18	700.46	708.15	-7.69	0.989140719
19	668.61	664.25	4.36	1.006563794
20	694.88	670.49	24.39	1.036376381
21	1390.01	1075.22	314.79	1.292767992
22	947.65	940.66	6.99	1.007430953
23	891.46	898.5	-7.04	0.992164719

24	789.88	792.65	-2.77	0.996505393
25	662.07	661.25	0.82	1.001240076
26	624.8	620.13	4.67	1.007530679
27	462.73	456.98	5.75	1.012582608
28	325.79	324.6	1.19	1.003666051
29	275.3	281.19	-5.89	0.979053309
30	270.93	274.65	-3.72	0.986455489
31	278.73	270.52	8.21	1.030348958
32	303.11	306.57	-3.46	0.988713834
33	326.22	309.98	16.24	1.052390477
34	284.57	284.49	0.08	1.000281205
35	282.02	277.92	4.1	1.014752447
36	277.21	276	1.21	1.004384058
37	262.55	280.72	-18.17	0.935273582
38	261.5	260.34	1.16	1.004455712
39	258.32	264.82	-6.5	0.975455026
40	256.42	261.2	-4.78	0.981699847
41	255.15	249.95	5.2	1.020804161
42	239.37	234.87	4.5	1.019159535
43	231.56	248.98	-17.42	0.930034541
44	233.19	231.01	2.18	1.009436821
45	232.39	241.71	-9.32	0.961441397
46	240.68	245.75	-5.07	0.979369278
47	244.13	248.03	-3.9	0.984276096
48	233.93	236.24	-2.31	0.990221808
49	231.03	228.85	2.18	1.00952589
50	224.79	229.38	-4.59	0.979989537
			15.513	1.010243602

Table 6.4: Timing in seconds during the iterative step for both HDD and ram-disk, together with the associated speedup.

6.2.2.5 Evaluation

A graphical representation of our results can be found in Figure 6.4. Here we can already see that there is no big difference in overall performance, and no outliers. When taking a closer look at Table 6.4, we can see that the average speedup equals 1.01.

6.2.2.6 Conclusion

If our ram-disk implementation would have had any significant effect, it should have showed an overall faster performance due to the speedup in the initialization phase, and an even larger speedup in the experiments in which Apache Spark was spilling (1-30 Gigabytes). As we only see a speedup of 1.01, we conclude that this implementation

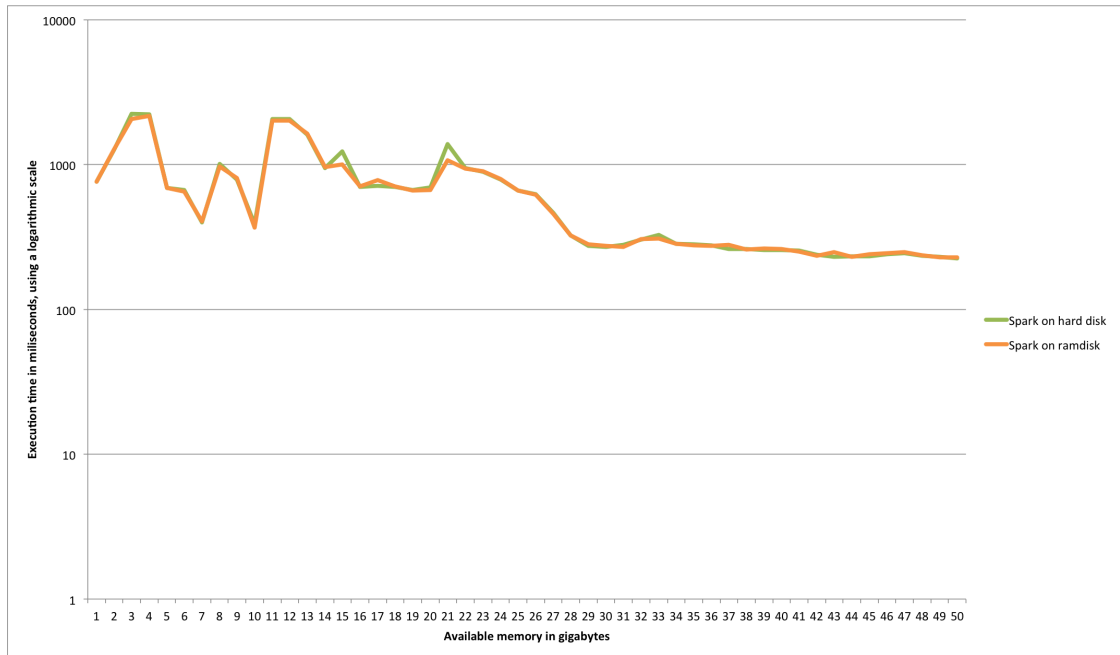


Figure 6.4: Hard disk versus ram-disk for Spark as default storage device.

with DRAM had little effect. As we explained in Table 2.1, the performance of Flash technology lays between that of a HDD and DRAM. As our file storage mounted on DRAM had little effect, we do not see much promise in creating a file storage mounted on Flash, and will shift our efforts to the next sub-experiment.

6.2.3 Experiment 2b: Can we improve Apache Spark by using Flash technology connected as secondary object store

In the previous experiment, we evaluated the use of a file system mounted on Flash for Apache Spark to spill intermediate data to. For this experiment, we have changed the Apache Spark core. Instead of spilling intermediate data to a file system, it will now write key_value pairs, as described in 5.3.2, to Flash connected using CAPI, as described in 5.2.3.2. This is a more direct way that does not involve the operating system and does not require a driver. However, data is still being aggregated before being written to Flash, as we did not make any alterations to this mechanism. We also performed a small isolated experiment to verify the writing speeds of HDD, DRAM and Flash for different block sizes.

6.2.3.1 Goal

The goal of this experiment is to evaluate the possible use of Flash technology connected as an object store using CAPI Flash to spill intermediate data to.

6.2.3.2 Settings

In order to support key_value stores from within Apache Spark, we identified the lines of code that were involved in spilling data. These can be found in Appendix B. We then received help from Jan Rellermeyer, to place hooks to connect our code for the FlashSystem into. The files he created can be found in Appendix A.3. The code for the FlashSystem can be found in Appendix A.4. We limited the memory to force spilling of intermediate data, by adding "SPARK_DRIVER_MEMORY=3g" in SPARK_HOME/conf/spark-env.sh. We changed the amount of cores to be used from the default amount, 4 cores, to single core, to better analyze the behavior. As this slowed down execution significantly, we limited the amount of iterations in the xDegreesOfSeparation to one. The code for our writing speed test can be found in Appendix A.5.

6.2.3.3 Expectations

As the data transfer does not involve the operating system, does not require a driver and makes use of Flash rather than HDD, we expect to see a noticeable speedup.

6.2.3.4 Results

When we do not limit the available amount of memory, the total execution time is 208 seconds. When we limit the available amount of memory in the original program, we have a total execution time of 262 seconds. When we use our modification of Apache Spark together with CAPI Flash, we have a total execution time of 225 seconds. Writing to HDD takes a total of 329ms. Writing the same data to DRAM takes 53ms. Writing this to DRAM using the API takes 130ms.

Writing speeds with variable block sizes for HDD, DRAM and Flash can be found in Table 6.5.

Size in bytes	DRAM	Flash	HDD
10	3.6	933.3	113.6
100	3.7	932.4	114.6
1000	3.5	927.2	114.9
10000	4.3	913.6	165.9
100000	14.6	719.3	244.7
1000000	75.7	427.9	573.1
10000000	675.8	4413.1	2982.2
100000000	9522.3	32360.7	28764.5
1000000000	103636.9	303727.5	276904.6

Table 6.5: Writing speeds in microseconds for DRAM, Flash and HDD for various block sizes.

6.2.3.5 Evaluation

Using Flash as a key_value store removed $(OldOverhead - NewOverhead)/OldOverhead * 100\% = ((262 - 208) - (225 - 208))/(262 - 208) * 100\% = 69\%$ of the time penalty for spilling. This results in a noticeable overall speedup of $T_{old}/T_{new} = 262/225 = 1.16$. This is as we expected, and we believe originates from a lack of involvement of the Operating System (OS) and driver. We believe that the remaining difference of $225 - 208 = 15$ second is mainly caused by data aggregation.

We also measured individual writing speeds for Flash, which are presented in Table 6.5. Here it appears that Flash is slower than HDD. We believe this is due to the implementation of a write buffer for HDD. We leave further investigation regarding this to future work, as the time required for actual writing of the data is orders of magnitude smaller than the total overhead, making it a lower priority. We do want to note that the FlashSystem is also capable of writing and retrieving data asynchronously, allowing easy implementation of such a buffer.

As an extra observation, we would like to point out that in our experiment, requiring 100 individual spills, all store and loads of unique keys were performed sequentially and only once. In other settings, we have noticed around 90% of the stores and loads are sequential and also only written and read once. If our focus did not lay on the evaluation of Flash technology versus HDD and DRAM, but on developing a faster spill mechanism, we would have had a different approach. The easiest approach would have been to keep the last stored key_value pair in memory after storing it in Flash, and check in this location first when retrieving data. A better implementation would focus on deleting keys in Apache Spark as soon as they are used and not needed, rather than at the end of the stage. We believe this would eliminate a lot of spilling, eliminating the need to spill data for these input sizes all together.

6.2.3.6 Conclusion

Storing data in key_value pairs to Flash, without invoking the operating system or using a driver significantly lowers the amount of overhead induced by spilling by 69% for our representative algorithm, resulting in an overall speedup of 1.16. We believe that the remaining overhead can largely be attributed to data aggregation. As we are using key_value pairs, we can store smaller amounts of data, requiring less data aggregation. We will focus on this in more detail in our future work. When optimizing for Apache Spark, instead of evaluating Flash, large benefits could be found by managing memory more efficiently, as 90% of intermediate data blocks are stored and retrieved once sequentially, but only get de-allocated at the end of a stage.

6.2.3.7 Future Work

In our future work, we are first going to work on removing the overhead from aggregation. We also want to take more accurate measurement of the writing speeds of HDD and the influence of a write buffer.

Conclusion

This chapter concludes this thesis by summarizing the contributions of this thesis, the answer to the research questions and describing possible future work.

In this thesis, we have first presented the reader with a theoretical evaluation of the two different generations of Big Data systems, after which we evaluated two specific implementations: Apache Hadoop for a first-generation Big Data system and Apache Spark for a second-generation Big Data system, with as major difference the in-memory computation of the latter, while still remaining resilient, scalable, and easy to use. We have then evaluated Flash technology, and the way this can be connected in a Power8 system, and can be leveraged by Apache Spark. After the theoretical evaluations, we have used this information in order to perform two experiments, both with two sub-experiments, in order to answer our research questions.

Our first question, "How does a second-generation Big Data system perform," questions if there is a need for improvement. We have split this up in two sub-questions. With "How does a second-generation Big Data system perform relative to a first-generation Big Data system," we would like to justify our focus on second-generation Big Data systems. For this, we took a representative iterative algorithm, and implemented this algorithm both in Apache Spark and Apache Hadoop on the same platform. Here we could clearly see the difference in number of Hard Disk Drive (HDD) accesses, and how this affected performance. In the initialization step of our program, Apache Spark only needed to load data from HDD, whereas Apache Hadoop also needed to write back the results to HDD. Because of this, Apache Spark was twice as fast. During each iterative step, Apache Spark was able to keep all data in-memory, whereas Apache Hadoop loaded, and saved the intermediate results to HDD between iterative steps. Because of this, we measured an overall speedup of 30 for our algorithm, and when looking at the individual map, sort and reduce phases, a per-phase speedup of 100. From this, we concluded that Apache Spark, for our representative iterative algorithm, performed equal-or-better than Apache Hadoop. For us, this justified our initial focus on second-generation Big Data systems.

As the number of HDD accesses greatly impact performs, we expected room for improvement in second-generation Big Data systems in phases where the HDD is still needed. For this, we asked "How does a second-generation Big Data system perform when we constrain available memory." In our experiment, we re-used our implementation for Apache Spark of the representative algorithm, and run this multiple times, while gradually constraining available memory. Here we could see that performance slowly degrades to more than an order of magnitude difference. From this, we concluded that

there is still room for improvement.

Our first attempt to provide this improvement with Flash technology was by investigating the possible benefit of mounting a file storage on Flash to store the Distributed File Systems (DFS) and use as a location to spill intermediate data towards. This corresponds to research question 2a: "Can we improve Apache Spark by using Flash technology connected as file storage?" We first performed a sanity check using a ram-disk, a faster, but more expensive and generally more limited in capacity alternative. When using this ram-disk, we measured no significant speedup. We believe that in our case, the amount of overhead generated for aggregating data and using a file system and the associated system and driver calls is orders of magnitude larger than the performance difference of using a different storage media. As DRAM did not provide any improvement, we reasoned that the same method using Flash would also not provide a performance improvement, and shifted our focus on the next sub-experiment.

For our final experiment, we used CAPI Flash as a replacement of the HDD in order to spill intermediate data to. This corresponds to research question 2b: "Can we improve Apache Spark by using Flash technology connected as secondary object store in the RDD storage region?"¹ We changed the Apache Spark code in order to facilitate this. By doing so, we eliminated the need for system calls and drivers, and removed 69% of the overhead for spilling data in our representative algorithm. This resulted in an overall speedup of 1.16. We believe that the remaining overhead is mainly in the data aggregation. As we are using key_value pairs, we can store smaller amounts of data, requiring less data aggregation. We will investigate this in more detail in future work.

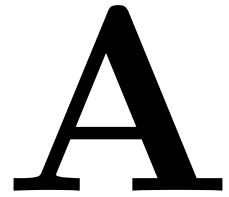
Based on our measurements, we believe that Apache Spark, and Big Data systems in general, can benefit from Flash technology connected as object store as it can remove overhead from data aggregation and has faster writing speeds than HDD, and especially CAPI Flash, as it also removes overhead from Operating System (OS) involvement. When we show that we can indeed also remove the overhead from data aggregation, thereby further reducing overhead of spilling, we can use Apache Spark, and Big Data systems in general, on a system with limited DRAM and a large amount of Flash, resulting in performance equal to in-memory computation, but with an orders of magnitude larger storage capacity, allowing to solve bigger problems with lower costs.

¹Due to time constraints we left the possibility of using Flash in the shuffle region as future work

Bibliography

- [1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Proceedings of Operating Systems Design and Implementation (OSDI)*, vol. 6, pp. 137–150, 2004.
- [2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*. USENIX Association, 2012, pp. 2–2.
- [3] L. C. T. Keh. (2014, Jul.) Apache spark in resource constrained states. [Online]. Available: <http://tech.blog.box.com/2014/07/apache-spark-in-resource-constrained-states/>
- [4] R. Xin. (2014, Nov.) Spark officially sets a new record in large-scale sorting. [Online]. Available: <https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html>
- [5] S. Milgram, “The small world problem,” *Psychology today*, vol. 2, no. 1, pp. 60–67, 1967.
- [6] SixDegrees.org. Sixdegrees. [Online]. Available: <http://www.sixdegrees.org/>
- [7] K. Bacon. Kevin bacon explains ”six degrees” story. [Online]. Available: <http://www.nbcchicago.com/video/#!/news/business/Kevin-Bacon-Explains-Six-Degrees-Story/131838958>
- [8] Omdb, open media database. [Online]. Available: <http://www.omdb.org/>
- [9] Creative commons license. [Online]. Available: <http://creativecommons.org/>
- [10] Solr is the popular, blazing-fast, open source enterprise search platform built on apache lucene. [Online]. Available: <http://lucene.apache.org/solr/>
- [11] D. L. Brinker and W. L. Bain, “Accelerating hadoop mapreduce using an in-memory data grid.” [Online]. Available: <http://www.scaleoutsoftware.com/downloads-resources/white-papers-videos/>
- [12] T. A. S. Foundation. Hadoop. [Online]. Available: <http://hadoop.apache.org/>
- [13] ——. Spark, lightning-fast cluster computing. [Online]. Available: <https://spark.apache.org/>
- [14] Hadoop wiki. [Online]. Available: <http://wiki.apache.org/hadoop/PoweredBy>
- [15] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

- [16] Hdfs users guide. [Online]. Available: https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html#Secondary_NameNode
- [17] Spark programming guide. [Online]. Available: <https://spark.apache.org/docs/latest/programming-guide.html>
- [18] Spark configuration guide. [Online]. Available: <https://spark.apache.org/docs/latest/configuration.html>
- [19] Documentation on javapairrdd. [Online]. Available: <http://people.apache.org/~pwendell/spark-1.1.0-rc2-docs/api/java/org/apache/spark/api/java/JavaPairRDD.html>
- [20] Getting started with apache spark and cassandra. [Online]. Available: http://planetcassandra.org/getting-started-with-apache-spark-and-cassandra/?gclid=Cj0KEQIA7tCjBRDulMny5rfM0dkBEiQA7feshZqlObGoDBcsS-R3Ip3Rr_qYdbgusDv8xUsOBiGzm5saAh0J8P8HAQ
- [21] G. DeCandia, D. Hastorun, M. Jampani, G. vardhan Kakulapati, A. Lakshman, A. Pilchin, S. inathan Sivasubramanian, P. Voshall, and W. Vogels, “Dynamo: Amazon?s highly available key-value store.”
- [22] A. Caldeira, B. Grabowski, V. Haug, M. E. Kahle, C. D. Maciel, and M. Sanchez, *IBM Power Systems S814 and S824 Technical Overview and Introduction*. IBM Redbooks, 2014. [Online]. Available: <http://www.redbooks.ibm.com/redpapers/pdfs/redp5102.pdf>
- [23] Ubuntu 14.04 for power8. [Online]. Available: <http://www.ubuntu.com/download/server/power8>
- [24] Apache hadoop 2.2.0 mirror. [Online]. Available: <http://www.apache.org/dyn/closer.cgi/hadoop/common/>
- [25] Spark 1.1 source code. [Online]. Available: <https://spark.apache.org/downloads.html>
- [26] “Tbm power system s824l.” [Online]. Available: <http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/redp5139.html?Open>
- [27] “Tbm flashsystem 840 product guide.” [Online]. Available: <http://www.redbooks.ibm.com/technotes/tips1079.pdf>
- [28] Linux ramdisk mini-howto. [Online]. Available: <http://www.vanemery.com/Linux/Ramdisk/ramdisk.html>



The first two appendixes contain our implementation of the six degrees of separation problem in Apache Hadoop and Apache Spark respectively. This is followed by two appendixes of code that can be added to Apache Spark in order to facilitate the spilling of intermediate data towards the CAPI Flash. The last appendix contains code used to measure writing speeds for HDD, DRAM and CAPI Flash.

