Processing Big Data in Main Memory and on GPU

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Meisam Fathi Salmi, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Master's Examination Committee:

Xiadong Zhang, Advisor

Yang Wang

Abstract

Many large-scale systems were designed with the assumption that I/O is the bottleneck, but this assumption has been challenged in the past decade with new trends in hardware capabilities and workload demands. The computational power of CPU cores has not improved proportional to the performance of disks and network interfaces in the past decade, but the demand for computational power in various workloads has grown out of proportion.

GPUs outperform CPUs for various workloads such as query processing and machine learning workloads. When such workloads runs on a single computer, the data processing systems must use GPUs to stay competitive. But GPUs have never been studied for large-scale data analytics systems. To maximize GPUs performance, core assumptions about the behavior of large-sclale systems should be shaken and the whole systems should be redesigned.

In this report, we used Apache Spark as a case to study the performance benefits of using GPUs in a large-scale, distributed, in-memory, data analytics system. Our system, Spark-GPU, exploits the massively parallel processing power of the GPUs in a large-scale, in-memory system and accelerates crucial data analytics workloads. Spark-GPU minimizes memory management overhead, reduces the extraneous garbage collection, minimizes internal and external data transfers, converts data into a GPU-friendly format, and provides batch processing. Spark-GPU detects GPU-friendly tasks based on predefined patterns in computation and automatically schedules them on the available GPUs in the cluster.

We have evaluated Spark-GPU with a set of representative data analytics workloads to show its effectiveness. The results show that Spark-GPU significantly accelerates data mining and statistical analysis workloads, but provides limited performance speedup for traditional query processing workloads.

To *Aylan* and all innocent children like him who perished in wars.

Acknowledgments

I started my graduate studies in a new country, 6,000 miles away from my family and friends. I was not sure if it was the best decision at the time and I struggled through it day and night.

My advisor, Dr. Xiadong Zhang, was always patient with me and supported me in my mistakes. We tried out different ideas before working on this but he never chided my tardiness and lack of intelligence and I'm extremely thankful for that.

My collaborators, in High Perfomance Computing and Software Lab (HPCS lab) and in Programming Languages and Software Systems Research Group (PLaSS), provided invaluable feedback every time I asked. Yuan Yuan, Rubao Lee, Kai Zhang, Yin Huai, and Siyuan Ma gave me critical feedback and suggestions every time I asked and I am grateful for that.

Michael Bond was a great teacher for my personal development and growth. I learned priceless life lessons from him and I am thankful for that.

My friends never gave up on me and never left me alone in my darkest and saddest moments. I don not know where I would be today without them and I am thankful for that.

My family is indeed my most valuable asset. They are an eternal and unconditional source of motivation and support for me and I'm grateful for that.

Vita

2006 ..... B.S. Computer Engineering

2009 ..... M.S. Computer Engineering

2011-present ..... Graduate Teaching Associate,
Computer Science and Engineering.

Publications

Research Publications

M. Fathi Salmi and S. Parsa "Automatic Detection of Infinite Recursion in AspectJ Programs". *FGIT, 2009, vol. 5899, pp. 190197.*

M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang "OCTET: Capturing and Controlling Cross-thread Dependences Efficiently". *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, 2013, pp. 693–712.*

M. D. Bond, M. Kulkarni, M. Cao, M. F. Salmi, and J. Huang "Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine". *Proceedings of the Principles and Practices of Programming on The Java Platform, 2015, pp. 90101.*

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1: Introduction

Data is a vital organ in the anatomy of the global economy. Without data, decisions to enhance productivity, to keep customers satisfied, and to increase growth are difficult or impossible to make. Until the beginning of the 21st century, we were producing limited amount of data and the scale-up model was responding to the increase in computational demand. Each year more powerful computers with faster CPUs were being manufactured. If your current system was slow to process your data, all you needed to do was to buy a new computer with the latest CPU. After the chip-manufacturing industry hit its limits and the improvements in the CPU frequencies stalled, the scale-up model was not responding to the ever increasing demand for more computational power. Around the same time, the amount of data and how we use it changed seismically We started flooding an endless stream of data into virtually every aspect of our lives.

To respond to the explosion of data and stall in CPU clock improvements, a new model is needed. Today, we create 2.5 Quintillion Bytes of data each day. That is more than 200,000,000 email, 2,000,000 million Facebook shares, 200,000 tweets, 20,000 Skype conversation, and 4,000,000 Google search queries each minute [51, 48], which all will be obsolete measurements in one year from now.

Take the Google case for example. Google served less than 10,000 search queries each day in 1998. In 2014 this number grew to more than 5,740,000,000 a day [41]. The insight

that Google gets from analyzing this stream of data drives its 56.4 billion dollar advertising business.

With all the changes in the technology and market, the ability to process big data is not an advantage anymore, it is a necessity. To have an advantage over the competitors, companies need to process data quickly or even in real time. Detecting fraud should return results in a few minutes, otherwise it is too late, and detecting customer churn should be in real time. High frequency trading needs decisions to be made in milliseconds.

Analyzing the massive influx of data and turning it into value is a new technical challenge. The scale-out model emerged to respond to this challenge with frameworks such as MapReduce [12], Hadoop [46], Hive [43], Flume [8], Dremel [29] , Impala [14], and many more. Instead of using one powerful system, the scale-out model uses many commodity systems to process the data. This model made it possible to store and analyze data on orders of magnitude higher, which in turn created new businesses and companies such as Twitter, Facbook, and LinkedIn.

Apache Spark is one of the most popular scale-out framework for processing big data. It scales out to thousands of nodes and performs fast compared to other big-data processing frameworks. Spark clusters have up to 8000 nodes at Tencents, crunch up to 1 Peta byte of data in a single job at Alibaba.com, and process up to 1 Tera byte of data per hour at Janelia Farm. It also broke the world record for sort the algorithm beating Hadoop with fewer cores.

The main advantage of Apache Spark over previous frameworks is that it keeps and processes data in the main memory and avoids disk accesses. In Hadoop jobs, disk access time and network bandwidth are bottlenecks, but CPU is the bottleneck in Apache Spark. This is particularly true if worker nodes have arrays of SSDs and fast network cards, which

are common in clusters. With CPU being the bottleneck, any improvement in the execution time of a job can significantly benefit Apache Spark.

Apache Spark rarely optimizes the tasks and relies on the Java virtual machine for low level optimizations even though there is considerable room in Apache Spark for optimizations. It has a very strict computation model, which makes most of the data immutable and allows only certain operations to be performed on the data. These restrictions allow further optimizations on Sprak jobs that are not available to a Java virtual machine.

In this project we introduce two optimizations to Apache Spark:

1. the data layout of objects in the Java virtual machine.

2. the GPU as an available but unused recourse.

When the two optimizations are used together, they improve the performance of many Spark jobs. We describe each optimization shortly here and detail them in chapter 3.

All objects in a Java virtual machine need headers. Bookkeeping and management of object headers adds non-negligible overhead to the computation of a Spark job. This overhead is the extra memory needed for storing the headers and the extra time needed for garbage collection. Using a columnar format for object layout can eliminate most of this overhead, but this is not the only benefit of a columnar data layout. A columnar layout also 1) makes data serialization/deserialization easier and faster, and 2) suits the computation model of a GPU.

While Spark can use all the CPU cores on a node, it neglects available GPUs. GPUs particularly suit the Spark computation model–executing the same set of instructions on a massive amount of data. In this report we explain the changes we made to Apache Spark

to make it utilize GPU. We also measure performance of various workloads and investigate how they benefit from our changes.

1.1 Introduction to GPU Programming

In mid 1990s, the demand for applications with 3D graphics grew. To respond to this demand, companies such as NVIDIA and ATI released affordable graphic cards that could perform lighting and transform computations directly on the graphic processor. The high throughput of these graphic processors attracted researches to use graphic hardware for computation. But until mid 2000s, the only way to offload any computation to the graphic hardware was expressing the computation as a rendering task, which tricked the graphic hardware into thinking that it was rendering a 3D scene. Offloading intensive computations to the graphic hardware unleashed the potential in GPUs but programming them remained a difficult task for two reasons. First, programmers had to learn a shading language such as OpenGPL or DirectX to express their computation. Second, GPUs lacked the necessary tools for testing and debugging programs.

In 2006, NVIDIA released the first GPU with components for general purpose computing—the GeForce 3 series. The release of programmable GPUs attracted many researchers to use graphics hardware for more than rendering. By 2010, GPUs were used in many applications, including Database Management Systems and processing SQL queries [5, 8, 7].