A.1 xDegreesOfSeparation for Apache Hadoop

```
/*
 * xDegreesOfSeparation
 *
 * Implementation of the six degrees of separation from Kevin
 * Bacon problem for Apache hadoop
 *
 * Author: Tom Hubregtsen
 * Email: tom@hubregtsen.com
 */

import java.lang.System.*;
import java.util.LinkedList;
import java.util.List;
import java.util.StringTokenizer;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;
import java.lang.Object;
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.BytesWritable;
```

```

import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.
    KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.jobcontrol.JobControl;
import org.apache.hadoop.util.GenericOptionsParser;

public class xDegreesOfSeparation
{
    public static boolean debug = false;

    public static class IntArrayWritable extends ArrayWritable
    {
        public IntArrayWritable() { super(IntWritable.class); }
    }

    /*
     * Goal: Create <actor, collaborating actors> pairs
     * Input: Text, Text (All actors in a movie, empty)
     * Output: Multiple IntWritable, IntWritable (key(actor),
        value(collaborating actor))
     */
    public static class initMapper extends Mapper<Text, Text,
        IntWritable, IntWritable>
    {
        long total = 0;
        long run = 0;
        long startTime;
        long stopTime;
        long mapperStartTime = System.currentTimeMillis();
        public void map(Text key, Text value, Context context)
            throws IOException, InterruptedException
        {
            startTime = System.currentTimeMillis();

```

```

run++;

List idList = Arrays.asList(key.toString().split(","));
//Now create the pairs
//We want both (a,b) and (b,a), to assign both a distance
    in the next function
//And (a,a) to remain state
for (int i = 0; i < idList.size(); i++)
{
    for (int j = 0; j < idList.size(); j++)
    {
        context.write(new IntWritable( Integer.parseInt((
            String) idList.get(i)) ), new IntWritable(
            Integer.parseInt((String)idList.get(j))));
    }
}
stopTime = System.currentTimeMillis();
total += stopTime - startTime;
if (9999 < run)
    System.out.print(" Initialization mapper start time:" +
        mapperStartTime + "\nInitialization mapper stop time
        : " + System.currentTimeMillis() + "\n");
}
}

/*
 * Reduce all pairs to // <actor, distance#collaboratingActor1
    -collaboratingActor2 -..>
 * Input: IntWritable, IntWritable (key(actor), value(
    collaborating actor))
 * Output: Text, Text (key(actor), value (encoded nested
    key_value pair(key(initial distance),value(list of
    collaborating actors)))
 */
public static class initReducer extends Reducer<IntWritable,
    IntWritable, Text, Text>
{
    int starter = 0;
    long total = 0;
    long run = 0;
    long startTime;
    long stopTime;
    long mapperStartTime = System.currentTimeMillis();
    public void reduce(IntWritable key, Iterable<IntWritable>
        values, Context context) throws IOException,

```

```

        InterruptedException
    {
        startTime = System.currentTimeMillis();
        run++;
        //Copy values into one list
        List<Integer> valueList = new ArrayList<Integer>();
        for (IntWritable i: values)
        {
            valueList.add(i.get());
        }
        int keyString = key.get();
        int kevinBeacon = 4724;
        int target = kevinBeacon;
        String valueString;
        //Init others to infinity
        if (keyString != target)
            valueString = "999999#";
        //Init Kevin to 0
        else
            valueString = "0#";
        //First ActorCollaboration does not need a comma
        if (valueList.size() > 0)
            valueString += valueList.get(0).toString();
        int i = 0;
        //Rest do
        for (i = 1; i < valueList.size(); i++)
        {
            if (true == debug)
                System.out.print("K/V: " + keyString + "+" +
                    valueList.get(i).toString() + "\n");
            valueString += "," + valueList.get(i).toString();
        }
        //Write results, ready for HDFS
        stopTime = System.currentTimeMillis();
        total += stopTime - startTime;
        context.write(new Text(Integer.toString(keyString)), new
            Text(valueString));

        //Catch last run (in our input file there are 99387
            unique ID's used)
        if(run == 99387)
            System.out.print(" Initialization reducer start time:" +
                mapperStartTime + "\nInitialization reducer stop
                    time:" + System.currentTimeMillis() + "\n");
    }

```

```

    }

/*
 * Goal: create pairs <Collaborating actor <distance, actor>>
 * Input: Text, Text (key(actor), value (encoded nested
        key_value pair(key(initial distance),value(list of
        collaborating actors)))
 * Output: IntWritable, IntArrayWritable (key(actor), nested
        value(key(distance),value(collaborating actor)))
 */
public static class IterativeMapper extends Mapper<Text, Text
    , IntWritable, IntArrayWritable>
{
    int starter = 0;
    long total = 0;
    long run = 0;
    long startTime;
    long stopTime;
    long mapperStartTime = System.currentTimeMillis();
    public void map(Text key, Text value, Context context)
        throws IOException, InterruptedException
    {
        startTime = System.currentTimeMillis();
        run++;

        //Get data
        int actor = Integer.parseInt(key.toString());
        List initialSplit = Arrays.asList(value.toString().split
            ("#"));
        int distance = Integer.parseInt((String) initialSplit.get
            (0));
        List collaboratingActors = Arrays.asList( ((String)
            initialSplit.get(1)).split(",") );

        // Write new distances in
        // <Collaborating actor <distance, actor>>
        // actor is needed to preserve state
        int i=0;
        for ( i = 0; i < collaboratingActors.size(); i++)
        {
            int currentActor = Integer.parseInt((String)
                collaboratingActors.get(i));
            // Set collaboratingActor
            IntWritable newKey = new IntWritable(currentActor);
            IntWritable[] inner = new IntWritable[2];

```

```

        // Set distance
        if(actor != currentActor)
            inner[0] = new IntWritable(distance+1);
        else
            inner[0] = new IntWritable(distance+0);
        // Set actor
        inner[1] = new IntWritable(actor);
        IntArrayWritable classer = new IntArrayWritable();
        classer.set(inner);
        context.write(newKey, classer);
        stopTime = System.currentTimeMillis();
        total += stopTime - startTime;
    }
    //System.out.print("run: " + run + "\n");
    if(run == 99387)
        System.out.print("Iterative mapper start time: " +
            mapperStartTime + "\nIterative mapper stop time:" +
            System.currentTimeMillis() + "\n");
    }
}

/*
 * Goal: Find minimal distance , remain state
 * Input: IntWritable , IntArrayWritable (key(actor), nested
        value(key(distance),value(collaborating actor)))
 * Output: Text, Text (key(actor), value (encoded nested
        key_value pair(key(initial distance),value(list of
        collaborating actors)))
 */
public static class IterativeReducer extends Reducer<
    IntWritable ,IntArrayWritable ,Text ,Text>
{
    long total = 0;
    long run = 0;
    long startTime;
    long stopTime;
    long mapperStartTime = System.currentTimeMillis();
    public void reduce(IntWritable key, Iterable<
        IntArrayWritable> values , Context context) throws
        IOException , InterruptedException
    {
        startTime = System.currentTimeMillis();
        run++;
    }
}

```

```

    int keyActor = key.get();
    int collaboratingActor;
    int distance;
    int minDistance = 999999;

    String collaboratingActors = "";
    List<IntArrayWritable> valueList = new ArrayList<
        IntArrayWritable>();
    //Find minimal distance
    for (IntArrayWritable i: values)
    {
        IntWritable[] inner = (IntWritable[]) i.toArray();
        distance = inner[0].get();
        if (distance < minDistance)
            minDistance = distance;
        collaboratingActor = inner[1].get();
        if (collaboratingActors.equals("") == false)
            collaboratingActors += "," + collaboratingActor;
        else
            collaboratingActors += collaboratingActor;
    }
    //Write minimal distance and collaborating actors
    context.write(new Text(Integer.toString(keyActor)), new
        Text(Integer.toString(minDistance) + "#" +
            collaboratingActors));
    stopTime = System.currentTimeMillis();
    total += stopTime - startTime;
    if(run == 99387)
        System.out.print(" Iterative reducer start time:" +
            mapperStartTime + "\nIterative reducer stop time:" +
            System.currentTimeMillis() + "\n");
    }
}

// Same as iterativeMapper, but able to forget collaborating
// actor
public static class FinalMapper extends Mapper<Text, Text,
    IntWritable, IntArrayWritable>
{
    int starter = 0;
    long total = 0;
    long run = 0;
    long startTime;
    long stopTime;

```

```

long mapperStartTime = System.currentTimeMillis();
public void map(Text key, Text value, Context context)
    throws IOException, InterruptedException
{
    startTime = System.currentTimeMillis();
    run++;
    int actor = Integer.parseInt(key.toString());
    List initialSplit = Arrays.asList(value.toString().split
        ("#"));
    int distance = Integer.parseInt((String) initialSplit.get
        (0));
    List collaboratingActors = Arrays.asList( ((String)
        initialSplit.get(1)).split(",") );
    int i=0;
    for ( i = 0; i < collaboratingActors.size(); i++)
    {
        int currentActor = Integer.parseInt((String)
            collaboratingActors.get(i));
        IntWritable newKey = new IntWritable(currentActor);
        IntWritable[] inner = new IntWritable[2];
        if(actor != currentActor)
            inner[0] = new IntWritable(distance+1);
        else
            inner[0] = new IntWritable(distance+0);
        inner[1] = new IntWritable(actor);
        IntArrayWritable classer = new IntArrayWritable();
        classer.set(inner);
        context.write(newKey, classer);
    }
    stopTime = System.currentTimeMillis();
    total += stopTime - startTime;
}
if(run == 99387)
    System.out.print(" Final mapper start time:" +
        mapperStartTime + "\nFinal mapper stop time:" +
        System.currentTimeMillis() + "\n");
}
}

//Same as iterativeReducer, but without the collaborating
actor
public static class FinalReducer extends Reducer<IntWritable,
    IntArrayWritable, Text, Text>
{
    long total = 0;

```

```

long run = 0;
long startTime;
long stopTime;
long mapperStartTime = System.currentTimeMillis();
public void reduce(IntWritable key, Iterable<
    IntArrayWritable> values, Context context) throws
    IOException, InterruptedException
{
    startTime = System.currentTimeMillis();
    run++;

    int keyActor = key.get();
    int collaboratingActor;
    int distance;
    int minDistance = 999999;
    String collaboratingActors = "";
    List<IntArrayWritable> valueList = new ArrayList<
        IntArrayWritable>();
    for (IntArrayWritable i: values)
    {
        IntWritable[] inner = (IntWritable[]) i.toArray();
        distance = inner[0].get();
        if (distance < minDistance)
            minDistance = distance;
        collaboratingActor = inner[1].get();
/*
        // This can now be skipped, in order to save data
        if (collaboratingActors.equals("")) == false)
            collaboratingActors += "," + collaboratingActor;
        else
            collaboratingActors += collaboratingActor;
*/
    }
    context.write(new Text(Integer.toString(keyActor)), new
        Text(Integer.toString(minDistance)));

    stopTime = System.currentTimeMillis();
    total += stopTime - startTime;
    if(run == 99387)
        System.out.print("Final reducer start time:" +
            mapperStartTime + "\nFinal reducer stop time:" +
            System.currentTimeMillis() + "\n");
}
}

```

```

public static void main(String[] args) throws Exception
{
    System.out.print("Program started..\n");
    Configuration conf;
    Job job;

    System.out.print("Program start time:" + System.
currentTimeMillis() + "\n");
    // Initialisation
    if (debug == true)
        System.out.print("\n***** Start initialisation *****\n\n
");
    conf = new Configuration();
    job = new Job(conf, "Initialisation");
    //job.setNumMapTasks(100);
    //job.setNumReduceTasks(100);
    job.setJarByClass(xDegreesOfSeparation.class);
    job.setMapperClass(initMapper.class);
    job.setReducerClass(initReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(IntWritable.class);
    //job.setInputFormatClass(FileInputFormat.class);
    job.setInputFormatClass(KeyValueTextInputFormat.class);

    //FileInputFormat.addInputPath(job, new Path("/user/tom/
generatedDB/1000004636.txt"));
    FileInputFormat.addInputPath(job, new Path("/user/tom/
generatedFile"));
    FileOutputFormat.setOutputPath(job, new Path("output0"));
    job.waitForCompletion(true);
    // Map/Reduce 1 -> n-1
    int i = 0;
    int iterations = 6;
    for (i = 0; i < (iterations - 1); i++)
    {
        if (debug == true)
            System.out.print("\n***** Start Map/Reduce Stage " + i
+ " *****\n\n");
        conf = new Configuration();
        job = new Job(conf, "Iterative" + i + "");
        //job.setNumMapTasks(100);
        //job.setNumReduceTasks(100);
        job.setJarByClass(xDegreesOfSeparation.class);
        job.setMapperClass(IterativeMapper.class);
        job.setReducerClass(IterativeReducer.class);

```

```

        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntArrayWritable.class);
        job.setInputFormatClass(KeyValueTextInputFormat.class);
        FileInputFormat.addInputPath(job, new Path("/user/tom/
            output" + i));
        FileOutputFormat.setOutputPath(job, new Path("output" + (
            i + 1)));
        job.waitForCompletion(true);
    }

    //Map/Reduce n (seperate in order to create proper output)
    if (debug == true)
        System.out.print("\n***** Start Map/Reduce Stage " + (
            iterations - 1) + " *****\n\n");
    conf = new Configuration();
    job = new Job(conf, "Iterative" + (iterations - 1) + "");
    //job.setNumMapTasks(100);
    //job.setNumReduceTasks(100);
    job.setJarByClass(xDegreesOfSeparation.class);
    job.setMapperClass(FinalMapper.class);
    job.setReducerClass(FinalReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(IntArrayWritable.class);
    job.setInputFormatClass(KeyValueTextInputFormat.class);
    FileInputFormat.addInputPath(job, new Path("/user/tom/
        output" + (iterations - 1)));
    FileOutputFormat.setOutputPath(job, new Path("output" + (
        iterations - 1 + 1)));
    job.waitForCompletion(true);
    System.out.print("Program end time:" + System.currentTimeMillis
        () + "\n");
    System.out.print("\nProgram finished\n");
}
}

```

A.2 xDegreesOfSeparation for Apache Spark

```

/*
 * xDegreesOfSeparation
 *
 * Implementation of the six degrees of separation from Kevin
 * Bacon problem for Apache Spark
 *
 * Author: Tom Hubregtsen
 * Email: tom@hubregtsen.com

```

```

*
*/

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.Function3;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.PairFlatMapFunction;
import org.apache.spark.storage.StorageLevel;

import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

import scala.Tuple2;

public class xDegreesOfSeparation {

    /*
    * Goal: Transform one input file into (actor, actor) pairs
    * Input: Tuple2<String, String> (file name, entire file
        content)
    * Output: Iterable list containing Tuple2<String, String> (
        actor, actor)
    */
    static class initClassCreateActorPairsFromWeb implements
        PairFlatMapFunction<String, Integer, Integer>
    {
        @Override
        public Iterable<Tuple2<Integer, Integer>> call(String line)
        {
            List idList = Arrays.asList(line.split(","));
            //Create the pairs
            List<Tuple2<Integer, Integer>> results = new ArrayList<
                Tuple2<Integer, Integer>>();
            Tuple2 kvPair;
            //We want both (a,b) and (b,a), to assign both a distance
                in the next function
            for (int i = 0; i < idList.size(); i++)
            {
                for (int j = 0; j < idList.size(); j++)

```

```

        {
            kvPair = new Tuple2<Integer, Integer>(Integer.
                parseInt((String) idList.get(i)), Integer.parseInt
                ((String) idList.get(j)));
            results.add(kvPair);
        }
    }
    return results;
}
}

/*
 * Goal: Initialize the distances, 0 for target, infinity for
       others
 * Input: Integer, List<Integer> (Actor, collaborations)
 * Output: Integer, Integer (Actor, Distance)
 */
static class initClassSetDistances implements PairFunction<
    Tuple2<Integer, Iterable<Integer>>, Integer, Integer>
{
    @Override
    public Tuple2<Integer, Integer> call(Tuple2<Integer,
        Iterable<Integer>> tuple)
    {
        // Get the key
        Integer key = (Integer) tuple._1;
        // Initialize distance to infinity
        Integer distance = 999999;
        Integer kevinBacon = 4724;
        //If you prefer a different target
        Integer target = kevinBacon;

        //Is this the target?-> 0, else infinity
        //If you use an Integer in the comparison, be sure to use
        something like .equals()!
        //keys == target is once in dataset, so code for common
        case
        if (key.equals(target) == false)
        {
            distance = 999999;
        }
        else
        {
            distance = 0;
        }
    }
}

```

```

        //No need to look for duplicates, thanks to distinct() in
        main()
        Tuple2 result = new Tuple2<Integer, Integer>(key,
            distance);

        return result;
    }
}

/*
 * Goal: Map phase. Assign new distance to actors, based on
        collaborations
 * Input: Integer, (Integer, List<Integer>) (actor, (distance,
        collaborations))
 * Output: Integer, Integer (actor, distance)
 */
static class mapClass implements PairFlatMapFunction<Tuple2<
    Integer, Tuple2<Integer, Iterable<Integer>>>, Integer,
    Integer>
{
    @Override
    public Iterable<Tuple2<Integer, Integer>> call(Tuple2<
        Integer, Tuple2<Integer, Iterable<Integer>>> inputKV)
    {
        Integer keyActor = (Integer) inputKV._1;
        Integer distance = (Integer) inputKV._2._1;
        List<Tuple2<Integer, Integer>> results = new ArrayList<
            Tuple2<Integer, Integer>>();
        // Copy all distances from collaborations and create a KV
        pair of (collaboratingActor, his new distance)
        for(Integer i: inputKV._2._2)
        {
            // if collaborating actor is self, remain state
            if (keyActor.equals(i))
                results.add(new Tuple2<Integer, Integer>(i, distance
                    + 0));
            // Set new distance for others
            else
                results.add(new Tuple2<Integer, Integer>(i, distance
                    + 1));
        }
        return results;
    }
}
}

```

```

/* Goal: Reduce all dinstances to find the minimum distance
   to target
   * Input: Integer, Integer (distance, distance)
   * Output: Integer (mininal distance)
   */
static class reduceClass implements Function2<Integer,
    Integer, Integer>
{
    @Override
    public Integer call(Integer valueA, Integer valueB)
    {
        Integer result = 999999;
        // Had problems with autoboxing
        if (valueA.intValue() < valueB.intValue())
            result = valueA;
        else
            result = valueB;
        return result;
    }
}

public static void main(String[] args)
{
    System.out.print("Program start time:" + System.
        currentTimeMillis() + "\n");
    // Do you want to turn on debug message?
    final boolean debug = false;
    //Entire dataset, or single?
    /*
     * Are you working on the entire database,
     * or a single file?
     * true = single file
     * false = entire database
     */
    final boolean single = false;
    /*
     * Choose a cache level
     * 0 = none
     * 1 = only iterative RDDs
     * 2 = RDDs connected to stages
     * 3 = all RDDs
     */
    final int cacheLevel = 1;

```

```

System.out.print(" Starting execution with the following
    settings:\n");
System.out.print(" Debug messages: " + debug + "\n");
System.out.print(" Executing just a single file: " + single
    + "\n");
System.out.print(" Cache level[0-3]: " + cacheLevel + "\n");

//Set the configuration and context
SparkConf conf = new SparkConf().setAppName("
    xDegreesOfSeparation");
JavaSparkContext sc = new JavaSparkContext(conf);

/*
//Load the movie data from HDFS
JavaPairRDD<String , String> wholeFiles;
if (false == single)
    wholeFiles = sc.wholeTextFiles(" hdfs://localhost:9000/
        user/tom/generatedDB/");
else
    wholeFiles = sc.wholeTextFiles(" file:/Users/rjan/projects
        /spark2/xDegreesOfSeparation/10000movies.txt");
//Single text file , containing Kevin Bacon, use for
    smaller tests
//wholeFiles = sc.wholeTextFiles(" hdfs://localhost:9000/
    user/tom/generatedDB/1000004636.txt");

if (3 <= cacheLevel)
    wholeFiles.persist(StorageLevel.MEMORY_ONLY());
*/

//Load data from file
JavaRDD<String> file = sc.textFile(" file:/home/tom/
    jan_async/xDegreesOfSeparation/10000movies.txt");
    System.out.print("We stopted in program: " + System.
        currentTimeMillis() + "\n");
if (3 <= cacheLevel)
    file.persist(StorageLevel.MEMORY_ONLY());

//Create <actor ,collaboratingActor> pairs from input file
//Spark Stage @ flatMapToPair
JavaPairRDD<Integer , Integer> actorCollaborationPairs =
    file.flatMapToPair(new initClassCreateActorPairsFromWeb
        ());

if (true == debug)

```

```

    System.out.print(" Class: initClassCreateActorPairsFromWeb
        : \n" + actorCollaborationPairs.collect() + "\n");
if (2 <= cacheLevel)
    actorCollaborationPairs.persist(StorageLevel.MEMORY_ONLY
        ());

//Reduce to <actor, Iterable<colloboratingActor>>
JavaPairRDD<Integer, Iterable<Integer>>
    actorCollaborationsAsList = actorCollaborationPairs.
        groupByKey();
if (true == debug)
    System.out.print(" Class: actorCollaborationsAsList: \n" +
        actorCollaborationsAsList.collect() + "\n");
if (1 <= cacheLevel)
    actorCollaborationsAsList.persist(StorageLevel.
        MEMORY_ONLY());

//Initialize the distances, by creating <actor,distance>
JavaPairRDD<Integer, Integer> actorDistancesBeforeDistinct
    = actorCollaborationsAsList.mapToPair(new
        initClassSetDistances());
if (true == debug)
    System.out.print(" Class: initClassSetDistances: \n" +
        actorDistancesBeforeDistinct.collect() + "\n");
if (3 <= cacheLevel)
    actorDistancesBeforeDistinct.persist(StorageLevel.
        MEMORY_ONLY());

//Remove the duplicates
//Spark double Stage @ distinct
JavaPairRDD<Integer, Integer> actorDistances =
    actorDistancesBeforeDistinct.distinct();
if (2 <= cacheLevel)
    actorDistances.persist(StorageLevel.MEMORY_ONLY());

//Because I am not sure if the interpreter can pipeline
    loops efficiently, I do it by hand
//Also handier to link log output to actual lines, and to
    cache
//Iteration 1
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations1 = actorDistances.join

```

```

        (actorCollaborationsAsList);
    if (3 <= cacheLevel)
        joinedDistanceAndCollaborations1.persist(StorageLevel.MEMORY_ONLY());
    if (true == debug)
        System.out.print(" Class: mapClass: \n" +
            joinedDistanceAndCollaborations1.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map1 =
    joinedDistanceAndCollaborations1.flatMapToPair(new
        mapClass());
    if (2 <= cacheLevel)
        map1.persist(StorageLevel.MEMORY_ONLY());
    if (true == debug)
        System.out.print(" Class: mapClass: \n" + map1.collect()
            + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce1 = map1.reduceByKey(
    new reduceClass());
    if (3 <= cacheLevel)
        reduce1.persist(StorageLevel.MEMORY_ONLY());
    if (true == debug)
        System.out.print(" Class: reduceClass: \n" + reduce1.
            collect() + "\n");

//Iteration 2
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations2 = reduce1.join(
        actorCollaborationsAsList);
    if (3 <= cacheLevel)
        joinedDistanceAndCollaborations2.persist(StorageLevel.
            MEMORY_ONLY());
    if (true == debug)
        System.out.print(" Class: mapClass: \n" +
            joinedDistanceAndCollaborations2.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map2 =
    joinedDistanceAndCollaborations2.flatMapToPair(new
        mapClass());
    if (2 <= cacheLevel)
        map2.persist(StorageLevel.MEMORY_ONLY());
    if (true == debug)

```

```

        System.out.print(" Class: mapClass: \n" + map2.collect()
            + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce2 = map2.reduceByKey(
    new reduceClass());
if (3 <= cacheLevel)
    reduce2.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: reduceClass: \n" + reduce2.
        collect() + "\n");

//Iteration 3
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations3 = reduce2.join(
        actorCollaborationsAsList);
if (3 <= cacheLevel)
    joinedDistanceAndCollaborations3.persist(StorageLevel.
        MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" +
        joinedDistanceAndCollaborations3.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map3 =
    joinedDistanceAndCollaborations3.flatMapToPair(new
        mapClass());
if (2 <= cacheLevel)
    map3.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" + map3.collect()
        + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce3 = map3.reduceByKey(
    new reduceClass());
if (3 <= cacheLevel)
    reduce3.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: reduceClass: \n" + reduce3.
        collect() + "\n");

//Iteration 4
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations4 = reduce3.join(

```

```

        actorCollaborationsAsList);
if (3 <= cacheLevel)
    joinedDistanceAndCollaborations4.persist(StorageLevel.
        MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" +
        joinedDistanceAndCollaborations4.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map4 =
    joinedDistanceAndCollaborations4.flatMapToPair(new
        mapClass());
if (2 <= cacheLevel)
    map4.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" + map4.collect()
        + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce4 = map4.reduceByKey(
    new reduceClass());
if (3 <= cacheLevel)
    reduce4.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: reduceClass: \n" + reduce4.
        collect() + "\n");

//Iteration 5
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations5 = reduce4.join(
        actorCollaborationsAsList);
if (3 <= cacheLevel)
    joinedDistanceAndCollaborations5.persist(StorageLevel.
        MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" +
        joinedDistanceAndCollaborations5.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map5 =
    joinedDistanceAndCollaborations5.flatMapToPair(new
        mapClass());
if (3 <= cacheLevel)
    map5.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)

```

```

        System.out.print(" Class: mapClass: \n" + map5.collect()
            + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce5 = map5.reduceByKey(
    new reduceClass());
if (3 <= cacheLevel)
    reduce5.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: reduceClass: \n" + reduce5.
        collect() + "\n");

//Iteration 6
//Join
JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
    joinedDistanceAndCollaborations6 = reduce5.join(
        actorCollaborationsAsList);
if (3 <= cacheLevel)
    joinedDistanceAndCollaborations6.persist(StorageLevel.
        MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" +
        joinedDistanceAndCollaborations6.collect() + "\n");
//Map
//Spark Stage @ flatmapToPair
JavaPairRDD<Integer, Integer> map6 =
    joinedDistanceAndCollaborations6.flatMapToPair(new
        mapClass());
if (2 <= cacheLevel)
    map6.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: mapClass: \n" + map6.collect()
        + "\n");
//Reduce
JavaPairRDD<Integer, Integer> reduce6 = map6.reduceByKey(
    new reduceClass());
if (3 <= cacheLevel)
    reduce6.persist(StorageLevel.MEMORY_ONLY());
if (true == debug)
    System.out.print(" Class: reduceClass: \n" + reduce6.
        collect() + "\n");

// Write back results to file
reduce6.saveAsTextFile("file:/home/tom/jan_async/
    xDegreesOfSeparation/results");
//Write back results to HDFS

```

```

        //reduce6.saveAsTextFile("hdfs://localhost:9000/user/tom/
        results");
    }
}

/*
//If using files from omdb.org
//Create the list to store the actorDistances between
    iterations
List<JavaPairRDD<Integer, Integer>> actorDistancesList =
    new ArrayList<JavaPairRDD<Integer, Integer>>();
//Add the initialised list
actorDistancesList.add(actorDistances);

//Set the number of degrees (of separation)
Integer iterations = 6;
for (int i = 0; i < iterations; i++)
{
    //Join
    JavaPairRDD<Integer, Tuple2<Integer, Iterable<Integer>>>
        joinedDistanceAndCollaborations = actorDistancesList.
        get(i).join(actorCollaborationsAsList);
    if (true == debug)
        System.out.print("Class: mapClass: \n" +
            joinedDistanceAndCollaborations.collect() + "\n");
//joinedDistanceAndCollaborations.persist(StorageLevel.
MEMORY_ONLY());
    //Map
    JavaPairRDD<Integer, Integer> map =
        joinedDistanceAndCollaborations.flatMapToPair(new
        mapClass());
    if (true == debug)
        System.out.print("Class: mapClass: \n" + map.collect()
            + "\n");
//map.persist(StorageLevel.MEMORY_ONLY());
    //Reduce
    JavaPairRDD<Integer, Integer> reduce = map.reduceByKey(
        new reduceClass());
//reduce.persist(StorageLevel.MEMORY_ONLY());
    actorDistancesList.add(reduce);
    if (true == debug)
        System.out.print("Class: reduceClass: \n" + reduce.
            collect() + "\n");
}
}

```

```

    }
    if (true == debug)
    {
        System.out.print(" Final distances: \n" +
            actorDistancesList.get(actorDistancesList.size()-1).
            collect() + "\n");
        System.out.print(" Number of elements: \n" +
            actorDistancesList.get(actorDistancesList.size()-1).
            count() + "\n");
    }
    //HDFS
    // actorDistancesList.get(actorDistancesList.size()-1).
    // saveAsTextFile(" hdfs://localhost:9000/user/tom/results")
    ;
    //Text
    actorDistancesList.get(actorDistancesList.size()-1).
    saveAsTextFile(" file:/Users/rjan/projects/spark2/
    xDegreesOfSeparation/results");
*/

```

A.3 Altered Apache Spark code

Code changes in Apache Spark implemented by Jan Rellermeyer. Only added functions or codeblocks are shown.

A.3.1 main/scala/org/apache/spark/storage/BlockManager.scala

```

private[spark] class BlockManager(
    executorId: String,
    actorSystem: ActorSystem,
    val master: BlockManagerMaster,
    defaultSerializer: Serializer,
    maxMemory: Long,
    val conf: SparkConf,
    securityManager: SecurityManager,
    mapOutputTracker: MapOutputTracker)
    extends Logging {

    val shuffleBlockManager = new ShuffleBlockManager(this)
    val diskBlockManager = new DiskBlockManager(
        shuffleBlockManager,
        conf.get("spark.local.dir", System.getProperty("java.io.
            tmpdir")))
    // JR: 11/25/14 added a flash block manager
    val flashBlockManager = FlashBlockManager

```

```
val connectionManager = new ConnectionManager(0, conf,
    securityManager)
```

A.3.2 main/scala/org/apache/spark/storage/capi/FlashBlockManager.scala

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more
 * contributor license agreements. See the NOTICE file
 * distributed with
 * this work for additional information regarding copyright
 * ownership.
 * The ASF licenses this file to You under the Apache License,
 * Version 2.0
 * (the "License"); you may not use this file except in
 * compliance with
 * the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS"
 * BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing
 * permissions and
 * limitations under the License.
 */
package org.apache.spark.storage.capi

import org.apache.spark.storage.BlockId
import org.apache.spark.storage.TempBlockId
import java.util.UUID

/**
 * JR: added block manager to support the CAPI system in spark
 */
object FlashBlockManager {
    final val UseFlash: Boolean = true
    System.loadLibrary("flashblock")
    init
    def newBlockId(): BlockId = new TempBlockId(UUID.randomUUID())
    )
    @native def init: Unit
```

```

    @native def putBlock(blockId: String, block: Array[Byte]):
      Unit
    @native def getBlock(blockId: String): Array[Byte]
    @native def deleteBlock(blockId: String): Unit
    @native def close: Unit
  }

```

A.3.3 main/scala/org/apache/spark/storage/capi/FlashBlockObjectWriter.scala

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more
 * contributor license agreements. See the NOTICE file
 * distributed with
 * this work for additional information regarding copyright
 * ownership.
 * The ASF licenses this file to You under the Apache License,
 * Version 2.0
 * (the "License"); you may not use this file except in
 * compliance with
 * the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS"
 * BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing
 * permissions and
 * limitations under the License.
 */
package org.apache.spark.storage.capi

import org.apache.spark.storage.BlockObjectWriter
import org.apache.spark.Logging
import org.apache.spark.storage.BlockId
import org.apache.spark.serializer.Serializer
import java.io.OutputStream
import org.apache.spark.serializer.SerializationStream
import java.io.ByteArrayOutputStream
import org.apache.spark.storage.FileSegment
import java.nio.ByteBuffer

```

```

/**
 * JR: added a block object writer for flash
 */
class FlashBlockObjectWriter(
  blockId: BlockId, serializer: Serializer,
  bufferSize: Int,
  compressStream: OutputStream => OutputStream,
  syncWrites: Boolean) extends BlockObjectWriter(blockId) with
  Logging {

  // TODO: should be the block size...
  //private var buf: ByteBuffer = ByteBuffer.allocateDirect
  //  (10000000)
  private var bos: ByteArrayOutputStream = null
  private var bs: OutputStream = null
  private var objOut: SerializationStream = null
  private var initialized: Boolean = false
  private var finalPosition: Long = -1

  override def open(): BlockObjectWriter = {
    bos = new ByteArrayOutputStream()
    bs = compressStream(bos)
    objOut = serializer.newInstance().serializeStream(bs)
    initialized = true
    this
  }

  override def close() {
    if (initialized) {
      if (syncWrites) {
        // Force outstanding writes to disk and track how long
        // it takes
        objOut.flush()
      }
      objOut.close()

      bos = null
      bs = null
      objOut = null
      initialized = false
    }
  }

  override def isOpen: Boolean = objOut != null

```

```

override def commitAndClose(): Unit = {
  if (initialized) {
    objOut.flush()
    bs.flush()

    // write the block out to flash
    FlashBlockManager.putBlock(blockId.toString, bos.
      toByteArray);

    finalPosition = bos.size();

    close()
  }
}

override def revertPartialWritesAndClose() {
  // nop
}

override def write(value: Any) {
  if (!initialized) {
    open()
  }
  objOut.writeObject(value)
}

override def fileSegment(): FileSegment = {
  throw new UnsupportedOperationException(
    "FlashBlockObjectWriter does not support fileSegment()
    ");
}

override def timeWriting(): Long = 0

override def bytesWritten: Long = {
  assert(finalPosition != -1, "bytesWritten is only valid
    after successful commit()")
  finalPosition
}
}

```

A.3.4 main/scala/org/apache/spark/storage/capi/FlashMapIterator.scala

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more

```

```

* contributor license agreements. See the NOTICE file
  distributed with
* this work for additional information regarding copyright
  ownership.
* The ASF licenses this file to You under the Apache License,
  Version 2.0
* (the "License"); you may not use this file except in
  compliance with
* the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
  software
* distributed under the License is distributed on an "AS IS"
  BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
  or implied.
* See the License for the specific language governing
  permissions and
* limitations under the License.
*/
package org.apache.spark.storage.capi

import scala.collection.mutable.ArrayBuffer
import org.apache.spark.storage.BlockId
import org.apache.spark.serializer.DeserializationStream
import java.io.EOFException
import java.io.ByteArrayInputStream
import org.apache.spark.storage.BlockManager
import org.apache.spark.serializer.SerializerInstance
import org.apache.spark.util.collection.AbstractMapIterator
import org.apache.spark.Logging

/**
 * JR: added a flash map iterator to retrieve content of a
 * stored block back from flash
 */
class FlashMapIterator[K, C](
  blockManager: BlockManager,
  ser: SerializerInstance,
  blockId: BlockId, batchSize: ArrayBuffer[Long],
  serializerBatchSize: Long)
  extends AbstractMapIterator[K, C] with Logging {

```

```

private val batchOffsets = batchSizes.scanLeft(0L)(- + -) //
    Size will be batchSize.length + 1
// TODO: ???s
//assert(file.length() == batchOffsets(batchOffsets.length -
    1))

private var batchIndex = 0 // Which batch we're in
private var inStream: ByteArrayInputStream = null

// An intermediate stream that reads from exactly one batch
// This guards against pre-fetching and other arbitrary
    behavior of higher level streams
private var deserializeStream = nextBatchStream()
private var nextItem: (K, C) = null
private var objectsRead = 0

private var bytes: Array[Byte] = null;

/**
 * Construct a stream that reads only from the next batch.
 */
private def nextBatchStream(): DeserializationStream = {
    logInfo("@DEBUG@ I AM HERE, BATCH OFFSETS: " + batchOffsets
        +
        " AND BYTES=" + bytes + " AND BATCH INDEX = " +
            batchIndex +
        " BATCH SIZES WERE " + batchSizes)

    // check if we have read the block from flash
    if (bytes == null) {
        bytes = FlashBlockManager.getBlock(blockId.toString);
    }

    logInfo("GOT " + bytes.length + " bytes back from JNI")

    // Note that batchOffsets.length = numBatches + 1 since we
        did a scan above; check whether
    // we're still in a valid batch.
    if (batchIndex < batchOffsets.length - 1) {
        if (deserializeStream != null) {
            deserializeStream.close()
            inStream.close()
            deserializeStream = null
            inStream = null

```

```

    }

    val start = batchOffsets(batchIndex)
    batchIndex += 1

    val end = batchOffsets(batchIndex)

    assert(end >= start, "start = " + start + ", end = " +
        end +
        ", batchOffsets = " + batchOffsets.mkString("[", ", ", ", ",
            "]""))

    inStream = new ByteArrayInputStream(bytes, start.toInt, (
        end - start).toInt)

    val compressedStream = blockManager.wrapForCompression(
        blockId, inStream)
    ser.deserializeStream(compressedStream)
} else {
    // No more batches left
    cleanup()
    null
}
}
}

/**
 * Return the next (K, C) pair from the deserialization
 * stream.
 *
 *
 * If the current batch is drained, construct a stream for
 * the next batch and read from it.
 * If no more pairs are left, return null.
 */
private def readNextItem(): (K, C) = {
    try {
        val item = deserializeStream.readObject().asInstanceOf[(K
            , C)]
        objectsRead += 1
        if (objectsRead == serializerBatchSize) {
            objectsRead = 0
            deserializeStream = nextBatchStream()
        }
        item
    } catch {
        case e: EOFException =>

```

```

        cleanup()
        null
    }
}

override def hasNext: Boolean = {
    if (nextItem == null) {
        if (deserializeStream == null) {
            return false
        }
        nextItem = readNextItem()
    }
    nextItem != null
}

override def next(): (K, C) = {
    val item = if (nextItem == null) readNextItem() else
        nextItem
    if (item == null) {
        throw new NoSuchElementException
    }
    nextItem = null
    item
}

// TODO: Ensure this gets called even if the iterator isn't
// drained.
private def cleanup() {
    batchIndex = batchOffsets.length // Prevent reading any
        other batch
    val ds = deserializeStream
    deserializeStream = null
    inStream = null
    ds.close()
    FlashBlockManager.deleteBlock(blockId.toString)
}
}

```

A.3.5 main/scala/org/apache/spark/util/collection/ExternalAppendOnlyMap.scala

```

@DeveloperApi
class ExternalAppendOnlyMap[K, V, C](
    createCombiner: V => C,
    mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C,
    serializer: Serializer = SparkEnv.get.serializer,

```

```

blockManager: BlockManager = SparkEnv.get.blockManager)
extends Iterable[(K, C)] with Serializable with Logging {

private var currentMap = new SizeTrackingAppendOnlyMap[K, C]
// private val spilledMaps = new ArrayBuffer[DiskMapIterator]
// JR: 11/26/14
private val spilledMaps = new ArrayBuffer[AbstractMapIterator
  [K,C]]

private val sparkConf = SparkEnv.get.conf
// JR: 11/25/14 added flash block manager
private val flashBlockManager = blockManager.
  flashBlockManager
private val diskBlockManager = blockManager.diskBlockManager
private val shuffleMemoryManager = SparkEnv.get.
  shuffleMemoryManager

// Number of pairs inserted since last spill; note that we
// count them even if a value is merged
// with a previous key in case we're doing something like
// groupBy where the result grows
private var elementsRead = 0L

// Number of in-memory pairs inserted before tracking the map
// 's shuffle memory usage
private val trackMemoryThreshold = 1000

// How much of the shared memory pool this collection has
// claimed
private var myMemoryThreshold = 0L

/**
 * Size of object batches when reading/writing from
 * serializers.
 *
 * Objects are written in batches, with each batch using its
 * own serialization stream. This
 * cuts down on the size of reference-tracking maps
 * constructed when deserializing a stream.
 *
 * NOTE: Setting this too low can cause excessive copying
 * when serializing, since some serializers
 * grow internal data structures by growing + copying every
 * time the number of objects doubles.
 */

```

```

private val serializerBatchSize = sparkConf.getLong("spark.
    shuffle.spill.batchSize", 10000)

// How many times we have spilled so far
private var spillCount = 0

// Number of bytes spilled in total
private var _memoryBytesSpilled = 0L
private var _diskBytesSpilled = 0L

private val fileBufferSize = sparkConf.getInt("spark.shuffle.
    file.buffer.kb", 100) * 1024
private val keyComparator = new HashComparator[K]
private val ser = serializer.newInstance()

/**
 * Insert the given key and value into the map.
 */
def insert(key: K, value: V): Unit = {
    insertAll(Iterator((key, value)))
}

and

/**
 * Sort the existing contents of the in-memory map and spill
 * them to a temporary file on disk.
 */
private def spill(mapSize: Long): Unit = {
    spillCount += 1
    val threadId = Thread.currentThread().getId
    //logInfo("Thread %d spilling in-memory map of %d MB to
        disk (%d time%s so far)"
        // .format(threadId, mapSize / (1024 * 1024), spillCount,
        // if (spillCount > 1) "s" else "")
    //logInfo("Test2: %d"
    // .format(spillCount))

    // JR 11/25/14: inserted second code path for using Flash
    if (!flashBlockManager.UseFlash) {
        val (blockId, file) = diskBlockManager.createTempBlock()
        var writer = blockManager.getDiskWriter(blockId, file,
            serializer, fileBufferSize)
        var objectsWritten = 0

```

```

// List of batch sizes (bytes) in the order they are
// written to disk
val batchSizes = new ArrayBuffer[Long]

// Flush the disk writer's contents to disk, and update
// relevant variables
def flush() = {
    val w = writer
    writer = null
    w.commitAndClose()
    val bytesWritten = w.bytesWritten
    batchSizes.append(bytesWritten)
    _diskBytesSpilled += bytesWritten
    objectsWritten = 0
}

var success = false
try {
    val it = currentMap.destructiveSortedIterator(
        keyComparator)
    while (it.hasNext) {
        val kv = it.next()
        writer.write(kv)
        objectsWritten += 1

        if (objectsWritten == serializerBatchSize) {
            flush()
            writer = blockManager.getDiskWriter(blockId, file,
                serializer, fileBufferSize)
        }
    }
    if (objectsWritten > 0) {
        flush()
    } else if (writer != null) {
        val w = writer
        writer = null
        w.revertPartialWritesAndClose()
    }
    success = true
} finally {
    if (!success) {
        // This code path only happens if an exception was
        // thrown above before we set success;
        // close our stuff and let the exception be thrown
        // further
    }
}

```

```

        if (writer != null) {
            writer.revertPartialWritesAndClose()
        }
        if (file.exists()) {
            file.delete()
        }
    }
}

currentMap = new SizeTrackingAppendOnlyMap[K, C]
spilledMaps.append(new DiskMapIterator(file, blockId,
    batchSize))
} else {
    val blockId = flashBlockManager.newBlockId()
    var writer = blockManager.getFlashWriter(blockId,
        serializer, fileBufferSize)
    var objectsWritten = 0

    // List of batch sizes (bytes) in the order they are
    // written to disk
    val batchSize = new ArrayBuffer[Long]

    // Flush the disk writer's contents to disk, and update
    // relevant variables
    def flush() = {
        val w = writer
        writer = null
        w.commitAndClose()
        val bytesWritten = w.bytesWritten
        batchSize.append(bytesWritten)
        _diskBytesSpilled += bytesWritten
        objectsWritten = 0
    }

    var success = false
    try {
        val it = currentMap.destructiveSortedIterator(
            keyComparator)
        while (it.hasNext) {
            val kv = it.next()
            writer.write(kv)
            objectsWritten += 1

            if (objectsWritten == serializerBatchSize) {
                flush()
            }
        }
    }
}

```

```

        writer = blockManager.getFlashWriter(blockId,
            serializer, fileBufferSize)
    }
}
if (objectsWritten > 0) {
    flush()
} else if (writer != null) {
    val w = writer
    writer = null
    w.revertPartialWritesAndClose()
}
success = true
} finally {
    if (!success) {
        // This code path only happens if an exception was
        // thrown above before we set success;
        // close our stuff and let the exception be thrown
        // further
        if (writer != null) {
            writer.revertPartialWritesAndClose()
        }
        // TODO: deal with deleting the block...
    }
}

currentMap = new SizeTrackingAppendOnlyMap[K, C]
spilledMaps.append(new FlashMapIterator(
    blockManager, ser, blockId, batchSize,
    serializerBatchSize))
}

// Release our memory back to the shuffle pool so that
// other threads can grab it
shuffleMemoryManager.release(myMemoryThreshold)
myMemoryThreshold = 0L

elementsRead = 0
memoryBytesSpilled += mapSize
}

```

A.4 Code for the FlashSystem and Apache Spark

Code framework created by Jan Rellermeyer, code added by Tom Hubregtsen.