1.1.1 GPU Architecture and Execution Model

Each modern GPU has three component: a global scheduler, a set of streaming multiprocessors (SMs), and the global memory. Streaming multiprocessors are the basic building blocks of computation units. Each streaming multiprocessor typically has 32 SIMD cores and can mange its cores and schedule thousands of resident threads independent from other

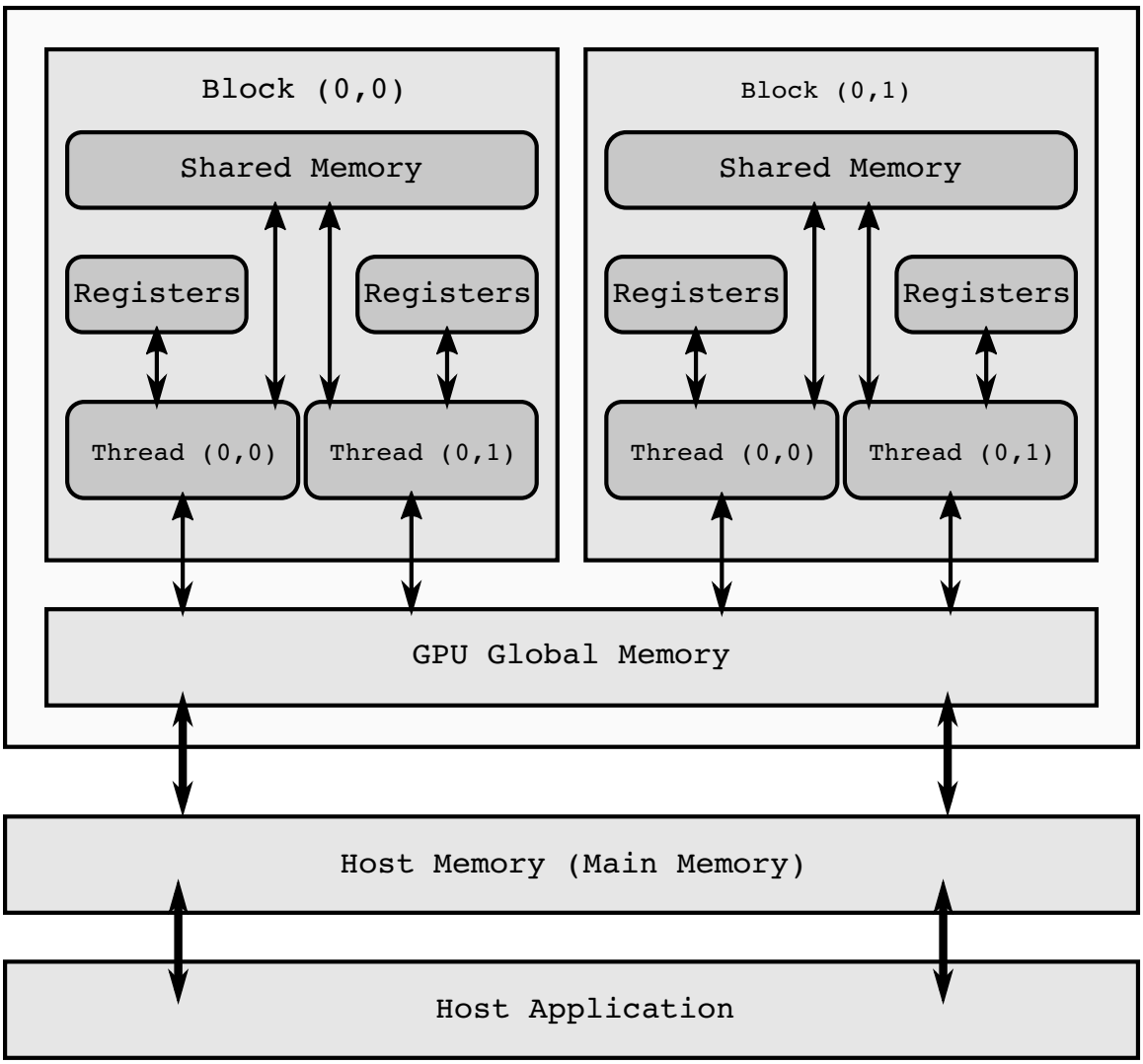

Figure 1.1: The architecture of a GPU and the execution model of a program on it. The host application dispatches the kernel on the GPU device. Working threads on the GPU device are grouped into blocks and access the data from the device's global memory. Working threads can use the shared memory or their registers, which are faster compared to the global memory, as a cache. Once the kernel is executed, the host application can copy the results to its main memory.

streaming multiprocessors. The global memory serves as the shared memory among all threads.

To run a program on a GPU, a host program on a CPU should initiate it. The GPU program is called a kernel. The host program allocates required buffers on the GPU and launches the kernel. After the execution, the host program frees the allocated buffers and terminates the GPU session. Figure 1.1 shows this process.

1.1.2 Thread Hierarchy in GPUs

GPU threads are uniquely identified with a one-dimensional, two-dimensional, or three-dimensional thread index. There is a limit to maximum the number of threads that can be identified with a thread index, which is the maximum number of resident threads that a streaming multiprocessor can mange (maximum number of resident threads depends on the hardware specification of each GPU). For kernels that need more threads, the host program should group and break down the threads into blocks. Each block can have up to maximum number of threads and is uniquely identified with a one-dimensional, two-dimensional, or three-dimensional thread index.

1.1.3 Memory Hierarchy in GPUs

GPUs lack the cache hierarchy of CPUs but they benefit from high memory bandwidth. When the characteristics of the GPU memory are exploited, it can yield to high throughput. Figure 1.1 shows the memory hierarchy of a typical GPU.

The DDR memory in a GPU serves as the global memory for all threads. Each SM has a small amount of shared memory per block that is accessible to all threads in the block. Finally each thread has a limited number of registers.

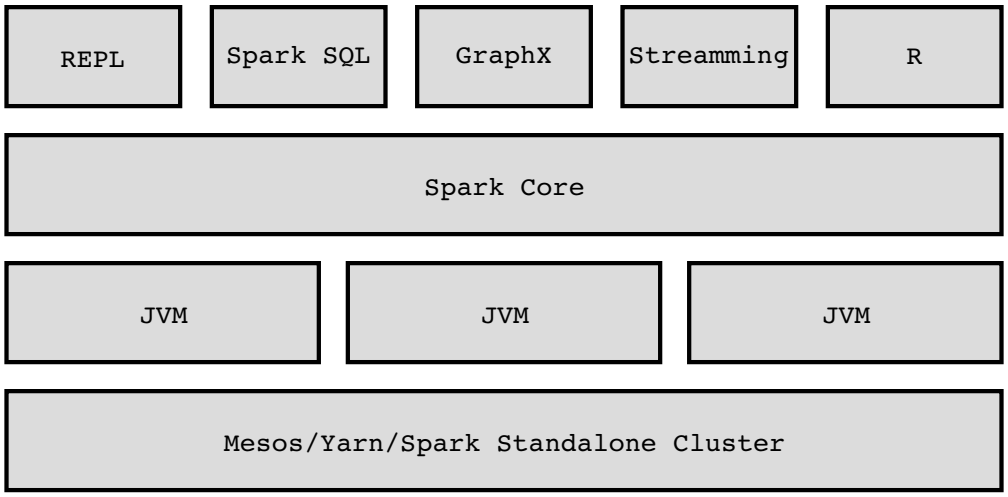

Figure 1.2: Spark Architecture. Spark core runs on Java Virtual Machines and schedules tasks on the underlying cluster. Other componenrts of Spark such as Spark SQL, use Spark Core to run jobs.

1.2 Introduction to Spark

Apache Spark [50] started as a project in University of California, Berkeley. It is a distributed computing framework that runs on a cluster and is designed to be scalable, fault tolerant, easy to use, and fast. Apache Spark uses a strict programming model to achieve scalability and fault tolerance. Any computation in Apache Spark is strictly defined by a set of operators on immutable datasets. If the computation fails an any point, Apache Spark recomputes the intermediate results up to a previous checkpoint. Apache Spark is also easy to use because it offers high level APIs in a verity of popular programming languages such as Python, Java, SQL, Scala, and R.

1.2.1 Apache Spark Architecture

Figure 1.2 shows the architecture of Apache Spark. The underlying cluster consists of a master node and a set of worker nodes. The master node keeps track of the worker

nodes and their available resources. The node that schedules the job is called the driver node and is not necessarily the same node as the master node. Different Apache Spark configurations allow the master node to do the scheduling of jobs even though that is not the default configuration. The Spark Core component, among other things, schedules tasks on the underlying cluster, manages resources, retries failed tasks, and communicates with the underlying cluster. Other components in Apache Spark, such as SQL, GraphX, and MLLib, use Spark Core to schedule and run jobs.