```

/*
 * Flash code for use with Apache Spark

```

```
*
*   Created 09/26/2014
*   Jan S. Rellermeyer
*   Create function framework for hooks in Apache Spark
*
*   Edit:
*   Tom Hubregtsen
*   tom@hubregtsen.com
*   Add code to the function framework
*/

#include <errno.h>
#include "flashblock.h"
#include "flash_block_manager.h"
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include "arkdb.h"
#include <ark.h>

//For ark
int bref = 34;
int grow = 1024;
uint64_t hc = 16;
uint64_t bs = 4096;
char *dev = NULL;
//char *dev = "/dev/cxl/afu0.0s";
uint64_t vlim = 256;
uint64_t size = 1024 * 1024;
#define NASYNC 128
static __thread ARC *arc;
ARK *ark = NULL;
jbyte* memStore;
jint memStoreSize;

//Timing variables
long int flash_set_sec = 0;
long int flash_set_msec = 0;
long int flash_set_loop_sec = 0;
long int flash_set_loop_msec = 0;
long int flash_get_sec = 0;
long int flash_get_msec = 0;
long int flash_get_loop_sec = 0;
```

```

long int flash_get_loop_msec = 0;
long int hdd_set_sec = 0;
long int hdd_set_msec = 0;
long int mem_set_sec = 0;
long int mem_set_msec = 0;
long int hdd_get_sec = 0;
long int hdd_get_msec = 0;
long int mem_get_sec = 0;
long int mem_get_msec = 0;
int errorCounter = 0;

static inline void throwIOExceptionE(const char* msg, int err)
{
    printf("IOException: %s; code:%d\n", msg, err);
    errorCounter++;
    printf("\n\n\n\n\n\n\n **Error*** \n\n\n\n\n\n\n\n");
}

static inline void throwOOMError(const char* msg) {
    printf("Out of memory: %s\n", msg);
    errorCounter++;
    printf("\n\n\n\n\n\n\n **Error*** \n\n\n\n\n\n\n\n");
}

static inline void throwIllegalStateException(const char* msg)
{
    printf("Illegal State: %s\n", msg);
    errorCounter++;
    printf("\n\n\n\n\n\n\n **Error*** \n\n\n\n\n\n\n\n");
}

typedef struct
{
    int errcode;
    int64_t res;
    uint64_t condvar;
} cb_info __attribute__((aligned(16)));

static inline void callback_set(int errcode, uint64_t dt,
    int64_t res)
{
    cb_info *s = (cb_info*) dt;
    s->errcode = errcode;
    s->res = res;
    s->condvar = 1;
}

```

```

    __sync_synchronize();
}

static inline void callback_exists(int errcode, uint64_t dt,
    int64_t res)
{
    cb_info *s = (cb_info*) dt;
    s->errcode = errcode;
    s->res = res;
    s->condvar = 1;
    __sync_synchronize();
}

static inline void callback_get(int errcode, uint64_t dt,
    int64_t res)
{
    cb_info *s = (cb_info*) dt;
    s->errcode = errcode;
    s->res = res;
    s->condvar = 1;
    __sync_synchronize();
}
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void* reserved) {
    UNUSED(vm)
    UNUSED(reserved)
    return JNI_VERSION_1_2;
}

JNIEXPORT void JNICALL JNI_OnUnLoad(JavaVM *vm, void *reserved)
{
    UNUSED(vm)
    UNUSED(reserved)
    ark_delete((ARK*) ark);
}

/*
 * Class:      org.apache.spark.storage_capi_FlashBlockManager__
 * Method:     init
 * Signature:  ()V
 */
JNIEXPORT void JNICALL
    Java_org_apache_spark_storage_capi_FlashBlockManager_00024_init
    (
        JNIEnv * env, jobject obj) {
    UNUSED(env);

```

```

    UNUSED(obj);
}

/*
 * Class:      org_apache_spark_storage_capi_FlashBlockManager_
 * Method:     putBlock
 * Goal:       save a block into flash
 * Signature:  (Ljava/lang/String;[B)V
 */
JNIEXPORT void JNICALL
    Java_org_apache_spark_storage_capi_FlashBlockManager_00024_putBlock
    (
        JNIEnv * env, jobject obj, jstring key,
        jbyteArray value) {
    UNUSED(obj)

    // get the size of the block
    jint _size = (*env)->GetArrayLength(env, value);
    // declare the block as a jbyte array
    jbyte* _block = (*env)->GetByteArrayElements(env, value, NULL
    );
    // get the key
    const char* _key = (*env)->GetStringUTFChars(env, key, NULL);

    // check if Flash is connected
    if(ark == NULL)
    {
        int rc_ark = ark_create_verbose(dev, &ark, size, bs, hc,
            vlim, bref, grow, 0,0);
        if (rc_ark != 0)
        {
            printf("Fail on Ark\n");
            fprintf(stderr, "ark_create failed\n");
            return;
        }
        else
        {
            printf("ark created okay\n");
        }
    }
    else
    {
        printf("Ark okay\n");
    }
}

```

```

// check if we have an ID for this process
if (arc==NULL)
{
    printf("\n\n Arc == null, creating again \n\n");
    ark_connect_verbose(&arc, ark, NASYNC);
    if (arc==NULL)
    {
        printf("Error creating arc\n");
    }
    else
    {
        printf("Arc create succesfull\n");
    }
}
else
{
    printf("ark okay\n");
}

cb_info s;
int rc = 0;
// start the block transfer in async mode
struct timeval start, stop;
struct timeval start_loop, stop_loop;
gettimeofday(&start, NULL);
rc = ark_set_async_cb(arc, (uint64_t) strlen(_key), (uint8_t
    *) _key, (uint64_t) _size, (jbyte*) _block, callback_set,
    (uint64_t) &s);
if (rc != 0)
{
    printf("Return code: %d\n", rc);
    throwIOExceptionE("Could not enqueue set command", rc);
}
else
{
    printf("Return code correct\n");
}
gettimeofday(&stop, NULL);
// Loop until block is saved
gettimeofday(&start_loop, NULL);
while (!__sync_bool_compare_and_swap(&s.condvar, 1, 0))
{
    // spin
}
printf("After spin: %ldu\n", s.condvar);

```

```

    if (s.errcode != 0)
    {
        throwIOExceptionE("put", s.errcode);
    }
    else
    {
        printf("put okay\n");
    }
    gettimeofday(&stop_loop, NULL);
    flash_set_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
    flash_set_msec = flash_set_msec - (start.tv_sec*1000) - (
        start.tv_usec/1000);
    flash_set_loop_msec += (stop_loop.tv_sec*1000)+(stop_loop.
        tv_usec/1000);
    flash_set_loop_msec = flash_set_loop_msec - (start_loop.
        tv_sec*1000) - (start_loop.tv_usec/1000);
    printf("Flash set command: %ldms\n", flash_set_msec);
    printf("Flash set loop: %ldms\n", flash_set_loop_msec);

//HDD
    FILE *f = fopen(_key, "w");
    if (f == NULL)
    {
        throwIOException(env, "could not open file spill.txt");
    }
    jbyte *_buf = malloc(_size * sizeof(jbyte*));
    if (_buf == NULL)
    {
        throwOutOfMemoryError(env, "malloc buf");
    }
    gettimeofday(&start, NULL);
    fwrite(_buf, sizeof(jbyte*), _size, f);
    gettimeofday(&stop, NULL);
    hdd_put_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
    hdd_put_msec = hdd_put_msec - (start.tv_sec*1000) - (start.
        tv_usec/1000);
    printf("hdd put: %ldms\n", hdd_put_msec);
    fclose(f);

//Mem
    memStore = (jbyte*) malloc(_size * sizeof(jbyte*));
    if (memStore == NULL)
    {
        fprintf(stderr, "OUT OF MEMORY\n");
        throwOutOfMemoryError(env, "malloc memStore");
    }

```

```

    }
    uint8_t* async_rbuf = malloc(_size*sizeof(uint8_t));
    gettimeofday(&start, NULL);
    memcpy(memStore, async_rbuf, _size);
    gettimeofday(&stop, NULL);
    mem_put_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
    mem_put_msec = mem_put_msec - (start.tv_sec*1000) - (start.
        tv_usec/1000);
    printf("mem put: %ldms\n", mem_get_msec);
    free(memStore);
}

/*
 * Class:      org.apache.spark.storage_capi_FlashBlockManager_
 * Method:     getBlock
 * Goal:       retrieve a saved block
 * Signature:  (Ljava/lang/String;) [B
 */
JNIEXPORT jbyteArray JNICALL
Java_org_apache_spark_storage_capi_FlashBlockManager_00024_getBlock
(
    JNIEnv * env, jobject obj, jstring key) {
    UNUSED(env)
    UNUSED(obj)
    const char* _key = (*env)->GetStringUTFChars(env, key, NULL);
    long int loopcount = 0;

    cb_info s_exists;
    int rc = 0;
    //Check if the block is in memory in async mode
    rc= ark_exists_async_cb(arc, (uint64_t)(strlen(_key)), (
        uint8_t *)(_key), callback_exists, (uint64_t) &s_exists);
    if (rc != 0)
    {
        throwIOExceptionE("Could not enqueue set command", rc);
    }
    else
        printf("Exist rc okay\n");
    // Wait until we get an answer
    while (!__sync_bool_compare_and_swap(&s_exists.condvar, 1, 0)
        )
    {
        // spin
        loopcount++;
    }
}

```

```

printf("Loopcount in exist: %ld\n", loopcount);
if (s_exists.errcode != 0)
{
    throwIOExceptionE("exist", s_exists.errcode);
}
printf("Exists errcode okay\n");
cb_info s2;
rc = 0;
uint64_t valueLength = (uint64_t) s_exists.res;
uint8_t* async_rbuf = malloc(valueLength*sizeof(uint8_t));

struct timeval start, stop;
struct timeval gaan, niet;
gettimeofday(&start, NULL);
gettimeofday(&gaan, NULL);
// Request the block in async mode
rc = ark_get_async_cb(arc, (uint64_t)(strlen(_key)), (uint8_t
    *)(_key), (uint64_t) valueLength, async_rbuf, 0,
    callback_get, (uint64_t) &s2);
gettimeofday(&stop, NULL);
if (rc != 0)
{
    throwIOExceptionE("Could not enqueue set command", rc);
}
else
    printf("get r okay\n");
loopcount = 0;
//Wait for the actual block
while (!__sync_bool_compare_and_swap(&s2.condvar, 1, 0))
{
    // spin
    loopcount++;
}
printf("Loopcount in get: %ld\n", loopcount);
if (s2.res < 0)
{
    printf("Return size < 0\n");
}
if (s2.errcode != 0)
{
    throwIOExceptionE("get", s2.errcode);
}
else
    printf("Get errcode okay\n");
gettimeofday(&niet, NULL);

```

```

flash_get_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
flash_get_msec = flash_get_msec - (start.tv_sec*1000) - (
    start.tv_usec/1000);
printf("Flash get command: %ldms\n", flash_get_msec);
printf("recognize start_tv_sec;%ld:\n", gaan.tv_sec);
printf("recognize start_tv_used;%ld:\n", gaan.tv_usec);
printf("recognize stop_tv_sec;%ld:\n", niet.tv_sec);
printf("recognize stop_tv_usec;%ld\n", niet.tv_usec);
printf("error: %d\n", errorCounter);

//HDD
FILE *f = fopen(_key, "r");
if (f == NULL)
{
    throwIOException(env, "could not open file spill.txt");
}
jbyte *_buf = malloc(valueLength * sizeof(jbyte*));
if (_buf == NULL)
{
    throwOutOfMemoryError(env, "malloc buf");
}
gettimeofday(&start, NULL);
fread(_buf, sizeof(jbyte*), valueLength, f);
gettimeofday(&stop, NULL);
hdd_get_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
hdd_get_msec = hdd_get_msec - (start.tv_sec*1000) - (start.
    tv_usec/1000);
printf("hdd get: %ldms\n", hdd_get_msec);
fclose(f);

//Mem
//printf("Putblock in mode: Memory\n");
memStore = (jbyte*) malloc(valueLength * sizeof(jbyte*));
if (memStore == NULL) {
    fprintf(stderr, "OUT OF MEMORY\n");
    throwOutOfMemoryError(env, "malloc memStore");
}
//memStore[0] = 33;
gettimeofday(&start, NULL);
memcpy(memStore, async_rbuf, valueLength);
gettimeofday(&stop, NULL);
mem_get_msec += (stop.tv_sec*1000)+(stop.tv_usec/1000);
mem_get_msec = mem_get_msec - (start.tv_sec*1000) - (start.
    tv_usec/1000);
printf("mem get: %ldms\n", mem_get_msec);

```

```

    free(memStore);
    jbyteArray result;
    result = (*env)->NewByteArray(env, valueLength);
    (*env)->SetByteArrayRegion(env, result, 0, valueLength, (
        jbyte *) async_rbuf);
    //(*env)->SetByteArrayRegion(env, result, 0, valueLength, (
        jbyte *) async_rbuf);
    return result;
}

/*
 * Class:      org_apache_spark_storage_capi_FlashBlockManager_
 * Method:     deleteBlock
 * Goal:       Delete a block from Flash
 * Signature:  (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
    Java_org_apache_spark_storage_capi_FlashBlockManager_00024_deleteBlock
    (
        JNIEnv * env, jobject obj, jstring key) {
    UNUSED(env)
    UNUSED(obj)
    UNUSED(key)
    const char* _key = (*env)->GetStringUTFChars(env, key, NULL);
    int64_t rval = 0;
    int rc = 0;
    // Delete the block from Flash
    rc = ark_del(ark, (uint64_t)(strlen(_key)), (uint8_t*)(_key),
        &rval);
    if (rc != 0)
        printf("error deleting key\n");
    (*env)->ReleaseStringUTFChars(env, key, _key);
}

/*
 * Class:      org_apache_spark_storage_capi_FlashBlockManager_
 * Method:     close
 * Signature:  ()V
 */
JNIEXPORT void JNICALL
    Java_org_apache_spark_storage_capi_FlashBlockManager_00024_close
    (
        JNIEnv * env, jobject obj) {
    UNUSED(env)
    UNUSED(obj)

```

```
}
```

A.5 Code for the writing speed test

```
/*
 * SpeedTest
 *
 * Test the difference in writing speeds between HDD DRAM and
 *   CAPI Flash in async
 *
 * Author: Tom Hubregtsen
 * Email: tom@hubregtsen.com
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <stdint.h>
#include <inttypes.h>
#include <sys/time.h>
#include "cl.h"
#include "bv.h"
#include "arkdb.h"
#include <ark.h>

//For ark
int bref = 34;
int grow = 1024;
uint64_t hc = 16;
uint64_t bs = 4096;
char *dev = NULL;
uint64_t vlim = 256;
uint64_t size = 1024 * 1024;

int tst_tom_entry(int argc, char **argv)
{
//init flash
ARK *ark = NULL;
int rc_ark = ark_create_verbose(dev, &ark, size, bs, hc, vlim
, bref, grow, 0,0);
```

```

if (rc_ark != 0)
{
    fprintf(stderr, "ark_create failed\n");
    return 1;
}

//Read argument
long fileSize = 0;
int argSize = strlen(argv[3]);
int i = 0;
long tens = 1;
char current = 0;
for (i = 0; i < argSize; i++)
{
    current = (int) argv[3][argSize-i-1] - 48;
    fileSize = fileSize + tens * current;
    tens*=10;
}
printf("%lu:", fileSize);

//prepare value
uint64_t valueLength = (uint64_t) fileSize;
uint8_t* valueBuffer = malloc(valueLength*sizeof(uint8_t));
//prepare result buffer
uint8_t* rbuf = malloc(valueLength*sizeof(uint8_t));

//Write to DRAM
printf("Start\n");
struct timeval stop, start;
gettimeofday(&start, NULL);
memcpy(valueBuffer, rbuf, fileSize);
gettimeofday(&stop, NULL);
printf("%lu:", (stop.tv_sec - start.tv_sec));
printf("%ld;", (stop.tv_usec - start.tv_usec));

//Write to Flash
//write
int64_t rval = 0;
gettimeofday(&start, NULL);
ark_set(ark, (uint64_t)(strlen("key")), (uint8_t *)("key"),
        valueLength, valueBuffer, &rval);
gettimeofday(&stop, NULL);
printf("%lu:", stop.tv_usec - start.tv_usec);

//exists

```

```
    rval = 0;
    //rc is always 0, rval works
    ark_exists(ark, (uint64_t)(strlen("key")), (uint8_t *)("key"),
               &rval);
    uint64_t length = rval;

    //get
    uint64_t voff = 0;
    rval = 0;
    ark_get(ark, (uint64_t)(strlen("key")), (uint8_t *)("key"),
            length, rbuf, voff, &rval);

    //clean
    free(rbuf);
    free(valueBuffer);
    ark_del(ark, (uint64_t)(strlen("key")), (uint8_t *)("key"), &
            rval);

    //Write to file
    gettimeofday(&start, NULL);
    FILE *f = fopen("speedTest", "w");
    if (f == NULL) {
        printf("Error, could not open file\n");
        return -1;
    }
    fwrite(valueBuffer, sizeof(uint8_t*), valueLength, f);
    fclose(f);
    gettimeofday(&stop, NULL);
    printf("%lu\n", stop.tv_usec - start.tv_usec);

    return 0;
}
```


B

Spilling code for reference

The following appendixes contain Apache Spark source code. We enclosed these solely for clarification and reference of Section 4.3. The original source can be found at the start of every section.

B.1 MemoryStore.scala

Source: <https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/storage>

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more
 * contributor license agreements. See the NOTICE file
 * distributed with
 * this work for additional information regarding copyright
 * ownership.
 * The ASF licenses this file to You under the Apache License,
 * Version 2.0
 * (the "License"); you may not use this file except in
 * compliance with
 * the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS"
 * BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing
 * permissions and
 * limitations under the License.
 */
```

```
package org.apache.spark.storage
```

```
import java.nio.ByteBuffer
import java.util.LinkedHashMap
```

```

import scala.collection.mutable
import scala.collection.mutable.ArrayBuffer

import org.apache.spark.util.{SizeEstimator, Utils}
import org.apache.spark.util.collection.SizeTrackingVector

private case class MemoryEntry(value: Any, size: Long,
    deserialized: Boolean)

/**
 * Stores blocks in memory, either as Arrays of deserialized
 *   Java objects or as
 *   serialized ByteBuffers.
 */
private[spark] class MemoryStore(blockManager: BlockManager,
    maxMemory: Long)
    extends BlockStore(blockManager) {

    private val conf = blockManager.conf
    private val entries = new LinkedHashMap[BlockId, MemoryEntry]
        (32, 0.75f, true)

    @volatile private var currentMemory = 0L

    // Ensure only one thread is putting, and if necessary,
    //   dropping blocks at any given time
    private val accountingLock = new Object

    // A mapping from thread ID to amount of memory used for
    //   unrolling a block (in bytes)
    // All accesses of this map are assumed to have manually
    //   synchronized on 'accountingLock'
    private val unrollMemoryMap = mutable.HashMap[Long, Long]()

    /**
     * The amount of space ensured for unrolling values in memory
     *   , shared across all cores.
     * This space is not reserved in advance, but allocated
     *   dynamically by dropping existing blocks.
     */
    private val maxUnrollMemory: Long = {
        val unrollFraction = conf.getDouble("spark.storage.
            unrollFraction", 0.2)
        (maxMemory * unrollFraction).toLong
    }

```

```

}

logInfo("MemoryStore started with capacity %s".format(Utils.
    bytesToString(maxMemory)))

/** Free memory not occupied by existing blocks. Note that
    this does not include unroll memory. */
def freeMemory: Long = maxMemory - currentMemory

override def getSize(blockId: BlockId): Long = {
    entries.synchronized {
        entries.get(blockId).size
    }
}

override def putBytes(blockId: BlockId, _bytes: ByteBuffer,
    level: StorageLevel): PutResult = {
    // Work on a duplicate - since the original input might be
    // used elsewhere.
    val bytes = _bytes.duplicate()
    bytes.rewind()
    if (level.deserialized) {
        val values = blockManager.dataDeserialize(blockId, bytes)
        putIterator(blockId, values, level, returnValues = true)
    } else {
        val putAttempt = tryToPut(blockId, bytes, bytes.limit(),
            deserialized = false)
        PutResult(bytes.limit(), Right(bytes.duplicate()),
            putAttempt.droppedBlocks)
    }
}

override def putArray(
    blockId: BlockId,
    values: Array[Any],
    level: StorageLevel,
    returnValues: Boolean): PutResult = {
    if (level.deserialized) {
        val sizeEstimate = SizeEstimator.estimate(values.
            asInstanceOf[AnyRef])
        val putAttempt = tryToPut(blockId, values, sizeEstimate,
            deserialized = true)
        PutResult(sizeEstimate, Left(values.iterator), putAttempt.
            droppedBlocks)
    } else {

```

```

    val bytes = blockManager.dataSerialize(blockId, values.
        iterator)
    val putAttempt = tryToPut(blockId, bytes, bytes.limit,
        deserialized = false)
    PutResult(bytes.limit(), Right(bytes.duplicate()),
        putAttempt.droppedBlocks)
  }
}

override def putIterator(
    blockId: BlockId,
    values: Iterator[Any],
    level: StorageLevel,
    returnValues: Boolean): PutResult = {
  putIterator(blockId, values, level, returnValues,
    allowPersistToDisk = true)
}

/**
 * Attempt to put the given block in memory store.
 *
 * There may not be enough space to fully unroll the iterator
 * in memory, in which case we
 * optionally drop the values to disk if
 * (1) the block's storage level specifies useDisk, and
 * (2) 'allowPersistToDisk' is true.
 *
 * One scenario in which 'allowPersistToDisk' is false is
 * when the BlockManager reads a block
 * back from disk and attempts to cache it in memory. In this
 * case, we should not persist the
 * block back on disk again, as it is already in disk store.
 */
private[storage] def putIterator(
    blockId: BlockId,
    values: Iterator[Any],
    level: StorageLevel,
    returnValues: Boolean,
    allowPersistToDisk: Boolean): PutResult = {
  val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]
  val unrolledValues = unrollSafely(blockId, values,
    droppedBlocks)
  unrolledValues match {
    case Left(arrayValues) =>