1.2.2 Apache Spark Programming Model

Each Spark application is a sequence of operation that transforms an input dataset to an output dataset. These datasets, along with the operations that transforms them, form a lineage graph where each dataset has one or more parent dataset and may have one or more children.

Spark datasets are immutable, and are strongly and statically typed. Values in a dataset are computed lazily and can be cached in the main memory of the worker nodes as long as they fit into the collective main memory of the cluster. In Sparks terminology, these datasets are called Resilient Distributed Datasets (RDD) Child datasets recursively compute their parents before computing their own. If the parents' data is available in the cache or at a checkpoint, the child process uses it, otherwise, Spark schedules tasks to compute the missing input data. In Spark, the *DAG-scheduler* breaks down the lineage graph into task-sets and schedules them on the underlying cluster. At the cluster level, a master node receives the task-sets and assigns them to available workers. As workers completed their assigned tasks, they notify the master node and they may receive new tasks to run from the master node.

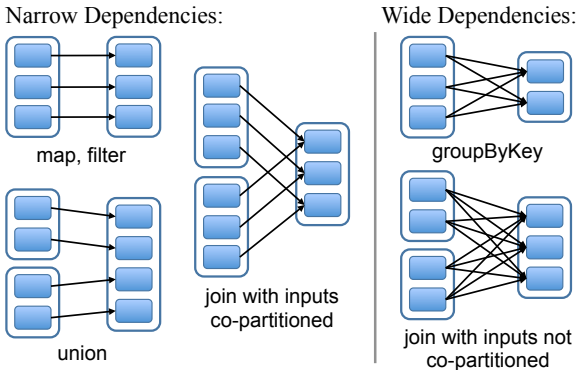

Figure 1.3: Narrow and wide dependencies in Spark. A wide dependency is where data needs to be shuffled on the network [50].

Spark uses a very simple mechanism for fault tolerance: it lets the tasks fail and restarts them on failure. If a subtask on a worker node fails, Spark scheduler knows how to recompute the data because all dependencies between parent and child datasets are known to the Spark scheduler. Spark provides different levels of storage and replication. Each RDD in Spark can be stored at one of the three available storage levels: Memory-Only, Disk-Only, and Memory-And-Disk. Moreover, each RDD can have a different level of replication (1, 2, 3, ...) to make it more resilient to failures and make recoveries from failures faster.

Each job in Apache Spark is a set of operations that creates a new RDD from one or more input RDDs. Spark scheduler breaks each job into one or more stages based on the operations on the RDDs. Spark scheduler furthermore breaks down each stage into tasks and dispatches tasks to the worker nodes in the cluster. Because Apache Spark uses a very constrained programming model, each worker nodes knows exactly how to get the input for its assigned task, how to generate the output for it, and where to store the output. If a task only needs to get its input from the same worker node, then Spark scheduler does not need to shuffle data on the network. In this case, Spark scheuler forms a linear dependency

```
1 val textFile = spark.textFile("hdfs://...")
2 val counts = textFile.flatMap(line => line.split(" "))
3     .map(word => (word, 1))
4     .reduceByKey{ case (count1, count2) => count1 + count2 }
5 counts.saveAsTextFile("hdfs://...")
```

Figure 1.4: A sample word count program in Apache Spark

between the task and its parent; and pipelines the operations of the two task. But if the input of a task depends on the output of a tasks on different worker nodes, Spark scheduler forms a wide dependency (or shuffle dependency) between the task and its parents, ends the boundary for this stage, and creates a new stage in the job execution. When tasks in a stage with wide dependency are scheduled, their input should be shuffled on the network. Spark scheduler cannot pipeline shuffled dependencies and has to wait for the output of the previous stage before it starts the current stage. Figure 1.3 depicts the two types of dependencies in Apache Spark.

As an example consider the simple Spark program in figure 1.4, which is taken from the sample programs shipped with Apache Spark. Line 1 reads an input text file and splits each line into an array of words. Line 2 maps each word to a tuple `(word, 1)`. In this tuple the word has an assigned value 1. Line 3 adds up all the assigned numbers to each word. Finally line 4 writes the results back to the file system. In this snippet, Spark operations are `flatMap`, `map`, and `reduceByKey`. The `flatMap` and `map` operations form a narrow dependency but the `reduceByKey` operation forms a wide dependency.

1.3 Organization of this Thesis

In this chapter, we discussed the benefits that GPUs can bring to computationally intensive workloads. We also discussed the benefits that Apache Spark, as a framework

for processing big data, brings. In Chapter 2, we discuss the characteristics of the current workloads for big data and will show that CPU is the bottleneck for many workloads. We also propose that GPUs can be used in big-data processing frameworks to speed up CPU bound computations. In chapter 3 we show how to integrate GPUs into big-data processing frameworks. We also explore the challenges that this solution faces. Chapter 4 presents the experiment results and gives pointers to future research that can be based on this work. Chapter 5 surveys the related work for both large-scale analytics systems and GPU-enabled high performance systems. Chapter 6 summarizes the contributions of this thesis and concludes the work.

Chapter 2: Background and Motivation

Apache Spark is a significant improvement over the MapReduce programming model [12]. It simplifies programming and speeds up the computation but it has its own drawbacks. In this chapter we identify two of the limitations of Apache Spark, particularly when Spark is used for CPU-bound computations. The first limitation is that the current data model in Apache Spark is wasteful and does not suit the computation model of a GPU. The second limitation is that Apache Spark is oblivious to the GPU resources in the cluster. Apache Spark ignores GPU resources despite the fact that its massive parallel computation model suits GPUs.

2.1 Object Layout and Data Format

Spark's iterator model processes one element at a time. Elements that Spark processes are Java objects with a header and fields that are pointers to other objects. Neither the one-elemet-at-a-time execution model nor the Java object layout are efficient for GPU processing because they underutilize GPU resources. Batch processing continuous data without headers or pointers is the suitable model for GPUs.

To use GPUs in Apache Spark, the data format and the execution model should be amended. Even though it is unrealistic to completely rewrite the current Spark's execution

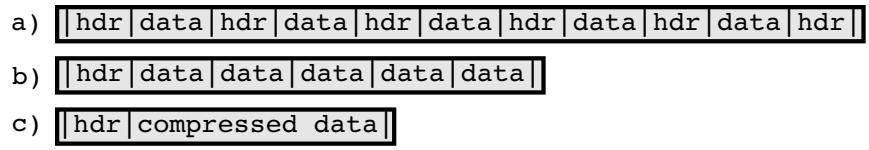

Figure 2.1: Java Object model is unsuitable for GPU. a) Object headers impose significant overhead, make data access patterns cache unfriendly, and are inefficient for standard compression algorithms b) Most of the objects in an RDD have the same type and lifespan. There is no need to repeat the header for each object. c) If object headers are stripped, data compression algorithms are more effective on the header-stripped data.

model, Spark RDDs have properties that makes use of a GPU feasible. We discuss these properties in this section here.

RDDs in Spark are strongly and statically typed, which means all the objects in an RDD have the same type. Moreover, objects in an RDD have the same lifespan, that is they are created, used, and garbage collected all together at the same time. The Java object model is general purpose and is designed to work for any object in any application. Apache Spark can use a simplified object model that incurs less overhead. We outline the design of this simplified object model in this section.

In the Java object model, an object has a header part and a data part. The data part stores data associated with the object, but the header is transparent to programmers. Implementation of the object headers varies from JVM to JVM, but at least the following information is stored in the object header in all major JVMs.

Object type information: Most JVM implementations have a pointer in the object header that points to the class descriptor that defines the object [26, 3].

Object reachability: In a managed language such as Java, the garbage collector needs to know which objects are still in use and which are available to be freed and collected.

All major JVM implementations store reachability status of on object in its header [39].

Synchronization and Locking status: Modern JVMs use more sophisticated algorithms, such as biased locking, for locking and synchronization. JVMs store the extra information about the owner thread and the status of the lock in the object header [25, 36].

Identity hash code: Most modern JVMs use the physical address of the object as its identity hash code. But Java garbage collection constantly moves objects around in the memory and, hence, changes the object's physical address. JVMs store the object's identity hash code in its header when the object is moved for the first time [6].

This general purpose object model serves well in many applications, but in Spark it adds significant overhead without being used. For example, Objects in RDDs are rarely locked or synchronized, which means that the synchronization information in the object header is extraneous and useless. As another example, one object header per partition usually suffices in Spark because all the objects in an RDD have the same type.

Not only is Java object model inefficient for processing data on GPUs it also makes data compression and serialization inefficient. Figure 2.1 shows the problems with the Java object model and how a continuous data format is more efficient for serialization, compression, and GPU processing.