```

```

        // Values are fully unrolled in memory, so store them
        as an array
        val res = putArray(blockId, arrayValues, level,
            returnValues)
        droppedBlocks += res.droppedBlocks
        PutResult(res.size, res.data, droppedBlocks)
    case Right(iteratorValues) =>
        // Not enough space to unroll this block; drop to disk
        if applicable
        logWarning(s"Not enough space to store block $blockId
            in memory! " +
            s"Free memory is $freeMemory bytes.")
        if (level.useDisk && allowPersistToDisk) {
            logWarning(s"Persisting block $blockId to disk
                instead.")
            val res = blockManager.diskStore.putIterator(blockId,
                iteratorValues, level, returnValues)
            PutResult(res.size, res.data, droppedBlocks)
        } else {
            PutResult(0, Left(iteratorValues), droppedBlocks)
        }
    }
}

override def getBytes(blockId: BlockId): Option[ByteBuffer] =
{
    val entry = entries.synchronized {
        entries.get(blockId)
    }
    if (entry == null) {
        None
    } else if (entry.deserialized) {
        Some(blockManager.dataSerialize(blockId, entry.value.
            asInstanceOf[Array[Any]].iterator))
    } else {
        Some(entry.value.asInstanceOf[ByteBuffer].duplicate()) //
            Doesn't actually copy the data
    }
}

override def getValues(blockId: BlockId): Option[Iterator[Any]] = {
    val entry = entries.synchronized {
        entries.get(blockId)
    }
}

```

```

    if (entry == null) {
        None
    } else if (entry.deserialized) {
        Some(entry.value.asInstanceOf[Array[Any]].iterator)
    } else {
        val buffer = entry.value.asInstanceOf[ByteBuffer].
            duplicate() // Doesn't actually copy data
        Some(blockManager.dataDeserialize(blockId, buffer))
    }
}

override def remove(blockId: BlockId): Boolean = {
    entries.synchronized {
        val entry = entries.remove(blockId)
        if (entry != null) {
            currentMemory -= entry.size
            logInfo(s"Block $blockId of size ${entry.size} dropped
                from memory (free $freeMemory)")
            true
        } else {
            false
        }
    }
}

override def clear() {
    entries.synchronized {
        entries.clear()
        currentMemory = 0
    }
    logInfo("MemoryStore cleared")
}

/**
 * Unroll the given block in memory safely.
 *
 * The safety of this operation refers to avoiding potential
 * OOM exceptions caused by
 * unrolling the entirety of the block in memory at once.
 * This is achieved by periodically
 * checking whether the memory restrictions for unrolling
 * blocks are still satisfied,
 * stopping immediately if not. This check is a safeguard
 * against the scenario in which

```

```

* there is not enough free memory to accommodate the
  entirety of a single block.
*
* This method returns either an array with the contents of
  the entire block or an iterator
* containing the values of the block (if the array would
  have exceeded available memory).
*/
def unrollSafely(
  blockId: BlockId,
  values: Iterator[Any],
  droppedBlocks: ArrayBuffer[(BlockId, BlockStatus)])
: Either[Array[Any], Iterator[Any]] = {

  // Number of elements unrolled so far
  var elementsUnrolled = 0
  // Whether there is still enough memory for us to continue
  unrolling this block
  var keepUnrolling = true
  // Initial per-thread memory to request for unrolling
  blocks (bytes). Exposed for testing.
  val initialMemoryThreshold = conf.getLong("spark.storage.
    unrollMemoryThreshold", 1024 * 1024)
  // How often to check whether we need to request more
  memory
  val memoryCheckPeriod = 16
  // Memory currently reserved by this thread for this
  particular unrolling operation
  var memoryThreshold = initialMemoryThreshold
  // Memory to request as a multiple of current vector size
  val memoryGrowthFactor = 1.5
  // Previous unroll memory held by this thread, for
  releasing later (only at the very end)
  val previousMemoryReserved =
    currentUnrollMemoryForThisThread
  // Underlying vector for unrolling the block
  var vector = new SizeTrackingVector[Any]

  // Request enough memory to begin unrolling
  keepUnrolling = reserveUnrollMemoryForThisThread(
    initialMemoryThreshold)

  // Unroll this block safely, checking whether we have
  exceeded our threshold periodically
  try {

```

```

while (values.hasNext && keepUnrolling) {
    vector += values.next()
    if (elementsUnrolled % memoryCheckPeriod == 0) {
        // If our vector's size has exceeded the threshold,
        // request more memory
        val currentSize = vector.estimateSize()
        if (currentSize >= memoryThreshold) {
            val amountToRequest = (currentSize * (
                memoryGrowthFactor - 1)).toLong
            // Hold the accounting lock, in case another thread
            // concurrently puts a block that
            // takes up the unrolling space we just ensured
            // here
            accountingLock.synchronized {
                if (!reserveUnrollMemoryForThisThread(
                    amountToRequest)) {
                    // If the first request is not granted, try
                    // again after ensuring free space
                    // If there is still not enough space, give up
                    // and drop the partition
                    val spaceToEnsure = maxUnrollMemory -
                        currentUnrollMemory
                    if (spaceToEnsure > 0) {
                        val result = ensureFreeSpace(blockId,
                            spaceToEnsure)
                        droppedBlocks += result.droppedBlocks
                    }
                    keepUnrolling =
                        reserveUnrollMemoryForThisThread(
                            amountToRequest)
                }
            }
            // New threshold is currentSize *
            // memoryGrowthFactor
            memoryThreshold = currentSize + amountToRequest
        }
        elementsUnrolled += 1
    }
}

if (keepUnrolling) {
    // We successfully unrolled the entirety of this block
    Left(vector.toArray)
} else {

```

```

        // We ran out of space while unrolling the values for
        // this block
        Right(vector.iterator ++ values)
    }

    } finally {
        // If we return an array, the values returned do not
        // depend on the underlying vector and
        // we can immediately free up space for other threads.
        // Otherwise, if we return an iterator,
        // we release the memory claimed by this thread later on
        // when the task finishes.
        if (keepUnrolling) {
            val amountToRelease = currentUnrollMemoryForThisThread
                                - previousMemoryReserved
            releaseUnrollMemoryForThisThread(amountToRelease)
        }
    }
}

/**
 * Return the RDD ID that a given block ID is from, or None
 * if it is not an RDD block.
 */
private def getRddId(blockId: BlockId): Option[Int] = {
    blockId.asRDDId.map(_._rddId)
}

/**
 * Try to put in a set of values, if we can free up enough
 * space. The value should either be
 * an Array if deserialized is true or a ByteBuffer otherwise
 * . Its (possibly estimated) size
 * must also be passed by the caller.
 *
 * Synchronize on 'accountingLock' to ensure that all the put
 * requests and its associated block
 * dropping is done by only one thread at a time. Otherwise
 * while one thread is dropping
 * blocks to free memory for one block, another thread may
 * use up the freed space for
 * another block.
 *
 * Return whether put was successful, along with the blocks
 * dropped in the process.

```

```

*/
private def tryToPut(
    blockId: BlockId,
    value: Any,
    size: Long,
    deserialized: Boolean): ResultWithDroppedBlocks = {

    /* TODO: Its possible to optimize the locking by locking
       entries only when selecting blocks
    * to be dropped. Once the to-be-dropped blocks have been
       selected, and lock on entries has
    * been released, it must be ensured that those to-be-
       dropped blocks are not double counted
    * for freeing up more space for another block that needs
       to be put. Only then the actually
    * dropping of blocks (and writing to disk if necessary)
       can proceed in parallel. */

    var putSuccess = false
    val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]

    accountingLock.synchronized {
        val freeSpaceResult = ensureFreeSpace(blockId, size)
        val enoughFreeSpace = freeSpaceResult.success
        droppedBlocks ++= freeSpaceResult.droppedBlocks

        if (enoughFreeSpace) {
            val entry = new MemoryEntry(value, size, deserialized)
            entries.synchronized {
                entries.put(blockId, entry)
                currentMemory += size
            }
            val valuesOrBytes = if (deserialized) "values" else "
                bytes"
            logInfo("Block %s stored as %s in memory (estimated
                size %s, free %s)".format(
                blockId, valuesOrBytes, Utils.bytesToString(size),
                Utils.bytesToString(freeMemory)))
            putSuccess = true
        } else {
            // Tell the block manager that we couldn't put it in
            // memory so that it can drop it to
            // disk if the block allows disk storage.
            val data = if (deserialized) {
                Left(value.asInstanceOf[Array[Any]])
            }
        }
    }
}

```

```

    } else {
      Right(value.asInstanceOf[ByteBuffer].duplicate())
    }
    val droppedBlockStatus = blockManager.dropFromMemory(
      blockId, data)
    droppedBlockStatus.foreach { status => droppedBlocks +=
      ((blockId, status)) }
  }
}
ResultWithDroppedBlocks(putSuccess, droppedBlocks)
}

/**
 * Try to free up a given amount of space to store a
 * particular block, but can fail if
 * either the block is bigger than our memory or it would
 * require replacing another block
 * from the same RDD (which leads to a wasteful cyclic
 * replacement pattern for RDDs that
 * don't fit into memory that we want to avoid).
 *
 * Assume that 'accountingLock' is held by the caller to
 * ensure only one thread is dropping
 * blocks. Otherwise, the freed space may fill up before the
 * caller puts in their new value.
 *
 * Return whether there is enough free space, along with the
 * blocks dropped in the process.
 */
private def ensureFreeSpace(
  blockIdToAdd: BlockId,
  space: Long): ResultWithDroppedBlocks = {
  logInfo(s"ensureFreeSpace($space) called with curMem=
    $currentMemory, maxMem=$maxMemory")

  val droppedBlocks = new ArrayBuffer[(BlockId, BlockStatus)]

  if (space > maxMemory) {
    logInfo(s"Will not store $blockIdToAdd as it is larger
      than our memory limit")
    return ResultWithDroppedBlocks(success = false,
      droppedBlocks)
  }
}

```

```

// Take into account the amount of memory currently
// occupied by unrolling blocks
val actualFreeMemory = freeMemory - currentUnrollMemory

if (actualFreeMemory < space) {
    val rddToAdd = getRddId(blockIdToAdd)
    val selectedBlocks = new ArrayBuffer[BlockId]
    var selectedMemory = 0L

    // This is synchronized to ensure that the set of entries
    // is not changed
    // (because of getValue or getBytes) while traversing the
    // iterator, as that
    // can lead to exceptions.
    entries.synchronized {
        val iterator = entries.entrySet().iterator()
        while (actualFreeMemory + selectedMemory < space &&
            iterator.hasNext) {
            val pair = iterator.next()
            val blockId = pair.getKey
            if (rddToAdd.isEmpty || rddToAdd != getRddId(blockId)) {
                selectedBlocks += blockId
                selectedMemory += pair.getValue.size
            }
        }
    }

    if (actualFreeMemory + selectedMemory >= space) {
        logInfo(s"${selectedBlocks.size} blocks selected for
            dropping")
        for (blockId <- selectedBlocks) {
            val entry = entries.synchronized { entries.get(
                blockId) }
            // This should never be null as only one thread
            // should be dropping
            // blocks and removing entries. However the check is
            // still here for
            // future safety.
            if (entry != null) {
                val data = if (entry.deserialized) {
                    Left(entry.value.asInstanceOf[Array[Any]])
                } else {
                    Right(entry.value.asInstanceOf[ByteBuffer].
                        duplicate())
                }
            }
        }
    }
}

```

```

        }
        val droppedBlockStatus = blockManager.
            dropFromMemory(blockId, data)
        droppedBlockStatus.foreach { status =>
            droppedBlocks += ((blockId, status)) }
    }
}
return ResultWithDroppedBlocks(success = true,
    droppedBlocks)
} else {
    logInfo(s"Will not store $blockIdToAdd as it would
        require dropping another block " +
        "from the same RDD")
    return ResultWithDroppedBlocks(success = false,
        droppedBlocks)
}
}
ResultWithDroppedBlocks(success = true, droppedBlocks)
}

override def contains(blockId: BlockId): Boolean = {
    entries.synchronized { entries.containsKey(blockId) }
}

/**
 * Reserve additional memory for unrolling blocks used by
 * this thread.
 * Return whether the request is granted.
 */
private[spark] def reserveUnrollMemoryForThisThread(memory:
    Long): Boolean = {
    accountingLock.synchronized {
        val granted = freeMemory > currentUnrollMemory + memory
        if (granted) {
            val threadId = Thread.currentThread().getId
            unrollMemoryMap(threadId) = unrollMemoryMap.getOrElse(
                threadId, 0L) + memory
        }
        granted
    }
}

/**
 * Release memory used by this thread for unrolling blocks.

```

```

    * If the amount is not specified , remove the current thread '
      s allocation altogether.
    */
private[spark] def releaseUnrollMemoryForThisThread(memory:
    Long = -1L): Unit = {
    val threadId = Thread.currentThread().getId
    accountingLock.synchronized {
        if (memory < 0) {
            unrollMemoryMap.remove(threadId)
        } else {
            unrollMemoryMap(threadId) = unrollMemoryMap.getOrElse(
                threadId, memory) - memory
            // If this thread claims no more unroll memory, release
              it completely
            if (unrollMemoryMap(threadId) <= 0) {
                unrollMemoryMap.remove(threadId)
            }
        }
    }
}

/**
 * Return the amount of memory currently occupied for
   unrolling blocks across all threads.
 */
private[spark] def currentUnrollMemory: Long = accountingLock
    .synchronized {
        unrollMemoryMap.values.sum
    }

/**
 * Return the amount of memory currently occupied for
   unrolling blocks by this thread.
 */
private[spark] def currentUnrollMemoryForThisThread: Long =
    accountingLock.synchronized {
        unrollMemoryMap.getOrElse(Thread.currentThread().getId, 0L)
    }
}

```

B.2 ExternalAppendOnlyMap.scala

Source: <https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/util/collection>

```
/*
```

```
* Licensed to the Apache Software Foundation (ASF) under one
* or more
* contributor license agreements. See the NOTICE file
* distributed with
* this work for additional information regarding copyright
* ownership.
* The ASF licenses this file to You under the Apache License,
* Version 2.0
* (the "License"); you may not use this file except in
* compliance with
* the License. You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software
* distributed under the License is distributed on an "AS IS"
* BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
* or implied.
* See the License for the specific language governing
* permissions and
* limitations under the License.
*/

package org.apache.spark.util.collection

import java.io._
import java.util.Comparator
import scala.collection.BufferedIterator
import scala.collection.mutable
import scala.collection.mutable.ArrayBuffer
import com.google.common.io.ByteStreams
import org.apache.spark.{ Logging, SparkEnv }
import org.apache.spark.annotation.DeveloperApi
import org.apache.spark.serializer.{ DeserializationStream,
  Serializer }
import org.apache.spark.storage.{ BlockId, BlockManager }
import org.apache.spark.util.collection.ExternalAppendOnlyMap.
  HashComparator
import org.apache.spark.storage.capi.FlashMapIterator

/**
 * :: DeveloperApi ::
```

```

* An append-only map that spills sorted content to disk when
  there is insufficient space for it
* to grow.
*
* This map takes two passes over the data:
*
*   (1) Values are merged into combiners, which are sorted and
       spilled to disk as necessary
*   (2) Combiners are read from disk and merged together
*
* The setting of the spill threshold faces the following trade
  -off: If the spill threshold is
* too high, the in-memory map may occupy more memory than is
  available, resulting in OOM.
* However, if the spill threshold is too low, we spill
  frequently and incur unnecessary disk
* writes. This may lead to a performance regression compared
  to the normal case of using the
* non-spilling AppendOnlyMap.
*
* Two parameters control the memory threshold:
*
*   'spark.shuffle.memoryFraction' specifies the collective
  amount of memory used for storing
* these maps as a fraction of the executor's total memory.
  Since each concurrently running
* task maintains one map, the actual threshold for each map
  is this quantity divided by the
* number of running tasks.
*
*   'spark.shuffle.safetyFraction' specifies an additional
  margin of safety as a fraction of
* this threshold, in case map size estimation is not
  sufficiently accurate.
*/
@DeveloperApi
class ExternalAppendOnlyMap[K, V, C](
  createCombiner: V => C,
  mergeValue: (C, V) => C,
  mergeCombiners: (C, C) => C,
  serializer: Serializer = SparkEnv.get.serializer,
  blockManager: BlockManager = SparkEnv.get.blockManager)
  extends Iterable[(K, C)] with Serializable with Logging {

  private var currentMap = new SizeTrackingAppendOnlyMap[K, C]

```

```
// private val spilledMaps = new ArrayBuffer[DiskMapIterator]
// JR: 11/26/14
private val spilledMaps = new ArrayBuffer[AbstractMapIterator
    [K,C]]

private val sparkConf = SparkEnv.get.conf
// JR: 11/25/14 added flash block manager
private val flashBlockManager = blockManager.
    flashBlockManager
private val diskBlockManager = blockManager.diskBlockManager
private val shuffleMemoryManager = SparkEnv.get.
    shuffleMemoryManager

// Number of pairs inserted since last spill; note that we
// count them even if a value is merged
// with a previous key in case we're doing something like
// groupBy where the result grows
private var elementsRead = 0L

// Number of in-memory pairs inserted before tracking the map
// 's shuffle memory usage
private val trackMemoryThreshold = 1000

// How much of the shared memory pool this collection has
// claimed
private var myMemoryThreshold = 0L

/**
 * Size of object batches when reading/writing from
 * serializers.
 *
 * Objects are written in batches, with each batch using its
 * own serialization stream. This
 * cuts down on the size of reference-tracking maps
 * constructed when deserializing a stream.
 *
 * NOTE: Setting this too low can cause excessive copying
 * when serializing, since some serializers
 * grow internal data structures by growing + copying every
 * time the number of objects doubles.
 */
private val serializerBatchSize = sparkConf.getLong("spark.
    shuffle.spill.batchSize", 10000)

// How many times we have spilled so far
```

```

private var spillCount = 0

// Number of bytes spilled in total
private var _memoryBytesSpilled = 0L
private var _diskBytesSpilled = 0L

private val fileBufferSize = sparkConf.getInt("spark.shuffle.
    file.buffer.kb", 100) * 1024
private val keyComparator = new HashComparator[K]
private val ser = serializer.newInstance()

/**
 * Insert the given key and value into the map.
 */
def insert(key: K, value: V): Unit = {
    insertAll(Iterator((key, value)))
}

/**
 * Insert the given iterator of keys and values into the map.
 *
 * When the underlying map needs to grow, check if the global
 * pool of shuffle memory has
 * enough room for this to happen. If so, allocate the memory
 * required to grow the map;
 * otherwise, spill the in-memory map to disk.
 *
 * The shuffle memory usage of the first trackMemoryThreshold
 * entries is not tracked.
 */
def insertAll(entries: Iterator[Product2[K, V]]): Unit = {
    // An update function for the map that we reuse across
    // entries to avoid allocating
    // a new closure each time
    var curEntry: Product2[K, V] = null
    val update: (Boolean, C) => C = (hadVal, oldVal) => {
        if (hadVal) mergeValue(oldVal, curEntry._2) else
            createCombiner(curEntry._2)
    }

    while (entries.hasNext) {
        curEntry = entries.next()
        if (elementsRead > trackMemoryThreshold && elementsRead %
            32 == 0 &&
            currentMap.estimateSize() >= myMemoryThreshold) {

```

```

        // Claim up to double our current memory from the
        // shuffle memory pool
        val currentMemory = currentMap.estimateSize()
        val amountToRequest = 2 * currentMemory -
            myMemoryThreshold
        val granted = shuffleMemoryManager.tryToAcquire(
            amountToRequest)
        myMemoryThreshold += granted
        //logInfo("Comparison is %d <= %d; requested: %d;
        //granted: %d"
        // .format(myMemoryThreshold, currentMemory,
        // amountToRequest, granted))
        if (myMemoryThreshold <= currentMemory) {
            // We were granted too little memory to grow further
            // (either tryToAcquire returned 0,
            // or we already had more memory than
            // myMemoryThreshold); spill the current collection
            logInfo("Spilling")
            spill(currentMemory) // Will also release memory back
            // to ShuffleMemoryManager
        }
    }
    currentMap.changeValue(curEntry._1, update)
    elementsRead += 1
}

/**
 * Insert the given iterable of keys and values into the map.
 *
 * When the underlying map needs to grow, check if the global
 * pool of shuffle memory has
 * enough room for this to happen. If so, allocate the memory
 * required to grow the map;
 * otherwise, spill the in-memory map to disk.
 *
 * The shuffle memory usage of the first trackMemoryThreshold
 * entries is not tracked.
 */
def insertAll(entries: Iterable[Product2[K, V]]): Unit = {
    insertAll(entries.iterator)
}

/**