2.2 GPU's Execution Characteristics

GPU kernels should be launched with a large number of GPU threads to hide GPU memory access latency. They also should access coalesced data to fully utilize GPU memory's high bandwidth. Data accesses are coalesced if consecutive threads read consecutive

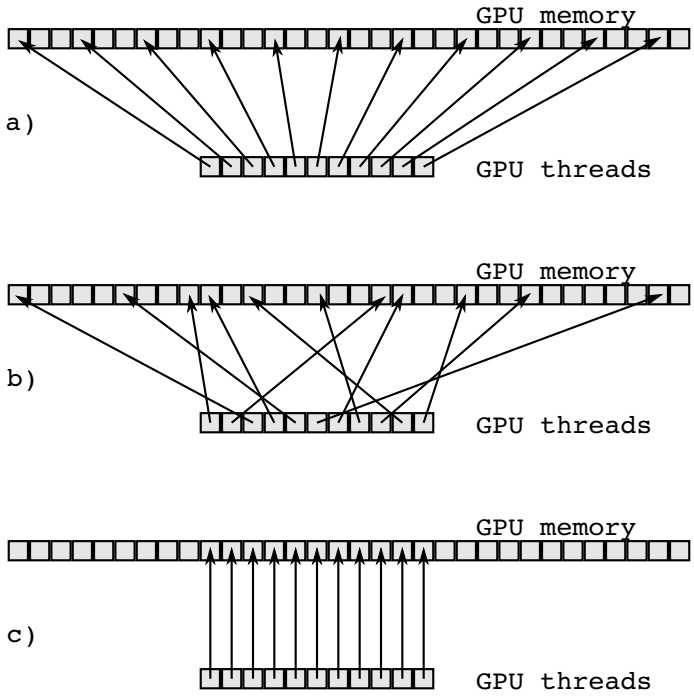

Figure 2.2: a) Stride access is not coalesced. b) Random access is not coalesced. c) Coalesced access.

addresses. This data access pattern is inefficient for CPU threads because it leads to many cache misses, but it is the best access pattern for GPU threads. Figure 2.2 contrasts coalesced vs. consecutive data access patterns. If data is coalesced, GPU can read the whole chunk in one transaction and feed the data to the GPU threads. For non-coalesced data, GPU has to access its memory more than once. The performance difference between accessing coalesced data and non-coalesced data is an order of magnitude.

Data accesses are coalesced in columnar format. To integrate GPU into Apache Spark, the data should already be in columnar format or should be converted to columnar format. Apache Spark already supports a variety of columnar formats such as Parquet [29], and RCFile [20] format.

Table 2.1: Hardware trends in disk, network, and CPUs from 2005 to 2015.

	2005	2015	Speedup
CPU Core Frequency (GHz)	3	3	–
Disk Speed (MB/s)	50	500	$\sim$ 10X
Network Speed (Gbps)	1	10	$\sim$ 10X

2.3 Potential for Improvement

Ousterhout et al. [33] observed that most of the work to improve performance of large-scale data analytics systems was focused on mitigating the network bottleneck, the disk bottleneck, or the straggler tasks. But in reality network optimizations can improve the job performance by at most 2%, disk access optimization can improve the performance by at most 19%, and optimizing stragglers can improve the the performance by at most 10%.

In the past decade the speed of network connections grew from 100 Mbps to 1 Gbps or even 10 Gbps. The disk speed grew from 50 MBps in HDDs to 500 MBps in the cutting edge SSDs. But the speed of CPU cores stagnated around 3GHz. Table 2.1 shows the trends. The emerging non-volatile byte-addressable memory technologies [17], and optic interconnect networks will amplify the trend.

In todays workloads, CPU cores are the new bottleneck. This is particularly true for in-memory data analytics systems such as Apache Spark. For in-memory data analytics systems, simple operations such as data compression or serialization can easily saturate the CPU. Any system that runs on a cluster has overhead at many layers. The cluster management, task scheduling, replication, failure recovery, and marshaling data burden

the CPUs with extraneous computation. We are interested in identifying this extraneous computations and eliminating them.

To estimate the amount of extraneous work that in Apache Spark, we compared its performance to two other efficient system: an single-node DBMS, and an efficient distributed DBMS. We selected YDB [49] as the single-node system. and Impala [14] as the distributed system.

YDB does not add the overheads associated with a distributed system such as communication costs. The performance of YDB give us an estimate on how much we can improve performance of Spark even though this estimate is optimistic and difficult to achieve in reality. We also compared the performance of Apache Spark to Impala. Since Impala only runs SQL queries it can avoids extraneous work that would be necessary for a more general purpose system. Hence Impala's overhead is minimal compared to Apache Spark. We estimated the overhead of Apache Spark by comparing its performance to Impala's performance. The difference between Impala's performance and YDB's performance is an estimate of the overheads caused by the distributed nature of Impala.

We compared Impala and Spark SQL using query 1.1 from the SSB benchmark on three different file formats: flat files, uncompressed Parquet files, and compressed Parquet files. Parquet [2] is a columnar file format and is wildly used in the Hadoop ecosystem. YDB uses its own columnar file format, which not supported by Impala or Spark SQL. We did the experiments under two configurations. In the first configuration, we kept the operating system caches across iterations, but for the second configuration we did not.

Figure 2.3 Shows the results. Examining This figure leads us to three conclusions.

- As expected, YDB outperform both Impala and Apache Spark. YDB's better performance shows the inherent overhead of running a distributed system.

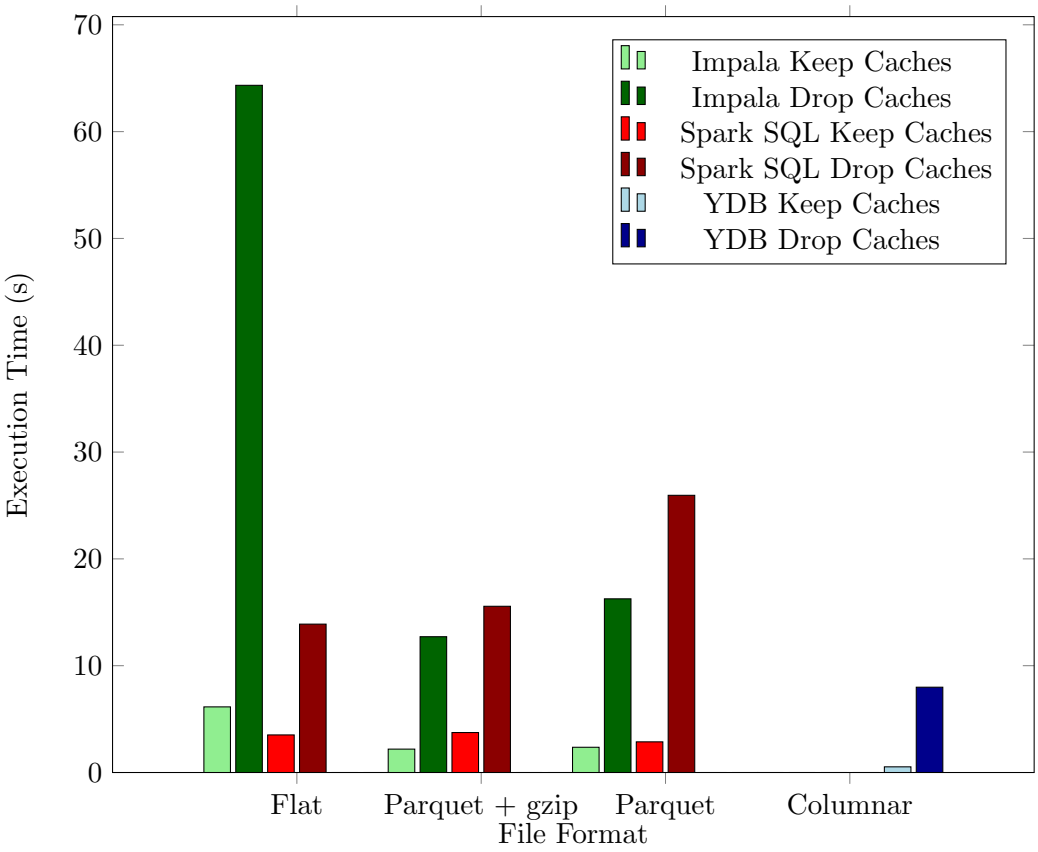

Figure 2.3: Spark SQL performance vs. Impala performance vs. YDB

- Impala performs poorly on flat files, particularly when the system is cold, but the flat files are rarely use with Impala in practice.

- Impala outperforms Spark SQL on Columnar File Formats, Since both Impala and Apache Spark execute the same job, access the same amount of data, and produce the same amount of results, the extra execution time in Apache Spark has to be the internal overhead of Spark, In this report, we are interested in identifying and eliminating the extraneous overhead of Apache Spark.

In this chapter we examined Apache Spark as a popular large-scale data analytics system and identified two limitations. We argued that supporting columnar storage in Apache Spark not only decreases the overhead of the program, but also helps integrating GPUs to Spark. We showed that the overhead of Apache Spark compared to other large-scale systems, particularly Impala, is high and there is room for speeding up the execution. We showed that during the past decade, CPUs have become the bottleneck for many large-scale applications. With CPU cores as the new bottleneck, large-scale systems should minimize all the overhead as much as possible and should exploit computational resources in their cluster to mitigate the bottleneck. Next chapter discusses how to mitigate the current limitations of the large-scale systems.

Chapter 3: Implementation

In chapter 2 we discussed two limitations of using GPUs in Apache Spark. In this chapter, we discuss how we extended and modified Apache Spark to solve its limitations. We added 3 components to Apache Spark to integrate GPUs into it. We call our modified version *Spark-GPU*.

- We added an API to Apache Spark for batch processing data on GPUs. We call this API the *batch processing* API.

- We extended Apache Spark RDDs and added five sample operations to the RDDs interface that run on GPUs. Our added operations are filter, join, map, aggregate, and sort, which are five common operations in Spark. They use reflection to automatically infer type of objects in RDDs and convert the data layout to the columnar format, which is the most suitable format for GPUs. We call this API, the *reflection based* API.

- We extended the SparkSQL component and added operations that run on the GPU. For example this component adds a join operation that runs on GPUs. Based on the cost and order of operations, our extension decides which operations to offload to the GPUs and which to keep on the CPU. We call this component *SparkSQL-GPU*.

Our implementation was 5,000 lines of Scala code, 1600 line of C code, and 7000 lines of CUDA code. In the following sections we discuss how we implemented each component.

3.1 Reflection Based API

The type of an object in a statically typed language like Java or Scala is partially known at compile time, but the Java virtual machine keeps checks and assertions ensure certain properties about the type at runtime. One common optimization in Java and Scala is to erase type checks when they are unnecessary. Scala by default does many type erasure optimizations, but it also provides mechanisms to keep type information when asked by the programmer. One such mechanism in Scala is `TypeTag`s. We used Scala `TypeTag`s to keep compile time information about the type of RDD and use them at Runtime. If objects in an RDD have complex types (e.g. tuples), we use the type tag information to infer and break down the type into multiple primitive types. The data in the RDD then is converted into columnar format and based on the inferred primitive types.

The Reflection Based API provides five basic operations that work on the columnar data: filter, map, join, aggregate, and sort. It also provides utility functions that take an iterator from Spark, partitions it into chunks, and convert each chunk into columnar format. Each chunk is small enough to fit into GPU memory but large enough to get the benefits of massive parallel processing from GPU.

Each operation works at the partition level and can be combined with other operations to express more complex computations. Because each operation works at the partition level, extra work is needed to make operations that work across all partitions. For filter and map operations, this is not an issue because dependencies between a filter operation to its parent and a map operation to its parent in Spark are narrow and do not need the data to

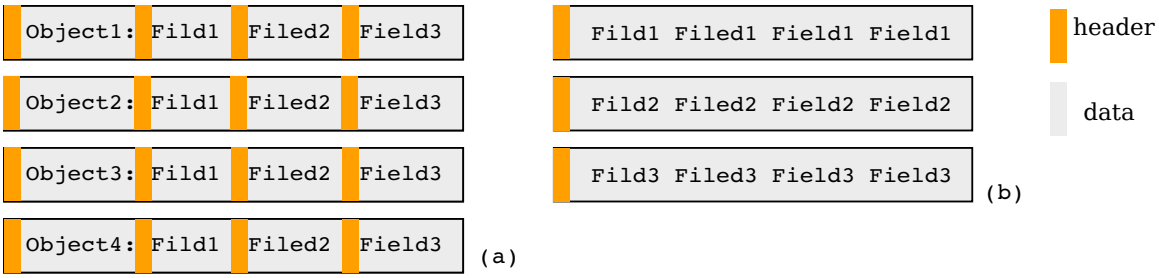

Figure 3.1: (a): Java virtual machine layout vs. (b) Columnar layout.

be shuffled. Aggregate and sort operations also work within a partition, but do not cause a problem in practice because the result of the aggregate and the sort operations are small enough to fit into one partition.

The join operation is useful when one side of the join is small to enough fit into one partition and can be used in a broadcast joins. Joining RDDs that do not fit into memory requires the programmer to implement a nested loop join.

3.1.1 Data Format

We changed the data layout of Spark to make it columnar. As shown in Figure 3.1, we decompose each object into its fields and store the same field of all objects in one buffer. To reconstruct the object, we take each field from its corresponding buffer and put them back together.

Columnar storage puts a field of all objects on consecutive memory offsets, which makes it easy for bulk transfer to GPU. Without columnar storage, each field of each object should be transferred to GPU separately.

If the buffer becomes larger than a specific predefined limit, we partition it into smaller buffers. The size of GPU memory dictates this predefined limit.

Our strategy is to keep data in the GPU memory only when needed because other parts of the program may need the GPU memory for their computation. When data is needed for a computation on GPU, we transfer the data to GPU, do the computation, and evacuate the data from the GPU memory. This policy is conservative, but it makes "out of memory" exceptions less likely. It also lets us select a bigger maximum size for partitions. The drawback is that if some data is needed for consecrative computations and enough GPU memory is available, we transfer data between GPU and CPU multiple times.

Once data is in columnar format and is partitioned, we use the original Spark computing model, i.e. RDDs (Resilient Distributed Datasets). In the design of Reflection Based API, instead of shuffling data, we broadcast data (the amount of data we need to send is relatively small in practice form many workloads). Avoiding data shuffles makes the design of Spark+GPU simpler because columnar data cannot be shuffled without object reconstruction.

We extended the existing RDDs to exploit GPU computation power. These RDDs are particularly designed to execute common operations in SQL queries. We added customized RDDs for selection, projection, join, and aggregation on top of the basic operations. All computations are represented by a directed-acyclic-graph (DAG) of our customized RDDs. To run a given SQL query, we manually parse the query and make an execution plan. Nodes in the execution plan directly translates into Spark-GPU RDDs and dependencies between nodes in the execution plan translate into dependencies between parent-child RDDs.

Our implementation of reflection based API is based on Spark 1.3. We particularly used Spark 1.3 because we needed two new features available in Apache Spark 1.3:

1. We needed to serialize/deserialize the types of fields and tuples, which was not available in previous version because of a bug in Scala-2.10.

2. We wanted to measure how much the new *code generation* feature in Spark SQL 1.3 improves the performance of our benchmarks.

We also used OpenCL[30] to implement GPU operations because it is available on a broader set of GPUs. The interface between Scala/Java and OpenCL uses Java bindings for OpenCL [21].

Currently we only support primitive type fields (int, float, char, etc.) that are put together in a tuple. Tuples in Scala are represented with the `Product` class, which takes a type parameter `T`. We require this `T` to be a subtype of `TypeTag` because it allows us to use Scala/Java reflection to decompose it into its primitive types with the API in the `scala.reflect.runtime` package. Scala type erasure by default removes any type information that is not needed at runtime, but a type tag forces the Scala compiler to generate code to keep the type information.

The base class for our implementation is `GpuPartition`. Each customized RDD in the Reflection Based API holds `GpuPartition[T]`s, where `T` is the underlying type of items in the RDD. For example an RDD of type `GpuPartition[[Int, String, Float]]` contains tuples with three fields, the first field is `Int`, the second one is `String`, and the third one is `Float`.

Buffer Management

Each `GpuPartition` has an array of buffers for each primitive type, which it initializes lazily and uses to store fields (each buffer is used for one field only). In our previous example, `GpuPartition[[Int, String, Float]]`, one `Int` buffer, one `String` buffer, and one `Float` buffer are used. String buffers are implemented as `Byte` buffers because Strings are not primitive types in Scala/Java.

The maximum size of each buffer is determined based on a predefined parameter. If the total size of the data is more than this limit, we partition the data such that each partition fits into the GPU memory. In our implementation, we could use buffers with 2^{28} elements before overflowing the integer offset of items in the buffers.

In our initial implementation, we allocated these buffers from Java heap memory, but because these buffers are huge, they burden the virtual machine's memory management and garbage collection. We changed the implementation and used direct buffers to avoid exhausting the Java heap using `ByteBuffer.allocateDirect()`. Direct buffers additionally allow us to copy data from Java virtual machine to GPU without extra steps. Without direct byte buffers, data should be copied to GPU in two steps: 1) from Java heap to VM/OS buffers 2) and then from VM/OS buffers to the GPU. The downside, however, is that garbage collection does not manage and reclaim direct buffers and we have to explicitly release them after use.