```

```

* Sort the existing contents of the in-memory map and spill
  them to a temporary file on disk.
*/
private def spill(mapSize: Long): Unit = {
  spillCount += 1
  val threadId = Thread.currentThread().getId
  //logInfo("Thread %d spilling in-memory map of %d MB to
    disk (%d time%s so far)"
    // .format(threadId, mapSize / (1024 * 1024), spillCount,
      if (spillCount > 1) "s" else "")
  //logInfo("Test2: %d"
  // .format(spillCount))

  // JR 11/25/14: inserted second code path for using Flash
  if (!flashBlockManager.UseFlash) {
    val (blockId, file) = diskBlockManager.createTempBlock()
    var writer = blockManager.getDiskWriter(blockId, file,
      serializer, fileBufferSize)
    var objectsWritten = 0

    // List of batch sizes (bytes) in the order they are
      written to disk
    val batchSizes = new ArrayBuffer[Long]

    // Flush the disk writer's contents to disk, and update
      relevant variables
    def flush() = {
      val w = writer
      writer = null
      w.commitAndClose()
      val bytesWritten = w.bytesWritten
      batchSizes.append(bytesWritten)
      _diskBytesSpilled += bytesWritten
      objectsWritten = 0
    }

    var success = false
    try {
      val it = currentMap.destructiveSortedIterator(
        keyComparator)
      while (it.hasNext) {
        val kv = it.next()
        writer.write(kv)
        objectsWritten += 1
      }
    } catch {
      case _ => success = false
    }
  }
}

```

```

        if (objectsWritten == serializerBatchSize) {
            flush()
            writer = blockManager.getDiskWriter(blockId, file,
                serializer, fileBufferSize)
        }
    }
    if (objectsWritten > 0) {
        flush()
    } else if (writer != null) {
        val w = writer
        writer = null
        w.revertPartialWritesAndClose()
    }
    success = true
} finally {
    if (!success) {
        // This code path only happens if an exception was
        // thrown above before we set success;
        // close our stuff and let the exception be thrown
        // further
        if (writer != null) {
            writer.revertPartialWritesAndClose()
        }
        if (file.exists()) {
            file.delete()
        }
    }
}

currentMap = new SizeTrackingAppendOnlyMap[K, C]
spilledMaps.append(new DiskMapIterator(file, blockId,
    batchSizes))
} else {
    val blockId = flashBlockManager.newBlockId();
    var writer = blockManager.getFlashWriter(blockId,
        serializer, fileBufferSize)
    var objectsWritten = 0

    // List of batch sizes (bytes) in the order they are
    // written to disk
    val batchSizes = new ArrayBuffer[Long]

    // Flush the disk writer's contents to disk, and update
    // relevant variables
    def flush() = {

```

```

    val w = writer
    writer = null
    w.commitAndClose()
    val bytesWritten = w.bytesWritten
    batchSizes.append(bytesWritten)
    _diskBytesSpilled += bytesWritten
    objectsWritten = 0
}

var success = false
try {
    val it = currentMap.destructiveSortedIterator(
        keyComparator)
    while (it.hasNext) {
        val kv = it.next()
        writer.write(kv)
        objectsWritten += 1

        if (objectsWritten == serializerBatchSize) {
            flush()
            writer = blockManager.getFlashWriter(blockId,
                serializer, fileBufferSize)
        }
    }
    if (objectsWritten > 0) {
        flush()
    } else if (writer != null) {
        val w = writer
        writer = null
        w.revertPartialWritesAndClose()
    }
    success = true
} finally {
    if (!success) {
        // This code path only happens if an exception was
        // thrown above before we set success;
        // close our stuff and let the exception be thrown
        // further
        if (writer != null) {
            writer.revertPartialWritesAndClose()
        }
        // TODO: deal with deleting the block...
    }
}

```

```

        currentMap = new SizeTrackingAppendOnlyMap[K, C]
        spilledMaps.append(new FlashMapIterator(
            blockManager, ser, blockId, batchSizes,
            serializerBatchSize))
    }

    // Release our memory back to the shuffle pool so that
    // other threads can grab it
    shuffleMemoryManager.release(myMemoryThreshold)
    myMemoryThreshold = 0L

    elementsRead = 0
    _memoryBytesSpilled += mapSize
}

def memoryBytesSpilled: Long = _memoryBytesSpilled
def diskBytesSpilled: Long = _diskBytesSpilled

/**
 * Return an iterator that merges the in-memory map with the
 * spilled maps.
 * If no spill has occurred, simply return the in-memory map's
 * iterator.
 */
override def iterator: Iterator[(K, C)] = {
    if (spilledMaps.isEmpty) {
        currentMap.iterator
    } else {
        new ExternalIterator()
    }
}

/**
 * An iterator that sort-merges (K, C) pairs from the in-
 * memory map and the spilled maps
 */
private class ExternalIterator extends Iterator[(K, C)] {

    // A queue that maintains a buffer for each stream we are
    // currently merging
    // This queue maintains the invariant that it only contains
    // non-empty buffers
    private val mergeHeap = new mutable.PriorityQueue[
        StreamBuffer]

```

```

// Input streams are derived both from the in-memory map
// and spilled maps on disk
// The in-memory map is sorted in place, while the spilled
// maps are already in sorted order
private val sortedMap = currentMap.
  destructiveSortedIterator(keyComparator)
private val inputStreams = (Seq(sortedMap) ++ spilledMaps).
  map(it => it.buffered)

inputStreams.foreach { it =>
  val kcPairs = new ArrayBuffer[(K, C)]
  readNextHashCode(it, kcPairs)
  if (kcPairs.length > 0) {
    mergeHeap.enqueue(new StreamBuffer(it, kcPairs))
  }
}

/**
 * Fill a buffer with the next set of keys with the same
 * hash code from a given iterator. We
 * read streams one hash code at a time to ensure we don't
 * miss elements when they are merged.
 *
 * Assumes the given iterator is in sorted order of hash
 * code.
 *
 * @param it iterator to read from
 * @param buf buffer to write the results into
 */
private def readNextHashCode(it: BufferedIterator[(K, C)],
  buf: ArrayBuffer[(K, C)]): Unit = {
  if (it.hasNext) {
    var kc = it.next()
    buf += kc
    val minHash = hashKey(kc)
    while (it.hasNext && it.head._1.hashCode() == minHash)
    {
      kc = it.next()
      buf += kc
    }
  }
}

/**

```

```

    * If the given buffer contains a value for the given key,
      merge that value into
    * baseCombiner and remove the corresponding (K, C) pair
      from the buffer.
    */
private def mergeIfKeyExists(key: K, baseCombiner: C,
    buffer: StreamBuffer): C = {
    var i = 0
    while (i < buffer.pairs.length) {
        val pair = buffer.pairs(i)
        if (pair._1 == key) {
            // Note that there's at most one pair in the buffer
            // with a given key, since we always
            // merge stuff in a map before spilling, so it's safe
            // to return after the first we find
            removeFromBuffer(buffer.pairs, i)
            return mergeCombiners(baseCombiner, pair._2)
        }
        i += 1
    }
    baseCombiner
}

/**
 * Remove the index'th element from an ArrayBuffer in
 * constant time, swapping another element
 * into its place. This is more efficient than the
 * ArrayBuffer.remove method because it does
 * not have to shift all the elements in the array over. It
 * works for our array buffers because
 * we don't care about the order of elements inside, we
 * just want to search them for a key.
 */
private def removeFromBuffer[T](buffer: ArrayBuffer[T],
    index: Int): T = {
    val elem = buffer(index)
    buffer(index) = buffer(buffer.size - 1) // This also
        works if index == buffer.size - 1
    buffer.reduceToSize(buffer.size - 1)
    elem
}

/**
 * Return true if there exists an input stream that still
 * has unvisited pairs.

```

```

*/
override def hasNext: Boolean = mergeHeap.length > 0

/**
 * Select a key with the minimum hash, then combine all
 * values with the same key from all
 * input streams.
 */
override def next(): (K, C) = {
  if (mergeHeap.length == 0) {
    throw new NoSuchElementException
  }
  // Select a key from the StreamBuffer that holds the
  // lowest key hash
  val minBuffer = mergeHeap.dequeue()
  val minPairs = minBuffer.pairs
  val minHash = minBuffer.minKeyHash
  val minPair = removeFromBuffer(minPairs, 0)
  val minKey = minPair._1
  var minCombiner = minPair._2
  assert(hashKey(minPair) == minHash)

  // For all other streams that may have this key (i.e.
  // have the same minimum key hash),
  // merge in the corresponding value (if any) from that
  // stream
  val mergedBuffers = ArrayBuffer[StreamBuffer](minBuffer)
  while (mergeHeap.length > 0 && mergeHeap.head.minKeyHash
    == minHash) {
    val newBuffer = mergeHeap.dequeue()
    minCombiner = mergeIfKeyExists(minKey, minCombiner,
      newBuffer)
    mergedBuffers += newBuffer
  }

  // Repopulate each visited stream buffer and add it back
  // to the queue if it is non-empty
  mergedBuffers.foreach { buffer =>
    if (buffer.isEmpty) {
      readNextHashCode(buffer.iterator, buffer.pairs)
    }
    if (!buffer.isEmpty) {
      mergeHeap.enqueue(buffer)
    }
  }
}

```

```

    (minKey, minCombiner)
  }

  /**
   * A buffer for streaming from a map iterator (in-memory or
   *   on-disk) sorted by key hash.
   * Each buffer maintains all of the key-value pairs with
   *   what is currently the lowest hash
   * code among keys in the stream. There may be multiple
   *   keys if there are hash collisions.
   * Note that because when we spill data out, we only spill
   *   one value for each key, there is
   * at most one element for each key.
   *
   * StreamBuffers are ordered by the minimum key hash
   *   currently available in their stream so
   * that we can put them into a heap and sort that.
   */
  private class StreamBuffer(
    val iterator: BufferedIterator[(K, C)],
    val pairs: ArrayBuffer[(K, C)])
    extends Comparable[StreamBuffer] {

    def isEmpty = pairs.length == 0

    // Invalid if there are no more pairs in this stream
    def minKeyHash: Int = {
      assert(pairs.length > 0)
      hashKey(pairs.head)
    }

    override def compareTo(other: StreamBuffer): Int = {
      // descending order because mutable.PriorityQueue
      // dequeues the max, not the min
      if (other.minKeyHash < minKeyHash) -1 else if (other.
        minKeyHash == minKeyHash) 0 else 1
    }
  }
}

/**
 * An iterator that returns (K, C) pairs in sorted order from
 *   an on-disk map
 */

```

```

private class DiskMapIterator(file: File, blockId: BlockId,
    batchSize: ArrayBuffer[Long])
    //extends Iterator[(K, C)] {
    extends AbstractMapIterator [K, C] {
    private val batchOffsets = batchSize.scanLeft(0L)(_ + _)
        // Size will be batchSize.length + 1
    assert(file.length() == batchOffsets(batchOffsets.length -
        1))

    private var batchIndex = 0 // Which batch we're in
    private var fileStream: FileInputStream = null

    // An intermediate stream that reads from exactly one batch
    // This guards against pre-fetching and other arbitrary
        behavior of higher level streams
    private var deserializeStream = nextBatchStream()
    private var nextItem: (K, C) = null
    private var objectsRead = 0

    /**
     * Construct a stream that reads only from the next batch.
     */
    private def nextBatchStream(): DeserializationStream = {
        logInfo("@DEBUG@ I AM HERE, BATCH OFFSETS: " +
            batchOffsets +
            " AND BYTES=" + null + " AND BATCH INDEX = " +
            batchIndex +
            " BATCH SIZES WERE " + batchSize)

        // Note that batchOffsets.length = numBatches + 1 since
            we did a scan above; check whether
        // we're still in a valid batch.
        if (batchIndex < batchOffsets.length - 1) {
            if (deserializeStream != null) {
                deserializeStream.close()
                fileStream.close()
                deserializeStream = null
                fileStream = null
            }

            val start = batchOffsets(batchIndex)
            fileStream = new FileInputStream(file)
            fileStream.getChannel().position(start)
            batchIndex += 1

```

```

    val end = batchOffsets(batchIndex)

    assert(end >= start, "start = " + start + ", end = " +
        end +
        ", batchOffsets = " + batchOffsets.mkString("[", ", ",
            ", "]"))

    val bufferedStream = new BufferedInputStream(
        ByteStreams.limit(fileStream, end - start))
    val compressedStream = blockManager.wrapForCompression(
        blockId, bufferedStream)
    ser.deserializeStream(compressedStream)
  } else {
    // No more batches left
    cleanup()
    null
  }
}

/**
 * Return the next (K, C) pair from the deserialization
 * stream.
 *
 * If the current batch is drained, construct a stream for
 * the next batch and read from it.
 * If no more pairs are left, return null.
 */
private def readNextItem(): (K, C) = {
  try {
    val item = deserializeStream.readObject().asInstanceOfOf
      [(K, C)]
    objectsRead += 1
    if (objectsRead == serializerBatchSize) {
      objectsRead = 0
      deserializeStream = nextBatchStream()
    }
    item
  } catch {
    case e: EOFException =>
      cleanup()
      null
  }
}

override def hasNext: Boolean = {

```

```

    if (nextItem == null) {
      if (deserializeStream == null) {
        return false
      }
      nextItem = readNextItem()
    }
    nextItem != null
  }

  override def next(): (K, C) = {
    val item = if (nextItem == null) readNextItem() else
      nextItem
    if (item == null) {
      throw new NoSuchElementException
    }
    nextItem = null
    item
  }

  // TODO: Ensure this gets called even if the iterator isn't
  // drained.
  private def cleanup() {
    batchIndex = batchOffsets.length // Prevent reading any
      other batch
    val ds = deserializeStream
    deserializeStream = null
    fileStream = null
    ds.close()
    file.delete()
  }
}

/** Convenience function to hash the given (K, C) pair by the
  key. */
private def hashKey(kc: (K, C)): Int = ExternalAppendOnlyMap.
  hash(kc._1)
}

private[spark] object ExternalAppendOnlyMap {

  /**
   * Return the hash code of the given object. If the object is
   * null, return a special hash code.
   */
  private def hash[T](obj: T): Int = {

```

```

    if (obj == null) 0 else obj.hashCode()
  }

  /**
   * A comparator which sorts arbitrary keys based on their
   * hash codes.
   */
  private class HashComparator[K] extends Comparator[K] {
    def compare(key1: K, key2: K): Int = {
      val hash1 = hash(key1)
      val hash2 = hash(key2)
      if (hash1 < hash2) -1 else if (hash1 == hash2) 0 else 1
    }
  }
}

```

B.3 ExternalSorter.scala

Source: <https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/util/c>

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more
 * contributor license agreements. See the NOTICE file
 * distributed with
 * this work for additional information regarding copyright
 * ownership.
 * The ASF licenses this file to You under the Apache License,
 * Version 2.0
 * (the "License"); you may not use this file except in
 * compliance with
 * the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing,
 * software
 * distributed under the License is distributed on an "AS IS"
 * BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
 * or implied.
 * See the License for the specific language governing
 * permissions and
 * limitations under the License.
 */

```

```

package org.apache.spark.util.collection

import java.io._
import java.util.Comparator

import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable

import com.google.common.io.ByteStreams

import org.apache.spark.{Aggregator, SparkEnv, Logging,
  Partitioner}
import org.apache.spark.serializer.{DeserializationStream,
  Serializer}
import org.apache.spark.storage.BlockId

/**
 * Sorts and potentially merges a number of key-value pairs of
 * type (K, V) to produce key-combiner
 * pairs of type (K, C). Uses a Partitioner to first group the
 * keys into partitions, and then
 * optionally sorts keys within each partition using a custom
 * Comparator. Can output a single
 * partitioned file with a different byte range for each
 * partition, suitable for shuffle fetches.
 *
 * If combining is disabled, the type C must equal V — we'll
 * cast the objects at the end.
 *
 * @param aggregator optional Aggregator with combine functions
 *   to use for merging data
 * @param partitioner optional Partitioner; if given, sort by
 *   partition ID and then key
 * @param ordering optional Ordering to sort keys within each
 *   partition; should be a total ordering
 * @param serializer serializer to use when spilling to disk
 *
 * Note that if an Ordering is given, we'll always sort using
 * it, so only provide it if you really
 * want the output keys to be sorted. In a map task without map
 * -side combine for example, you
 * probably want to pass None as the ordering to avoid extra
 * sorting. On the other hand, if you do

```

```

* want to do combining, having an Ordering is more efficient
  than not having it.
*
* At a high level, this class works as follows:
*
* - We repeatedly fill up buffers of in-memory data, using
  either a SizeTrackingAppendOnlyMap if
* we want to combine by key, or an simple SizeTrackingBuffer
  if we don't. Inside these buffers,
* we sort elements of type ((Int, K), C) where the Int is
  the partition ID. This is done to
* avoid calling the partitioner multiple times on the same
  key (e.g. for RangePartitioner).
*
* - When each buffer reaches our memory limit, we spill it to
  a file. This file is sorted first
* by partition ID and possibly second by key or by hash code
  of the key, if we want to do
* aggregation. For each file, we track how many objects were
  in each partition in memory, so we
* don't have to write out the partition ID for every element
  .
*
* - When the user requests an iterator, the spilled files are
  merged, along with any remaining
* in-memory data, using the same sort order defined above (
  unless both sorting and aggregation
* are disabled). If we need to aggregate by key, we either
  use a total ordering from the
* ordering parameter, or read the keys with the same hash
  code and compare them with each other
* for equality to merge values.
*
* - Users are expected to call stop() at the end to delete all
  the intermediate files.
*/
private[spark] class ExternalSorter[K, V, C](
  aggregator: Option[Aggregator[K, V, C]] = None,
  partitioner: Option[Partitioner] = None,
  ordering: Option[Ordering[K]] = None,
  serializer: Option[Serializer] = None) extends Logging {

  private val numPartitions = partitioner.map(_.numPartitions).
    getOrElse(1)
  private val shouldPartition = numPartitions > 1

```

```

private val blockManager = SparkEnv.get.blockManager
private val diskBlockManager = blockManager.diskBlockManager
private val shuffleMemoryManager = SparkEnv.get.
    shuffleMemoryManager
private val ser = Serializer.getSerializer(serializer)
private val serInstance = ser.newInstance()

private val conf = SparkEnv.get.conf
private val spillingEnabled = conf.getBoolean("spark.shuffle.
    spill", true)
private val fileBufferSize = conf.getInt("spark.shuffle.file.
    buffer.kb", 100) * 1024

// Size of object batches when reading/writing from
// serializers.
//
// Objects are written in batches, with each batch using its
// own serialization stream. This
// cuts down on the size of reference-tracking maps
// constructed when deserializing a stream.
//
// NOTE: Setting this too low can cause excessive copying
// when serializing, since some serializers
// grow internal data structures by growing + copying every
// time the number of objects doubles.
private val serializerBatchSize = conf.getLong("spark.shuffle
    .spill.batchSize", 10000)

private def getPartition(key: K): Int = {
    if (shouldPartition) partitioner.get.getPartition(key) else
        0
}

// Data structures to store in-memory objects before we spill
// . Depending on whether we have an
// Aggregator set, we either put objects into an
// AppendOnlyMap where we combine them, or we
// store them in an array buffer.
private var map = new SizeTrackingAppendOnlyMap[(Int, K), C]
private var buffer = new SizeTrackingPairBuffer[(Int, K), C]

// Number of pairs read from input since last spill; note
// that we count them even if a value is

```

```

// merged with a previous key in case we're doing something
// like groupBy where the result grows
private var elementsRead = 0L

// What threshold of elementsRead we start estimating map
// size at.
private val trackMemoryThreshold = 1000

// Spilling statistics
private var spillCount = 0
private var _memoryBytesSpilled = 0L
private var _diskBytesSpilled = 0L

// How much of the shared memory pool this collection has
// claimed
private var myMemoryThreshold = 0L

// A comparator for keys K that orders them within a
// partition to allow aggregation or sorting.
// Can be a partial ordering by hash code if a total ordering
// is not provided through by the
// user. (A partial ordering means that equal keys have
// comparator.compare(k, k) = 0, but some
// non-equal keys also have this, so we need to do a later
// pass to find truly equal keys).
// Note that we ignore this if no aggregator and no ordering
// are given.
private val keyComparator: Comparator[K] = ordering.getOrElse
  (new Comparator[K] {
    override def compare(a: K, b: K): Int = {
      val h1 = if (a == null) 0 else a.hashCode()
      val h2 = if (b == null) 0 else b.hashCode()
      h1 - h2
    }
  })

// A comparator for (Int, K) elements that orders them by
// partition and then possibly by key
private val partitionKeyComparator: Comparator[(Int, K)] = {
  if (ordering.isDefined || aggregator.isDefined) {
    // Sort by partition ID then key comparator
    new Comparator[(Int, K)] {
      override def compare(a: (Int, K), b: (Int, K)): Int = {
        val partitionDiff = a._1 - b._1
        if (partitionDiff != 0) {

```

```

        partitionDiff
    } else {
        keyComparator.compare(a._2, b._2)
    }
}
}
} else {
    // Just sort it by partition ID
    new Comparator[(Int, K)] {
        override def compare(a: (Int, K), b: (Int, K)): Int = {
            a._1 - b._1
        }
    }
}
}

// Information about a spilled file. Includes sizes in bytes
// of "batches" written by the
// serializer as we periodically reset its stream, as well as
// number of elements in each
// partition, used to efficiently keep track of partitions
// when merging.
private[this] case class SpilledFile(
    file: File,
    blockId: BlockId,
    serializerBatchSizes: Array[Long],
    elementsPerPartition: Array[Long])
private val spills = new ArrayBuffer[SpilledFile]

def write(records: Iterator[_ <: Product2[K, V]]): Unit = {
    // TODO: stop combining if we find that the reduction
    // factor isn't high
    val shouldCombine = aggregator.isDefined

    if (shouldCombine) {
        // Combine values in-memory first using our AppendOnlyMap
        val mergeValue = aggregator.get.mergeValue
        val createCombiner = aggregator.get.createCombiner
        var kv: Product2[K, V] = null
        val update = (hadValue: Boolean, oldValue: C) => {
            if (hadValue) mergeValue(oldValue, kv._2) else
                createCombiner(kv._2)
        }
        while (records.hasNext) {
            elementsRead += 1

```

```

        kv = records.next()
        map.changeValue((getPartition(kv._1), kv._1), update)
        maybeSpill(usingMap = true)
    }
  } else {
    // Stick values into our buffer
    while (records.hasNext) {
      elementsRead += 1
      val kv = records.next()
      buffer.insert((getPartition(kv._1), kv._1), kv._2.
        asInstanceOf[C])
      maybeSpill(usingMap = false)
    }
  }
}

/**
 * Spill the current in-memory collection to disk if needed.
 *
 * @param usingMap whether we're using a map or buffer as our
 *   current in-memory collection
 */
private def maybeSpill(usingMap: Boolean): Unit = {
  if (!spillingEnabled) {
    return
  }

  val collection: SizeTrackingPairCollection[(Int, K), C] =
    if (usingMap) map else buffer

  // TODO: factor this out of both here and
  // ExternalAppendOnlyMap
  if (elementsRead > trackMemoryThreshold && elementsRead %
    32 == 0 &&
    collection.estimateSize() >= myMemoryThreshold)
  {
    // Claim up to double our current memory from the shuffle
    // memory pool
    val currentMemory = collection.estimateSize()
    val amountToRequest = 2 * currentMemory -
      myMemoryThreshold
    val granted = shuffleMemoryManager.tryToAcquire(
      amountToRequest)
    myMemoryThreshold += granted
  }
}
//ADDED

```

```

logInfo("***** Called because %d <= %d *****"
    .format(myMemoryThreshold, currentMemory))
    if (myMemoryThreshold <= currentMemory) {
        // We were granted too little memory to grow further (
            either tryToAcquire returned 0,
        // or we already had more memory than myMemoryThreshold
        ); spill the current collection
        spill(currentMemory, usingMap) // Will also release
            memory back to ShuffleMemoryManager
    }
}

/**
 * Spill the current in-memory collection to disk, adding a
 * new file to spills, and clear it.
 *
 * @param usingMap whether we're using a map or buffer as our
 * current in-memory collection
 */
private def spill(memorySize: Long, usingMap: Boolean): Unit
    = {
        val collection: SizeTrackingPairCollection[(Int, K), C] =
            if (usingMap) map else buffer
        val memorySize = collection.estimateSize()

        spillCount += 1
        val threadId = Thread.currentThread().getId
        logInfo("Thread %d spilling in-memory batch of %d MB to
            disk (%d spill%s so far)"
            .format(threadId, memorySize / (1024 * 1024), spillCount,
                if (spillCount > 1) "s" else ""))
        logInfo("Test: (spillcount: %D)"
            .format(spillCount))
        val (blockId, file) = diskBlockManager.createTempBlock()
        var writer = blockManager.getDiskWriter(blockId, file, ser,
            fileBufferSize)
        var objectsWritten = 0 // Objects written since the last
            flush

        // List of batch sizes (bytes) in the order they are
        // written to disk
        val batchSizes = new ArrayBuffer[Long]

        // How many elements we have in each partition

```

```

val elementsPerPartition = new Array[Long](numPartitions)

// Flush the disk writer's contents to disk, and update
// relevant variables.
// The writer is closed at the end of this process, and
// cannot be reused.
def flush() = {
  val w = writer
  writer = null
  w.commitAndClose()
  val bytesWritten = w.bytesWritten
  batchSizes.append(bytesWritten)
  _diskBytesSpilled += bytesWritten
  objectsWritten = 0
}

var success = false
try {
  val it = collection.destructiveSortedIterator(
    partitionKeyComparator)
  while (it.hasNext) {
    val elem = it.next()
    val partitionId = elem._1._1
    val key = elem._1._2
    val value = elem._2
    writer.write(key)
    writer.write(value)
    elementsPerPartition(partitionId) += 1
    objectsWritten += 1

    if (objectsWritten == serializerBatchSize) {
      flush()
      writer = blockManager.getDiskWriter(blockId, file,
        ser, fileBufferSize)
    }
  }
  if (objectsWritten > 0) {
    flush()
  } else if (writer != null) {
    val w = writer
    writer = null
    w.revertPartialWritesAndClose()
  }
  success = true
} finally {

```

```

    if (!success) {
        // This code path only happens if an exception was
        // thrown above before we set success;
        // close our stuff and let the exception be thrown
        // further
        if (writer != null) {
            writer.revertPartialWritesAndClose()
        }
        if (file.exists()) {
            file.delete()
        }
    }
}

if (usingMap) {
    map = new SizeTrackingAppendOnlyMap[(Int, K), C]
} else {
    buffer = new SizeTrackingPairBuffer[(Int, K), C]
}

// Release our memory back to the shuffle pool so that
// other threads can grab it
shuffleMemoryManager.release(myMemoryThreshold)
myMemoryThreshold = 0

spills.append(SpilledFile(file, blockId, batchSizes.toArray
    , elementsPerPartition))
_memoryBytesSpilled += memorySize
}

/**
 * Merge a sequence of sorted files, giving an iterator over
 * partitions and then over elements
 * inside each partition. This can be used to either write
 * out a new file or return data to
 * the user.
 *
 * Returns an iterator over all the data written to this
 * object, grouped by partition. For each
 * partition we then have an iterator over its contents, and
 * these are expected to be accessed
 * in order (you can't "skip ahead" to one partition without
 * reading the previous one).
 * Guaranteed to return a key-value pair for each partition,
 * in order of partition ID.

```

```

*/
private def merge(spills: Seq[SpilledFile], inMemory:
  Iterator[((Int, K), C)])
  : Iterator[(Int, Iterator[Product2[K, C]])] = {
  val readers = spills.map(new SpillReader(_))
  val inMemBuffered = inMemory.buffered
  (0 until numPartitions).iterator.map { p =>
    val inMemIterator = new IteratorForPartition(p,
      inMemBuffered)
    val iterators = readers.map(_.readNextPartition()) ++ Seq
      (inMemIterator)
    if (aggregator.isDefined) {
      // Perform partial aggregation across partitions
      (p, mergeWithAggregation(
        iterators, aggregator.get.mergeCombiners,
        keyComparator, ordering.isDefined))
    } else if (ordering.isDefined) {
      // No aggregator given, but we have an ordering (e.g.
        used by reduce tasks in sortByKey);
      // sort the elements without trying to merge them
      (p, mergeSort(iterators, ordering.get))
    } else {
      (p, iterators.iterator.flatten)
    }
  }
}

/**
 * Merge-sort a sequence of (K, C) iterators using a given a
 * comparator for the keys.
 */
private def mergeSort(iterators: Seq[Iterator[Product2[K, C]
  ]]), comparator: Comparator[K])
  : Iterator[Product2[K, C]] =
{
  val bufferedIters = iterators.filter(_.hasNext).map(_.
    buffered)
  type Iter = BufferedIterator[Product2[K, C]]
  val heap = new mutable.PriorityQueue[Iter]() (new Ordering[
    Iter] {
    // Use the reverse of comparator.compare because
    PriorityQueue dequeues the max
    override def compare(x: Iter, y: Iter): Int = -comparator
      .compare(x.head._1, y.head._1)
  })
}

```

```

heap.enqueue(bufferedIters: _*) // Will contain only the
    iterators with hasNext = true
new Iterator[Product2[K, C]] {
    override def hasNext: Boolean = !heap.isEmpty

    override def next(): Product2[K, C] = {
        if (!hasNext) {
            throw new NoSuchElementException
        }
        val firstBuf = heap.dequeue()
        val firstPair = firstBuf.next()
        if (firstBuf.hasNext) {
            heap.enqueue(firstBuf)
        }
        firstPair
    }
}

/**
 * Merge a sequence of (K, C) iterators by aggregating values
 *   for each key, assuming that each
 * iterator is sorted by key with a given comparator. If the
 *   comparator is not a total ordering
 * (e.g. when we sort objects by hash code and different keys
 *   may compare as equal although
 * they're not), we still merge them by doing equality tests
 *   for all keys that compare as equal.
 */
private def mergeWithAggregation(
    iterators: Seq[Iterator[Product2[K, C]]],
    mergeCombiners: (C, C) => C,
    comparator: Comparator[K],
    totalOrder: Boolean)
    : Iterator[Product2[K, C]] =
{
    if (!totalOrder) {
        // We only have a partial ordering, e.g. comparing the
        //   keys by hash code, which means that
        // multiple distinct keys might be treated as equal by
        //   the ordering. To deal with this, we
        // need to read all keys considered equal by the ordering
        //   at once and compare them.
        new Iterator[Iterator[Product2[K, C]]] {
            val sorted = mergeSort(iterators, comparator).buffered

```

```

// Buffers reused across elements to decrease memory
// allocation
val keys = new ArrayBuffer[K]
val combiners = new ArrayBuffer[C]

override def hasNext: Boolean = sorted.hasNext

override def next(): Iterator[Product2[K, C]] = {
  if (!hasNext) {
    throw new NoSuchElementException
  }
  keys.clear()
  combiners.clear()
  val firstPair = sorted.next()
  keys += firstPair._1
  combiners += firstPair._2
  val key = firstPair._1
  while (sorted.hasNext && comparator.compare(sorted.
    head._1, key) == 0) {
    val pair = sorted.next()
    var i = 0
    var foundKey = false
    while (i < keys.size && !foundKey) {
      if (keys(i) == pair._1) {
        combiners(i) = mergeCombiners(combiners(i),
          pair._2)
        foundKey = true
      }
      i += 1
    }
    if (!foundKey) {
      keys += pair._1
      combiners += pair._2
    }
  }
}

// Note that we return an iterator of elements since
// we could've had many keys marked
// equal by the partial order; we flatten this below
// to get a flat iterator of (K, C).
keys.iterator.zip(combiners.iterator)
}
}.flatMap(i => i)
} else {

```

```

// We have a total ordering, so the objects with the same
// key are sequential.
new Iterator[Product2[K, C]] {
  val sorted = mergeSort(iterators, comparator).buffered

  override def hasNext: Boolean = sorted.hasNext

  override def next(): Product2[K, C] = {
    if (!hasNext) {
      throw new NoSuchElementException
    }
    val elem = sorted.next()
    val k = elem._1
    var c = elem._2
    while (sorted.hasNext && sorted.head._1 == k) {
      c = mergeCombiners(c, sorted.head._2)
    }
    (k, c)
  }
}

}

/**
 * An internal class for reading a spilled file partition by
 * partition. Expects all the
 * partitions to be requested in order.
 */
private[this] class SpillReader(spill: SpilledFile) {
  // Serializer batch offsets; size will be batchSize.length
  // + 1
  val batchOffsets = spill.serializerBatchSizes.scanLeft(0L)(
    _ + _)

  // Track which partition and which batch stream we're in.
  // These will be the indices of
  // the next element we will read. We'll also store the last
  // partition read so that
  // readNextPartition() can figure out what partition that
  // was from.
  var partitionId = 0
  var indexInPartition = 0L
  var batchId = 0
  var indexInBatch = 0
  var lastPartitionId = 0

```

```

skipToNextPartition()

// Intermediate file and deserializer streams that read
// from exactly one batch
// This guards against pre-fetching and other arbitrary
// behavior of higher level streams
var fileStream: FileInputStream = null
var deserializeStream = nextBatchStream() // Also sets
    fileStream

var nextItem: (K, C) = null
var finished = false

/** Construct a stream that only reads from the next batch
    */
def nextBatchStream(): DeserializationStream = {
    // Note that batchOffsets.length = numBatches + 1 since
    // we did a scan above; check whether
    // we're still in a valid batch.
    if (batchId < batchOffsets.length - 1) {
        if (deserializeStream != null) {
            deserializeStream.close()
            fileStream.close()
            deserializeStream = null
            fileStream = null
        }

        val start = batchOffsets(batchId)
        fileStream = new FileInputStream(spill.file)
        fileStream.getChannel.position(start)
        batchId += 1

        val end = batchOffsets(batchId)

        assert(end >= start, "start = " + start + ", end = " +
            end +
            ", batchOffsets = " + batchOffsets.mkString("[", ", ",
                ", "]"))

        val bufferedStream = new BufferedInputStream(
            ByteStreams.limit(fileStream, end - start))
        val compressedStream = blockManager.wrapForCompression(
            spill.blockId, bufferedStream)

```

```

        serInstance.deserializeStream(compressedStream)
    } else {
        // No more batches left
        cleanup()
        null
    }
}

/**
 * Update partitionId if we have reached the end of our
 * current partition, possibly skipping
 * empty partitions on the way.
 */
private def skipToNextPartition() {
    while (partitionId < numPartitions &&
           indexInPartition == spill.elementsPerPartition(
               partitionId)) {
        partitionId += 1
        indexInPartition = 0L
    }
}

/**
 * Return the next (K, C) pair from the deserialization
 * stream and update partitionId,
 * indexInPartition, indexInBatch and such to match its
 * location.
 *
 * If the current batch is drained, construct a stream for
 * the next batch and read from it.
 * If no more pairs are left, return null.
 */
private def readNextItem(): (K, C) = {
    if (finished || deserializeStream == null) {
        return null
    }
    val k = deserializeStream.readObject().asInstanceOf[K]
    val c = deserializeStream.readObject().asInstanceOf[C]
    lastPartitionId = partitionId
    // Start reading the next batch if we're done with this
    one
    indexInBatch += 1
    if (indexInBatch == serializerBatchSize) {
        indexInBatch = 0
        deserializeStream = nextBatchStream()
    }
}

```

```

    }
    // Update the partition location of the element we're
    // reading
    indexInPartition += 1
    skipToNextPartition()
    // If we've finished reading the last partition, remember
    // that we're done
    if (partitionId == numPartitions) {
      finished = true
      if (deserializeStream != null) {
        deserializeStream.close()
      }
    }
  }
  (k, c)
}

var nextPartitionToRead = 0

def readNextPartition(): Iterator[Product2[K, C]] = new
  Iterator[Product2[K, C]] {
    val myPartition = nextPartitionToRead
    nextPartitionToRead += 1

    override def hasNext: Boolean = {
      if (nextItem == null) {
        nextItem = readNextItem()
        if (nextItem == null) {
          return false
        }
      }
    }
    assert(lastPartitionId >= myPartition)
    // Check that we're still in the right partition; note
    // that readNextItem will have returned
    // null at EOF above so we would've returned false
    // there
    lastPartitionId == myPartition
  }

  override def next(): Product2[K, C] = {
    if (!hasNext) {
      throw new NoSuchElementException
    }
    val item = nextItem
    nextItem = null
    item
  }

```

```

    }
  }

  // Clean up our open streams and put us in a state where we
  // can't read any more data
  def cleanup() {
    batchSize = batchOffsets.length // Prevent reading any
    // other batch
    val ds = deserializeStream
    deserializeStream = null
    fileStream = null
    ds.close()
    // NOTE: We don't do file.delete() here because that is
    // done in ExternalSorter.stop().
    // This should also be fixed in ExternalAppendOnlyMap.
  }
}

/**
 * Return an iterator over all the data written to this
 * object, grouped by partition and
 * aggregated by the requested aggregator. For each partition
 * we then have an iterator over its
 * contents, and these are expected to be accessed in order (
 * you can't "skip ahead" to one
 * partition without reading the previous one). Guaranteed to
 * return a key-value pair for each
 * partition, in order of partition ID.
 *
 * For now, we just merge all the spilled files in once pass,
 * but this can be modified to
 * support hierarchical merging.
 */
def partitionedIterator: Iterator[(Int, Iterator[Product2[K,
C]])] = {
  val usingMap = aggregator.isDefined
  val collection: SizeTrackingPairCollection[(Int, K), C] =
    if (usingMap) map else buffer
  if (spills.isEmpty) {
    // Special case: if we have only in-memory data, we don't
    // need to merge streams, and perhaps
    // we don't even need to sort by anything other than
    // partition ID
    if (!ordering.isDefined) {

```

```

    // The user isn't requested sorted keys, so only sort
    // by partition ID, not key
    val partitionComparator = new Comparator[(Int, K)] {
      override def compare(a: (Int, K), b: (Int, K)): Int =
        {
          a._1 - b._1
        }
    }
    groupByPartition(collection.destructiveSortedIterator(
      partitionComparator))
  } else {
    // We do need to sort by both partition ID and key
    groupByPartition(collection.destructiveSortedIterator(
      partitionKeyComparator))
  }
} else {
  // General case: merge spilled and in-memory data
  merge(spills, collection.destructiveSortedIterator(
    partitionKeyComparator))
}
}

/**
 * Return an iterator over all the data written to this
 * object, aggregated by our aggregator.
 */
def iterator: Iterator[Product2[K, C]] = partitionedIterator.
  flatMap(pair => pair._2)

def stop(): Unit = {
  spills.foreach(s => s.file.delete())
  spills.clear()
}

def memoryBytesSpilled: Long = _memoryBytesSpilled

def diskBytesSpilled: Long = _diskBytesSpilled

/**
 * Given a stream of ((partition, key), combiner) pairs *
 * assumed to be sorted by partition ID*,
 * group together the pairs for each partition into a sub-
 * iterator.
 */

```

```

    * @param data an iterator of elements, assumed to already be
      sorted by partition ID
    */
private def groupByPartition(data: Iterator[((Int, K), C)])
    : Iterator[(Int, Iterator[Product2[K, C]])] =
{
    val buffered = data.buffered
    (0 until numPartitions).iterator.map(p => (p, new
        IteratorForPartition(p, buffered)))
}

/**
 * An iterator that reads only the elements for a given
   partition ID from an underlying buffered
 * stream, assuming this partition is the next one to be read
   . Used to make it easier to return
 * partitioned iterators from our in-memory collection.
 */
private[this] class IteratorForPartition(partitionId: Int,
    data: BufferedIterator[((Int, K), C)])
    extends Iterator[Product2[K, C]]
{
    override def hasNext: Boolean = data.hasNext && data.head.
        _1._1 == partitionId

    override def next(): Product2[K, C] = {
        if (!hasNext) {
            throw new NoSuchElementException
        }
        val elem = data.next()
        (elem._1._2, elem._2)
    }
}
}

```

B.4 ShuffleMemoryManager.scala

Source: <https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/shuffle>

```

/*
 * Licensed to the Apache Software Foundation (ASF) under one
   or more
 * contributor license agreements. See the NOTICE file
   distributed with

```

```
* this work for additional information regarding copyright
  ownership.
* The ASF licenses this file to You under the Apache License,
  Version 2.0
* (the "License"); you may not use this file except in
  compliance with
* the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
  software
* distributed under the License is distributed on an "AS IS"
  BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
  or implied.
* See the License for the specific language governing
  permissions and
* limitations under the License.
*/

package org.apache.spark.shuffle

import scala.collection.mutable

import org.apache.spark.{Logging, SparkException, SparkConf}

/**
 * Allocates a pool of memory to task threads for use in
 * shuffle operations. Each disk-spilling
 * collection (ExternalAppendOnlyMap or ExternalSorter) used by
 * these tasks can acquire memory
 * from this pool and release it as it spills data out. When a
 * task ends, all its memory will be
 * released by the Executor.
 *
 * This class tries to ensure that each thread gets a
 * reasonable share of memory, instead of some
 * thread ramping up to a large amount first and then causing
 * others to spill to disk repeatedly.
 * If there are N threads, it ensures that each thread can
 * acquire at least 1 / 2N of the memory
 * before it has to spill, and at most 1 / N. Because N varies
 * dynamically, we keep track of the
```

```

* set of active threads and redo the calculations of 1 / 2N
  and 1 / N in waiting threads whenever
* this set changes. This is all done by synchronizing access
  on "this" to mutate state and using
* wait() and notifyAll() to signal changes.
*/
private[spark] class ShuffleMemoryManager(maxMemory: Long)
  extends Logging {
  private val threadMemory = new mutable.HashMap[Long, Long]()
    // threadId -> memory bytes

  def this(conf: SparkConf) = this(ShuffleMemoryManager.
    getMaxMemory(conf))

  /**
   * Try to acquire up to numBytes memory for the current
   * thread, and return the number of bytes
   * obtained, or 0 if none can be allocated. This call may
   * block until there is enough free memory
   * in some situations, to make sure each thread has a chance
   * to ramp up to at least 1 / 2N of the
   * total memory pool (where N is the # of active threads)
   * before it is forced to spill. This can
   * happen if the number of threads increases but an older
   * thread had a lot of memory already.
   */
  def tryToAcquire(numBytes: Long): Long = synchronized {
    val threadId = Thread.currentThread().getId
    assert(numBytes > 0, "invalid number of bytes requested: "
      + numBytes)