Byte Ordering

Scala/Java has a big-endian byte ordering system, but C/OpenCL is platform dependent. Our implementation targets x86_64 platforms, which use a little-endian byte ordering. To avoid converting data back and forth between little-endian and big-endian systems, we keep data in Java direct buffers in little-endian as well. Data sitting in direct buffers rarely needs to be accessed in Scala/Java because the majority of the computation on buffers is done in C/OpenCL code. Scala/Java may access a buffer's data only at the end of the computation to get the final results. Since the final results are usually limited in practice, the cost of converting data into big-endian byte order is not negligible.

3.1.2 Reading Columnar Files

`GpuPartition` and its subclasses can read data from in memory Java/Scala data structures or from columnar files using to custom RDDs that we added. `InMemoryGpuRDD` reads Java/Scala data structures and converts them to columnar format upon reading. `ColumnarFileGpuRDD` reads from columnar files. Our columnar file format saves each buffer in a separate file. The file has a header that describes the type of the data in the file, the total number of partitions in the file, and the total number of elements in each partition. After the header, all the data is stored in binary format with little-endian byte ordering. We do not compress columnar files. This file format is a simplified version of the file format supported by YDB [49].

3.1.3 GPU Physical Operations

We added the following special RDDs to our Reflection Based API.

1. A ScanRDD that reads columnar data from a (distributed) files system.

2. A FilteredRDD that has one parent and one criterion. It checks the given criterion on its parent's data and selects the part that passes the criterion. For example a criterion can be *values more than 50*. Users can chain simpler criteria to make more complex criteria.

3. A JoinRDD that has a left parent and a right parent. This RDD joins its parents data based on a specified join column.

4. An AggregationRDD that has one parent and a set of functions. Each function either groups or summarizes data in one or more columns.

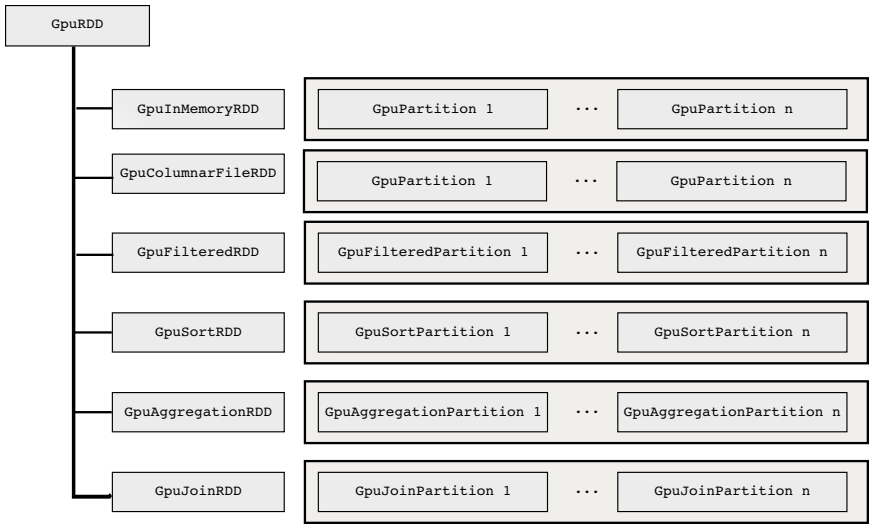

Figure 3.2: Design of RDDs and their corresponding partitions in Reflection Based API

5. A ProjectionRDD that has one parent and returns only projected data items from its parent. This RDD is useful for dropping columns that are not needed after a certain point in the query execution. For example, if a column is used only at the first stage of a query plan in a FilteredRDD, that column can be dropped from the query after the Filter operation finishes.

6. A SortRDD that has one parent and sorts its parent's data.

All the customized RDDs for Reflection Based API and their corresponding GPU-partitions are shown in Figure 3.2.

Scan Implementation

The scan operation runs only on CPU and does not use GPU and we implemented two versions of it: 1) to scan data from Java/Scala data structures 2) to scan data from a columnar file.

The first scan operation is particularly useful for converting original Spark RDDs into columnar RDDs.

The columnar file scan is useful when data is already in columnar format on disk. It eliminates the cost of converting data to columnar format, which can be significant for large datasets.

Filter Implementation

The filter operation is implemented in `GpuFilteredPartition` and `GpuFilteredRDD` The filter operator starts by sending the filter column to the GPU. A work group of GPU cores are assigned to scan the column. The work group size is 264 at the moment.

The filter operation on GPU has three stages. First stage marks all the elements that match the criterion.

In the second stage, each GPU core counts the number of elements before it computes the offset in the output buffer where it should start putting its results. In the third stage, other columns for selected rows are copied to the results.

Join Implementation

The join operation is implemented in `GpuJoinPartition` and `GpuJoinRDD` Join implementation is based on the assumption that one side of the join is small enough to fit into the GPU memory (We always assume that the right parent of the join is small enough to fit into the GPU memory and to be broadcast). This is true for SSB queries because they have a star schema. This assumption allows us to broadcast the small side of the join instead of shuffling it.

The join implementation is a hash join and is done in five stages. After sending the data to GPU, the join operation hashes the join column of the right parent and counts the number of unique hash values. Then it creates a hash table for the join columns.

In the second stage, the left parent probes the hash table and sets a bitset if there is a match for the join column. In the third stage, each GPU core counts the number of join results to find the output buffer offset it should use. In the forth stage, all columns needed for the join result are transferred to GPU and based on the constructed bitset map, all the columns from right parent are materialized, In the last stage, all columns from the left parent are materialized as well. Finally all the results are transferred from the GPU memory to the CPU memory.

Aggregation Implementation

The aggregation operation is implemented in `GpuAggregationPartition` and `GpuAggregationRDD` Aggregation implementation summarizes the data in each partition separately from other partitions, i.e. if data does not fit into one partition, one summary per partition is generated. In SSB queries, we found that by the time an aggregation is executed, the data is small enough to fit into one partition.

Each aggregation partition takes a parent partition, a set of columns to group by and a set of functions to aggregate data. Currently we support MAX, MIN, AVERAGE, and SUM functions on `Byte, Char, Short, Boolean, Int, Float` data. Data types that take 64-bit or more (like `Double, Long`) are not supported yet because most GPUs lack atomic operation support for 64-bit data.

Aggregation starts by sending the group-by columns to the GPU memory. Then a hash table of group-by columns is created. Aggregation uses this hash table to find the total number of groups needed and the number of data items in each group.

Then each GPU core counts the number of results. Extra columns that are needed for aggregation are sent to the GPU. The GPU summarizes the data in each column based on the specified aggregation operation.

Sort Implementation

The sort operation is implemented in `GpuSortPartition` and `GpuSortRDD` and can sort data based on up to two keys. Sort implementation sorts each partition independent of other partitions, but in the SSB queries, we found that by the time a sort operation is required in the query execution plan, dataset is small enough to fit into one partition.

Sort operation starts with sending the first sort key column to the GPU memory. Then it counts the number of unique sort keys and number of elements for each unique sort key. Then it gathers all the second sort columns and buckets them with one bucket for each first key. Then each bucket is sorted based on the value of the second key. Finally all the columns are gathered, materialized, and sent back to the CPU.

3.1.4 Limitations Reflection Based API

We added implemented Reflection Based API to demonstrate that we can automatically convert RDDs into columnar format. But our implementation has a few limitations, which are mostly because of the limitations in the underlying programming languages—Java and Scala—or the underlying hardware–NVIDIA and ATI graphic processors. We discuss the limitations here.

Full Type Inference

The reflection based API can correctly infer the number and the type of the columns based on the compile time type of the RDDs. For `FilterDDD` and `SortRDD`, it can infer the type based on the type of the parent RDD and the programmer does not need to specify the

type. For example if we have a columnar RDD with the type `[Int, String, Float]`, and we run a filter operation or a sort operation on it, the type of the result RDD will remain `[Int, String, Float]`. But for `ProjectRDD`, `JoinRDD`, and `AggregateRDD`, the type of result RDD should be specified by the programmer.

Support for Complex Aggregation Operations

Right now we support aggregation operations that are composed of at most two sub-expressions. We decided to support only two sub-expressions because our benchmarks had mostly simple aggregations. Supporting more than two sub-expression bloats the implementation but it does not add a valuable feature.

Aggregation on Double and Long Types

Many mainstream GPU processors do not support atomic operations on 64-bit words. Implement atomic operations on data with any arbitrary length is possible via locking mechanisms but it bloats the code. Newer versions of the GPU processors are equipped with 64bit atomic operations.

3.2 Batch Processing API

To support batch processing on the iterator model, we introduced a new type of RDD, *GPU-RDD*, along with a data structure, *GpuRecord*, into the system. GPU-RDD serves as a batching operator during the execution. It batches the data and converts them into columnar format. The *GpuRecord* data structure represents the batched data.

A *GpuRecord* batches one partition of date and contains both the columnar data and the metadata information. The metadata information includes number of elements, column types, and column sizes. Data in a GpuRecord can be stored in a continuous memory region for all columns all together or separately. Storing each column separately is beneficial for

workloads that use a subset of data. For example, most SQL queries use a subset of the columns. Storing all columns together is beneficial when number of columns are very large. For example many machine learning workloads work on data with thousands or even millions of features.

GpuRecord stores the data in the native memory, not in the Java/JVM heap memory. Data in the native memory can be directly copied to the GPU memory without using intermediate OS level buffers. It also does not burden the garbage collector and the memory management system of the JVM. Storing all the batched data in the Java Heap increases memory usage, which leads to frequent garbage collections. This can significantly degrade a task's performance.

GpuRecord releases the native memory when its corresponding RDD is garbage collected by JVM. Unfortunately Java language specification does not give any guarantee on when garbage collection is invoked. This means that a GpuRecord may stay alive and occupy memory longer even if the GpuRecord is not used anymore. The second problem with this schema is that the demand for Java heap is not proportional to the demand for the native memory and we may run out of the native memory before we run out of Java heap. We also added an interface to explicitly release the memory but in our experience with the benchmarks, this was not a real issue, particularly when the size of each partition was selected carefully .

A GPU-RDD can be created from a Spark RDD or other GPU-RDDs. We extended the interface for Spark RDDs to add a `toGpuRDD()` method, which converts the RDD's data into GpuRecords.

3.2.1 Batch Processing Interface

To offload computation to GPUs, users need to create a GPU-RDD from an existing RDD. Since we targeted this implementation for running SQL queries, we require the schema of the data when an GPU-RDD is created. After a GPU-RDD is created, any function can be applied to it as long as it accepts a GpuRecord.

GPU functions can be implemented in any language, including C/CUDA, but they need a wrapper function written in Java/JNI. In our implementation, wrapper functions call the CUDA kernels inside the JNI wrapper code. The wrapper function is responsible for passing the information about the GpuRecord to the GPU function. This information mostly is the address of buffers, their sizes, and number of elements in each buffer.

Users need to compile their GPU-Functions before using them, but we modified Apache Spark to ship the compiled libraries to all worker nodes. If worker nodes in the cluster have different architectures, the GPU functions should be compiled into a static library. We also modified Spark to automatically load the library. This requires the users to provide the compiled library when they submit their Spark application.

To explain the GPU batch processing interface, consider the code snippet in Figure 3.3, which shows how a filter operation can be offloaded to the GPU. The Scala wrapper only defines the signature of the method. The actual implementation is in C and calls a GPU kernel written in CUDA at line 5.

3.2.2 Batch Processing Overhead

Supporting batch processing on top of the Apache Spark execution model is not free. It introduces extra data movements for data analytics applications that wish to use GPUs.

```
// native function: Implements a filter operation on the GPU
JNIEXPORT jobject JNICALL
JAVA_filter(JNIEnv *, jobj) {
    ...
    filter_kernel<<<>>>();
    ...
}
```

```
// Scala wrapper for the native function
object GpuFilter(data: GpuRecord) {
  @native def filter(data: GpuRecord)
}
```

Figure 3.3: A sample filter operation implemented on GPU.

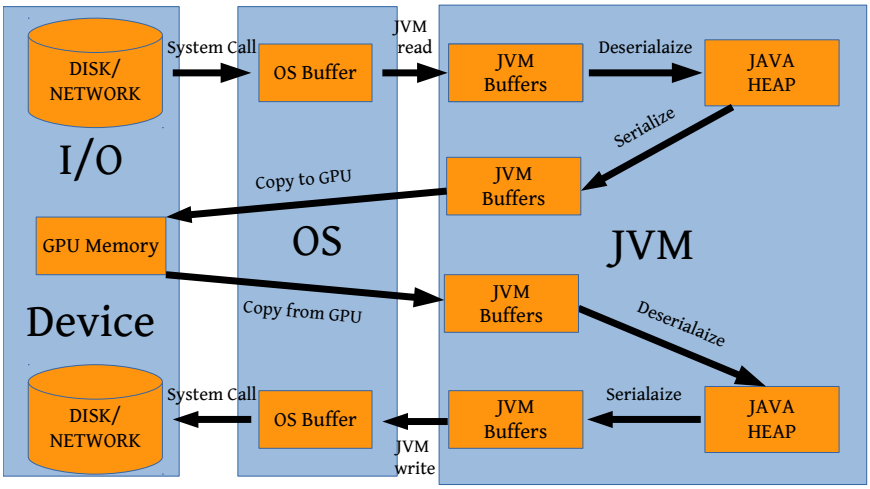

Figure 3.4: Data movements in Apache Spark when data is stored on the Java Heap.

We analyzed the overhead of the data movement for batch processing, which can help decide whether to offload an operation to GPUs or not.

In this section we will use the filter operation in figure 3.3 as an example to illustrate extra data movements in the GPU. The data movements and format transformation process is shown in Figure 3.4. We assume that the input data is in HDFS in a columnar format such as Parquet, RC file, or ORC. First the input data should be loaded into the native memory and then copied to the Java heap. Then it should be parsed and stored in either columnar format or row-wise format. To process the data on GPU, a GPU-RDD is required, which copies the data from Java heap to the native memory. If the data is not in columnar format, GPU-RDD will convert it to columnar format. When the GPU function is called on the GPU-RDD, the columnar data is transferred from the native memory to the GPU memory through the PCIe bus. The GPU function then processes the data on the GPU and transfers the results back to the native memory in the columnar format. Finally columns are copied back to the Java heap and stitched together to form Java objects, and rows.

Offloading the filter operation needs 6 data movements and transforms the data format twice. None of these operations are free and they add extra overhead to the execution. Data movements are expensive and should be avoided when possible. Considering the extra cost of data movements and transformations, an operation should be offloaded to the GPU only when it is computation-intensive. Consecutive GPU operations on the same data also can be chained together to make a computation-intensive workload.

3.2.3 Resource Management of Batch Jobs

We changed the Spark scheduling algorithm to handle nodes with GPUs and nodes without GPUs differently. When a worker node joins the Spark cluster, it communicates the number of GPUs it has to the resource manager to the cluster. We also added a customized

RDD, GPU-RDD. All the tasks for a GPU-RDD will be scheduled on the worker nodes with GPUs. If an RDD does not need GPU but one of its parent RDDs with a narrow dependency needs the GPU, Spark schedules all of them on worker nodes with GPUs to avoid shuffling data over the network. But if the dependency is wide and needs a shuffle, our modified scheduler can schedule the task on any worker node in the cluster.

The GPUs on a worker node are shared among all the tasks that are running concurrently on the worker node—a worker node with m CPU cores concurrently runs m tasks that share the GPUs.

We implemented utility methods that efficiently convert data in Spark RDDs to columnar format, which is suitable for GPU, and back to row-wise format after the computation is done. The GPU-RDD implementation has a `toColumnarRDD` and a `toRowRDD`.

We implemented two popular machine learning algorithms to demonstrate the usability of our API: K-Means and the Logistic Regression.

3.3 Processing Queries on GPUs

We implemented a set of GPU query operators to accelerate processing SQL queries on Apache Spark. We also extended the SparkSQL's Catalyst to build query execution plans with both CPU and the GPU operators. In this section we describe how we implemented these query operators.

3.3.1 Native Database Primitives

SparkSQL requires all query operators to implement a `doExecute` method, which returns and RDD of rows. The method will be called when the operator is executed on the worker nodes. We added a set of GPU query operators to extend SparkSQL. Each operator has a Scala wrapper and a native function. The Scala wrapper connects the GPU query operator

to other query operators and implements the `doExecute` method. The body of the wrapper method calls a native method that implements a database primitive. The database primitive we implemented are scan, join, aggregation, and sort. Our database primitives all accept the input in `GpuRecord` format and run on the GPU. The wrapper method converts the results to row format and releases the GPU memory. Our database primitives can be chained to build a wide range of SQL queries. They can be combined with SparkSQL primitives that run on the CPU. We will discuss our database primitives in rest of this section.

Native Scan

The scan primitive accepts a set of predicates and columnar data as input and returns all the tuples that satisfy the predicates. The returned tuples are in columnar format. The primitive first evaluates the predicates one by one and maintains a `0/1` vector. To avoid synchronization and data races, each GPU threads should know what part of the output it should write its results to. After predicate evaluation, a prefix sum is calculated on the `0/1` vector to find the start writing position for GPU threads.