    // Add this thread to the threadMemory map just so we can
    // keep an accurate count of the number
    // of active threads, to let other threads ramp down their
    // memory in calls to tryToAcquire
    if (!threadMemory.contains(threadId)) {
      threadMemory(threadId) = 0L
      notifyAll() // Will later cause waiting threads to wake
        up and check numThreads again
    }

    // Keep looping until we're either sure that we don't want
    // to grant this request (because this
    // thread would have more than 1 / numActiveThreads of the
    // memory) or we have enough free

```

```

// memory to give it (we always let each thread get at
  least 1 / (2 * numActiveThreads)).
while (true) {
  val numActiveThreads = threadMemory.keys.size
  val curMem = threadMemory(threadId)
  val freeMemory = maxMemory - threadMemory.values.sum

  // How much we can grant this thread; don't let it grow
  to more than 1 / numActiveThreads
  val maxToGrant = math.min(numBytes, (maxMemory /
    numActiveThreads) - curMem)

  if (curMem < maxMemory / (2 * numActiveThreads)) {
    // We want to let each thread get at least 1 / (2 *
    numActiveThreads) before blocking;
    // if we can't give it this much now, wait for other
    threads to free up memory
    // (this happens if older threads allocated lots of
    memory before N grew)
    if (freeMemory >= math.min(maxToGrant, maxMemory / (2 *
      numActiveThreads) - curMem)) {
      val toGrant = math.min(maxToGrant, freeMemory)
      threadMemory(threadId) += toGrant
      return toGrant
    } else {
      logInfo(s"Thread $threadId waiting for at least 1/2N
        of shuffle memory pool to be free")
      wait()
    }
  } else {
    // Only give it as much memory as is free, which might
    be none if it reached 1 / numThreads
    val toGrant = math.min(maxToGrant, freeMemory)
    threadMemory(threadId) += toGrant
    return toGrant
  }
}
0L // Never reached
}

/** Release numBytes bytes for the current thread. */
def release(numBytes: Long): Unit = synchronized {
  val threadId = Thread.currentThread().getId
  val curMem = threadMemory.getOrElse(threadId, 0L)
  if (curMem < numBytes) {

```

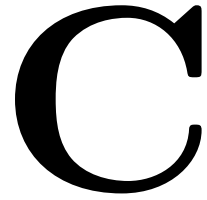
```

        throw new SparkException(
            s"Internal error: release called on ${numBytes} bytes
            but thread only has ${curMem}")
    }
    threadMemory(threadId) -= numBytes
    notifyAll() // Notify waiters who locked "this" in
                tryToAcquire that memory has been freed
}

/** Release all memory for the current thread and mark it as
    inactive (e.g. when a task ends). */
def releaseMemoryForThisThread(): Unit = synchronized {
    val threadId = Thread.currentThread().getId
    threadMemory.remove(threadId)
    notifyAll() // Notify waiters who locked "this" in
                tryToAcquire that memory has been freed
}
}

private object ShuffleMemoryManager {
    /**
     * Figure out the shuffle memory limit from a SparkConf. We
     * currently have both a fraction
     * of the memory pool and a safety factor since collections
     * can sometimes grow bigger than
     * the size we target before we estimate their sizes again.
     */
    def getMaxMemory(conf: SparkConf): Long = {
        val memoryFraction = conf.getDouble("spark.shuffle.
            memoryFraction", 0.2)
        val safetyFraction = conf.getDouble("spark.shuffle.
            safetyFraction", 0.8)
        (Runtime.getRuntime.maxMemory * memoryFraction *
            safetyFraction).toLong
    }
}

```



Code for reference

Enclosed are the manuals to install Ubuntu, Apache Spark and Apache Hadoop on Power8.

C.1 Ubuntu 14.04 Little Endian on Power

The following manual describes how to install Ubuntu 14.04 Little Endian on Power7. For Power8, Sapphire is already installed, and there is no need for virtualization.

```
*****
* This guide consists of three parts:
* 1) Install Sapphire
* 2) Install PowerKVM
* 3) Install Ubuntu 14.04 (as a virtual machine on PowerKVM)
*
* The following sources are used:
* - Sapphire: https://ltc3.linux.ibm.com/wiki/Sapphire/Install
* - Sapphire: http://www-01.ibm.com/support/knowledgecenter/POWER6/iphath/iphblresetserverp6.htm?cp=POWER6%2F2-5-2-5-0-8&lang=en
* - PowerKVM: http://www.redbooks.ibm.com/redpieces/pdfs/sg248231.pdf
*
* The following images are required
* - PowerKVM: Version 17.4 (available within IBM at https://ltc3.linux.ibm.com/wiki/PowerKVM/Builds/Releases/201406031019)
* - Ubuntu 14.04 LTS for Power: http://cdimage.ubuntu.com/releases/14.04/release/ubuntu-14.04-server-ppc64el.iso
* - flash image: We used http://pfd.austin.ibm.com/images/apollo/b0506p\_1420.770/SIGNED/01AL770\_075\_032.img
* based on https://ltc3.linux.ibm.com/wiki/Sapphire/Install
*
* For questions or remarks, please send an email to:
* - Tom Hubregtsen [th@us.ibm.com] or
* - Peter Hofstee [hofstee@us.ibm.com]
*
* The following people were consulted:
* - Peter Hofstee
```

```
* - Paul Clarke
* - Anton Blanchard
* - Jason M Furmanek
* - Bishop Brock
*
```

```
*****
```

```
*****
** 1: Install Sapphire **
*****
```

```
#Check the document if you have the right hardware
Source : https://ltc3.linux.ibm.com/wiki/Sapphire/Install
```

```
#Log into the ASM:
https://<IP>
```

```
#Reset to Manufacturing default config (MDC)
(Following steps from source: http://www-01.ibm.com/support/knowledgecenter/POWER6/iphath/iphblresetserverp6.htm?cp=POWER6%2F2-5-2-5-0-8&lang=en)
```

```
#Get the system in "Standby"
```

```
- Log in to the ASM: in a browser, go to https://IP and accept the certificate
```

```
#Power off
```

```
- "Power/Restart Control -> Power On/Off System"
```

```
# Change startup mode
```

```
- Set "Server firmware start policy" to "Standby (User-initiated)"
```

```
#Power on in the new mode
```

```
- Click "Save settings and power on"
```

```
#Verify it is up in the correct mode
```

```
- Verify by going back to "Power On/Off System" -> Look for "Current system server firmware state: " to be "Not Running", with "Current system power state: On"
```

```
* "Current system server firmware state: Unknown" -> wait and refresh page"
```

```
#From the HMC, initialize the profile data
```

```
- On the HMC in the work pane, select the managed system, click the Tasks button, and click "Configuration > Manage Partition Data > Initialize".
```

```

* Make sure *not* to select a partition in the window "
  Systems Management > Servers > x"

# From the HMC, clear the logical partition configuration data
  on the managed system. Complete the following at your HMC (
    not connected remotely):
- In the navigation pane, click HMC Management.
- In the work pane, click Open Restricted Shell Terminal.
  * This Terminal is not visible from the web interface!
# Type the command: (where managed.system.name is the name of
  the managed system as it is displayed in the work pane.)
- $lpcfgop -m <managed_system_name> -o clear
- Enter 1 to confirm.
  *This step takes several seconds to complete.

#Remove all HMC control
- Go back to the ASM: in a browser: go to https://IP
- System Configuration > Hardware Management Consoles: "There
  are no HMC connections at this time" -> "Reset the server to
  a non-HMC managed configuration"
  * In case there are HMC connections, remove them first. If
    you are not able to remove them from here, write down the
    IP and go to "network services > Network Access" and add
    the IP to the "Denied IP addresses". (May need a reset)
    Now go back and you should be able to see the non-HMC
    message.

#Boot into Linux under PowerVM normally
- In "Power/Restart Control > Power On/Off System", set "Server
  firmware start policy" back to "Running (Auto-Start Always)"
- "Save settings"
- Go back to "Power/Restart Control > Power On/Off System"
- if (Current system power state == on && (Current system
  server firmware state == Unknown)
  -> first do "Save settings and power off"
  -> then "Save settings and power on"
- else if (Current system power state == on && (Current system
  server firmware state == running)
  -> do nothing
- else if (Current system power state == off)
  -> "Save settings and power off"
- Wait until it is on (Current system power state: On &&
  Current system firmware state: Running)

```

```

#Access the FSP
- The source recommends using telnet. This did *NOT* work for
  me:
  - ASM > Network services > Debug Virtual TTY
  - [1, 1, 30]
- I connected to the serial port from a different machine using
  "screen /dev/ttyUSB0 19200"
#Make sure you have the flash image here. I used a Flash disk
  that I mounted
- $update-flash -f <image_file>
#Wait for the machine to reboot. I did it overnight, because
  last time I think it took me 30~60 minutes.
#Then verify that the FSP is back up, and verify that the flash
  was successful
  * At least in my terminal, there is no indication here that
    it came back to life. Last message is "FLASH: this will
    take several minutes. Do not power off!"
  - Access the ASM (https://IP) (it's up)
  - System Configuration > Sapphire Hypervisor Configuration (
    if this option is there, the flash was successful

#Reboot FSP
- "System Service Aids > Reset Service Processor > continue"
#Once it is back up, go back into the ASM and switch to
  Sapphire
- "System configuration > Sapphire Hypervisor Configuration".
  Change "Hypervisor Mode: " from "PowerVM" to "Sapphire"
- On this same page, set a IPMI password, you'll need it in the
  next step
- Press "Continue"

*****
** 2: Install PowerKVM **
*****

#Put the dvd in the disk drive
#(re)boot the machine
#connect either over the physical port, or via ipmitool's SOL
  a) using the serial port in combination with "screen /dev/
    ttyUSB0 19200", as described above
  b) ipmitool -I lanplus -H <IP> -P <password> sol activate (
    This did not work in the previous step because we did not
    have Sapphire yet)
#Install PowerKVM

```

- In petitboot (Which is what you should see after the reboot), select "POWERKVMLIVECD"
 - * If you wait long enough, it seems like petitboot makes the choice for you. Since I had both SOL and ipmi open, ipmi continued and SOL seemed to hang.
 - * So if you end up with "Info: performing kexec reboot", try using ipmi, or be quick enough ;)
- Follow install instructions

Note: You probably want to set up things like network on the PowerKVM. It will not show in your virtual machine until you do it here first.

```
*****
** 3: Install Ubuntu **
*****
```

```
#Reboot the system
#log in
#Create disk space
- $virsh
#What's there already?
- $vol-list default
  * Probably only the "Lost + found"
- $vol-create-as default ubuntuVolume.qcow2 --format qcow2 300G
  --allocation 150G
- $exit
#Verify if you see the disk space, also look where it is
  located
- $vol-list default
- $exit
#Install Ubuntu as a virtual machine
- Put the disk into the drive
- $virt-install --name="<name>" --cdrom=<location> --memory=<
  int> --disk=<location> --vcpus=<int>
  * My test settings: ["Ubuntu",/dev/sr0,100000,/var/lib/
    libvirt/images/ubuntuVolume.qcow2,32]
- Follow install instructions
- Enjoy!
```

```
*****
** 4: Internet **
*****
```

```
*Physically connect 2 ethernet wires to the adapter
```

```

* in powerKVM *
*Make sure both interfaces are up:
sudo ifup enP1p1s0f1/0 or enp160s0f1/0
#now you should be able to see them in ifconfig
#Start the VM
$ virsh start ubuntuTest
virsh attach-interface ubuntuTest bridge virbr0 --live
#Check if the interface card is connected to the bridge
$ brctl show virbr0
#if not, connect it
$ brctl addif virbr0 enP1p1s0f1
#check again
$ brctl show virbr0
#Access the VM and restart it
$ virsh console ubuntu
$ sudo shutdown -r now
#Reconnect to it
$virsh console ubuntu
#View interfaces that you do not see with ifconfig -a
$ sudo lshw -class network
#Add the interface that you do see there (eth0 - eth2), to the
    following path
vim /etc/network/interfaces
->
"auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet dhcp

auto eth2
iface eth2 inet dhcp"
#Now start them
$ sudo ifup eth0
-> In my case, nothing
$ sudo ifup eth1
-> In my case, success
-> P.S. you do now also see them in ifconfig -a

* if it does not start, restart

#Now for ssh:

```

```

$ sudo apt-get install openssh-server
-> if not available, sudo apt-get update
$ sudo service ssh restart

*****
** 5: Adding a disk: **
*****

#create space
$vol-create-as default ubuntuData.qcow2 --format qcow2 500G --
  allocation 500G
#In powerKVM, dump xml:
$ virsh dumpxml ubuntuTest > ~/ubuntuTest.xml
#Open and add the disks
  <disk type='file' device='disk'>
    <driver name='qemu' type='qcow2'/>
    <source file='/var/lib/libvirt/images/ubuntuCode.qcow2'/>
    <target dev='sdn' bus='scsi'/>
    <alias name='scsi0-0-0-2'/>
    <address type='drive' controller='0' bus='0' target='0'
      unit='2'/>
  </disk>
#create VM
$ virsh create ubuntuTest.xml
#Wait for the VM to boot, and log in through SSH
#Check if the disk is added:
sudo lshw -C disk
#Go into fdisk
$ sudo fdisk /dev/sdb
#Create new partition
$ n
#primary partition
$ p
#Only one partition for the disk, so this is number 1
$ 1
#Write out the changes
$ w
#Make them ext4
$ sudo mkfs -t ext4 /dev/sdb1
#Mount
$ sudo mkdir /mnt/name
$ sudo mount /dev/sdb1 /mnt/name

```

C.2 Apache Hadoop on Ubuntu with Power

We used the following commands to build Apache Hadoop 2.2.0, available at <http://apache.mirrors.tds.net/hadoop/common/hadoop-2.2.0/hadoop-2.2.0-src.tar.gz>.

```
sudo apt-get install gcc autoconf automake libtool zlib1g-dev
    maven protobuf-compiler cmake libssl-dev libsnappy-dev
#maven installs maven 3, whereas maven2 installs maven 2.
    Therefor; use maven
wget http://apache.mirrors.tds.net/hadoop/common/hadoop-2.2.0/
    hadoop-2.2.0-src.tar.gz
tar -xzf hadoop-2.4.1-src.tar.gz
mvn package -Pdist,native -DskipTests -Dtar -Drequire.snappy
```

We encountered the following errors:

```
#for
[ERROR] /home/tom/hadoop/hadoop-2.4.1-native/hadoop-2.2.0-src/
    hadoop-common-project/hadoop-auth/src/test/java/org/apache/
    hadoop/security/authentication/client/AuthenticatorTestCase.
    java:[88,11] error: cannot access AbstractLifecycle
#Use this patch
https://issues.apache.org/jira/secure/attachment/12614482/
    HADOOP-10110.patch

#for
[ERROR] /home/tom/hadoop/hadoop-2.4.1-native/hadoop-2.2.0-src/
    hadoop-common-project/hadoop-common/src/test/java/org/apache
    /hadoop/security/ssl/KeyStoreTestUtil.java:[22,24] error:
    cannot find symbol
#go into
vim /home/tom/hadoop/hadoop-2.4.1-native/hadoop-2.2.0-src/
    hadoop-common-project/hadoop-common/src/test/java/org/apache
    /hadoop/security/ssl/KeyStoreTestUtil.java
#and replace all occurances [22-32] of
import sun.security.x509.AlgorithmId;
#with
import com.ibm.security.x509.AlgorithmId;

#for
testSocketIOWithTimeout(org.apache.hadoop.net.
    TestSocketIOWithTimeout) Time elapsed: 5079 sec <<<
    FAILURE!
#Commented out:
fail("Did not throw");
#in file at line at line 157:
```

```

./hadoop-common-project/hadoop-common/src/test/java/org/apache/
  hadoop/net/TestSocketIOWithTimeout.java
#Because of comment in:
vim ./hadoop-common-project/hadoop-common/src/main/java/org/
  apache/hadoop/net/SocketOutputStream.java
#at line 210

#for
/home/tom/hadoop/hadoop-2.4.1-native/hadoop-2.2.0-src/hadoop-
  hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/
  hdfs/tools/offlineEditsViewer/XmlEditsVisitor.java:[32,48]
  error: package com.sun.org.apache.xml.internal.serialize
  does not exist
#Change into
//import com.sun.org.apache.xml.internal.serialize.OutputFormat
  ;
import org.apache.xml.serialize.OutputFormat;
//import com.sun.org.apache.xml.internal.serialize.
  XMLSerializer;
import org.apache.xml.serialize.XMLSerializer;

#for error:
constituent[31]: file:/usr/share/maven/lib/wagon-provider-api.
  jar
Exception in thread "pool-2-thread-3"
-----
java.lang.LinkageError: org.apache.maven.wagon.providers.http.
  httpclient.cookie.ClientCookieException in thread "main"
java.lang.LinkageError: org.apache.maven.wagon.providers.http.
  httpclient.cookie.ClientCookie
#add to pom.xml
  <!-- added for http -->
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-site-plugin</artifactId>
    <version>3.0</version>
    <dependencies>
      <dependency>
        <groupId>org.apache.maven.wagon</groupId>
        <artifactId>wagon-http</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
  </plugin>
#after

```

```

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.0</version>
      <dependencies>
        <dependency><!-- add support for ssh/scp -->
          <groupId>org.apache.maven.wagon</groupId>
          <artifactId>wagon-ssh</artifactId>
          <version>1.0</version>
        </dependency>
      </dependencies>
    </plugin>

#for
constituent [31]: file:/usr/share/maven/lib/wagon-provider-api.
  jar
Exception in thread "pool-2-thread-3"
-----
java.lang.LinkageError: org.apache.maven.wagon.providers.http.
  httpclient.cookie.ClientCookieException in thread "main"
java.lang.LinkageError: org.apache.maven.wagon.providers.http.
  httpclient.cookie.ClientCookie
    at java.lang.ClassLoader.defineClassImpl(Native Method)
#add
  <dependency>
    <groupId>org.sonatype.aether</groupId>
    <artifactId>aether-connector-wagon</artifactId>
    <version>1.13.1</version>
  </dependency>
#under [see below] in pom.xml
  <dependencyManagement>
    <dependencies>

#for
    at org.apache.hadoop.hdfs.web.TestWebHdfsTimeouts.setUp
      (TestWebHdfsTimeouts.java:76)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native
      Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(
      NativeMethodAccessorImpl.java:94)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(
      DelegatingMethodAccessorImpl.java:55)
#Change the imports
./hadoop-common-project/hadoop-common/src/main/java/org/apache/
  hadoop/util/SignalLogger.java:import sun.misc.Signal;

```

```

./hadoop-common-project/hadoop-common/src/main/java/org/apache/
  hadoop/util/SignalLogger.java:import sun.misc.SignalHandler;
./hadoop-common-project/hadoop-common/src/main/java/org/apache/
  hadoop/security/SecurityUtil.java:import sun.net.dns.
  ResolverConfiguration;
./hadoop-common-project/hadoop-common/src/main/java/org/apache/
  hadoop/security/SecurityUtil.java:import sun.net.util.
  IPAddressUtil;
./hadoop-common-project/hadoop-common/src/main/java/org/apache/
  hadoop/io/FastByteComparisons.java:import sun.misc.Unsafe;
./hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/
  hadoop/hdfs/TestDFSClientFailover.java:import sun.net.spi.
  nameservice.NameService;

#for
hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-
  mapreduce-client-core/src/main/java/org/apache/hadoop/
  mapreduce/lib/jobcontrol/JobControl.java
#Change
import org.apache.hadoop.mapred.jobcontrol.Job;
#to
import org.apache.hadoop.mapreduce.Job;

```

C.3 Apache Spark on Ubuntu with Power

We used the following commands to build Apache Spark 1.1.0, available at <http://spark.apache.org/downloads.html>.

```

Source:
http://spark.apache.org/docs/latest/building-with-maven.html
#make sure you have the IBM Java VM
$ export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:
  ReservedCodeCacheSize=512m"
$ mvn -Pyarn -Phadoop-2.2 -Dhadoop.version=2.2.0 -DskipTests
  clean package
$ mvn -Dhadoop.version=2.2.0 test

```

We encountered the following errors:

```

#For error
- Verify special files exist with compression *** FAILED ***
  org.xerial.snappy.SnappyError: [FAILED_TO_LOAD_NATIVE_LIBRARY
    ] no native library is found for os.name=Linux and os.arch
    =ppc64le
  at org.xerial.snappy.SnappyLoader.findNativeLibrary(
    SnappyLoader.java:460)

```

```
at org.xerial.snappy.SnappyLoader.loadNativeLibrary(
    SnappyLoader.java:318)
at org.xerial.snappy.SnappyLoader.load(SnappyLoader.java:229)
#Change file to:
pom.xml
    <dependency>
        <groupId>org.xerial.snappy</groupId>
        <artifactId>snappy-java</artifactId>
        <!--<version>1.0.5</version>-->
        <version>1.1.1-M2</version>
    </dependency>

#for error
[ERROR] Failed to execute goal org.scalatest:scalatest-maven-
    plugin:1.0-RC2:test (test) on project spark-core_2.10: There
    are test failures -> [Help 1]
org.apache.maven.lifecycle.LifecycleExecutionException: Failed
    to execute goal org.scalatest:scalatest-maven-plugin:1.0-RC2
    :test (test) on project spark-core_2.10: There are test
    failures
#changed java version to 1.7 in pom.xml
#old:
    <java.version>1.6</java.version>
#new:
    <java.version>1.7</java.version>
```