Native Join

The join primitive is implemented as a primary key foreign key join. Our native join implements a hash join because it performs well on GPUs. The primitive accepts columnar data from a build table and a probe table and calculates the join results in two phases: a build phase and a probe phase.

The build phase passes the build column twice and generate a hash table without synchronization. In the first pass, each GPU thread counts the number of tuples it will write to each hash bucket Based on these counts, a prefix sum is calculated to decide the start

writing position in each bucket for each GPU thread. In the second pass, each GPU thread simply fills the hash table.

The probe phase generates a `0/1` vector to mark the tuples that should be projected in the join result. Then it calculates a prefix sum on the `0/1` vector such that the results can be generated without synchronization.

Native Aggregation

The aggregation primitive also uses a hash aggregation. It accepts its input data in columnar format. Our aggregation primitive first calculates a hash value for each group by key. Then it counts the number of groups based on the hash values, scans the input data, and applies the aggregate functions to each group. We used the standard GPU atomic instructions to synchronize GPU threads when calculating aggregation results. Because many GPUs do not support atomic operations on 64-bit words, we convert long and double values to float before carrying the primitive operations.

Native Sort

The sort primitive implements the bitonic sort. Since sort is usually executed after aggregation, the number of the tuples to sort is relatively small and GPU's shared memory can be use for sorting. Our sort primitive first sorts the keys for all the *order by* columns. After keys are sorted, our sort primitive generates the result with a gather operation.

3.3.2 GPU Query Operators

Using the native database primitive we described, we implemented five GPU physical operators: 1) GPU in-memory scan, 2) GPU broadcast join, 3) GPU hash join, and 4) GPU aggregation. Each query operator returns its results as an iterator of rows. We discuss these operators in detail here.

GPU In-Memory Scan

The in-memory scan works on columnar cached data that is stored on Java Heap. This operator simply passes the data, the schema, and the predicates to the native scan, which runs on the GPUs.

GPU Broadcast Join

The broadcast join joins two table in three phases. First it assigns the smaller table as the build table and the larger table as the probe table. Then, it broadcasts the build table to all worker nodes. Finally, worker nodes call the native join primitive on the GPUs for probe table's partitions and join them with the build table.

GPU Hash Join

The hash join operator first repartitions the data based on the join keys. Then for each part of the partitioned data, the operator batches the data into a `GpuRecord` and calls the native join on the GPUs.

GPU Aggregation

The aggregation operator is implemented in two phases: a partial phase and a global phase. The partial phase locally aggregates the results for partitions on worker nodes. Thee global phase combines and aggregates all the local aggregations to compute the final results. The partial aggregation phase batches the data into `GpuRecord`s and calls the native aggregation on GPU. Partial aggregation returns the results as an iterator of rows. The global aggregation first shuffles data and then aggregates the results on the CPUs.

```
1 FROM lineorder, supplier
2 WHERE lo_suppkey = s_suppkey
3   AND s_region = 'ASIA'
4   AND lo_discount < 5
```

Figure 3.5: A sample SQL query based on query 1.1 in the SSB benchmark.

3.3.3 Optimizations

To achieve optimal query performance, we should carefully place the query operators on the cluster CPUs or GPUs. CPU operators outperform GPU operators when data is small or is in row format. GPU operators perform best when the input data is large and in columnar format. To explain how data format and size effects performance of query operators, consider the following query.

This query first scans a large table, *lineorder*, and a small table, *supplier*, and joins the tuples that satisfy the scan predicates. When executed on the GPUs, this query can be processed with two GPU in-memory scans followed by a GPU broadcast join. The GPU in-memory scan on the lineorder table copies the results from the native memory to Java heap and materializes them into columnar format. The GPU broadcast join fetches the tuples from Spark's iterator interface, batches the data into columnar format, and copies the data from Java heap to the native memory. The join operation runs on the GPU. It is unnecessary to copy tuples in the lineorder table that satisfy the join condition because they are already in columnar format and can be directly used in the GPU broadcast join.

The problem is that the Spark interface expects the input and output of each SQL operation in row format. To solve the problem, we added a new method, `gpuDoExcecute()`, to the interface of all GPU operators. The `gpuDoExecute()` method executes the operator on GPUs and returns the results in columnar format in the native memory. After adding

the `gpuDoExecute()` interface, GPU operators check their input data. If their input data is generated by another GPU operator, they directly call `gpuDoExecute()` and gets their input in columnar format. Otherwise they call the `doExecute()` and bactch the data into columnar format before processing them on GPUs.

The `doExecute()` method in GPU operators calls the `gpuDoExecute()` method and returns the results as an iterator over rows. The row iterator in a GPU operator lazily materializes the data when the row is needed. With the `gpuDoExecute()` interface, our GPU operators can avoid many unnecessary data copies between GPU and CPU.

3.3.4 GPU-Aware Catalyst

We extended SparkSQL's Catalyst to generate queries execution plans with both CPU operators and GPU operators. Catalyst is the query optimizer in SparkSQL. Given an SQL query, Catalyst first uses a set of rules to generate the optimized logical plan and then generates the physical execution plans with a set of strategies. Curently, catalyst does not have a cost model to estimate the performance of physical plans. Therefore it simply picks the first physical plan to execute the query.

Catalyst is extensible. To make Catalyst GPU-aware, we added our GPU strategies that generate physical plans with GPU operators. Spark-GPU guarantees that the physical plan with GPU query operators is the first physical plan among all plans and, thus, will be used to run the query. The overall structure of Catalyst remains unchanged. Figure 3.6 shows the optimization process.

The GPU strategies decide whether a query operator should be executed on GPUs. The following GPU strategies are added to Catalyst.

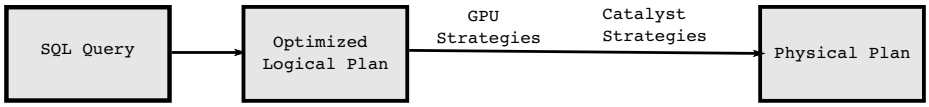

Figure 3.6: Architecture of GPU-aware Catalyst

- Join operators are offloaded to GPUs. If the size of one table is smaller than Spark's broadcast join threshold, GPU broadcast join is used, otherwise GPU hash join is used.
- The children of a broadcast join are offloaded to the GPUs whenever possible.
- If an aggregation can be divided to a partial aggregation and a aglobal aggregation, both the partial aggregation and its children are offloaded to GPUs.

We offload the operators to GPUs if there is a chain of GPU operators that process the same columnar data before converting the data back to row format. Additionally, each operator should benefit from running on the GPUs. Otherwise we do not offload the operators to the GPUs. Having a chain of operators to offload to the GPUs is crucial because otherwise converting the data back and forth to columnar format would impose high overhead. For example, the performance of the join operator is bound by memory accesses on the CPU. It can benefit from GPU's high bandwidth, thus it will be offloaded to GPUs. But if a query contains only a scan operator, it will be executed on the CPUs. Scan will be offloaded to GPUs if it is followed by a GPU broadcast join, a GPU aggregation, or a GPU sort, where at least two GPU operators can work on the columnar data.

3.4 GPU Resouce Management

A straightforward approach to manage the GPUs in a cluster is to treat a GPU as a CPU core. When a task requires a GPU, a GPU will be exclusively given to the task until the task finishes. This coarse-grained management approach has been adopted in MapReduce Hadoop systems (e.g. [18]) even though it underutilizes GPU resources, which degrades the performance of GPU applications.

To illustrate the problem, we implemented a synthetic GPU workload on Apache Spark and measured its performance with different configurations. We changed the number of tasks that concurrently run on the GPU. The workload transfers the data to the GPU memory, reads it a number of times, and generates a constant number of results. We control the number of times that data is read to control the computation to PCIe transfer ratio. We used three different computation/PCIe ratios: 1:1, 10:1, and 100:1. We conducted the experiment on a 16-core node with one GPU. The maximum number of concurrent GPU tasks in our experiment was 16 (one per CPU core). The input data were partitioned into 16 partitions and was stored on HDFS. The size of each partition was 128MB. We normalized the experiment results to the configuration with one concurrent GPU task as the baseline. The execution results are shown in Figure 3.7.

Two observations can be obtained from Figure 3.7. First, GPU sharing improves the performance. When computation/PCIe ratio is 1:1, performance improves more than 2X.

Second, increasing computation/PCIe ratio has diminishing returns. The fundamental factor that determines the benefits of CPU sharing is the amount of work that can be executed in parallel when running tasks on GPUs. To offload an operation to GPUs, we cook the data into GpuRecords, transfer data between native memory and GPU device memory through the PCIe bus, and execute kernel GPU on GPUs.

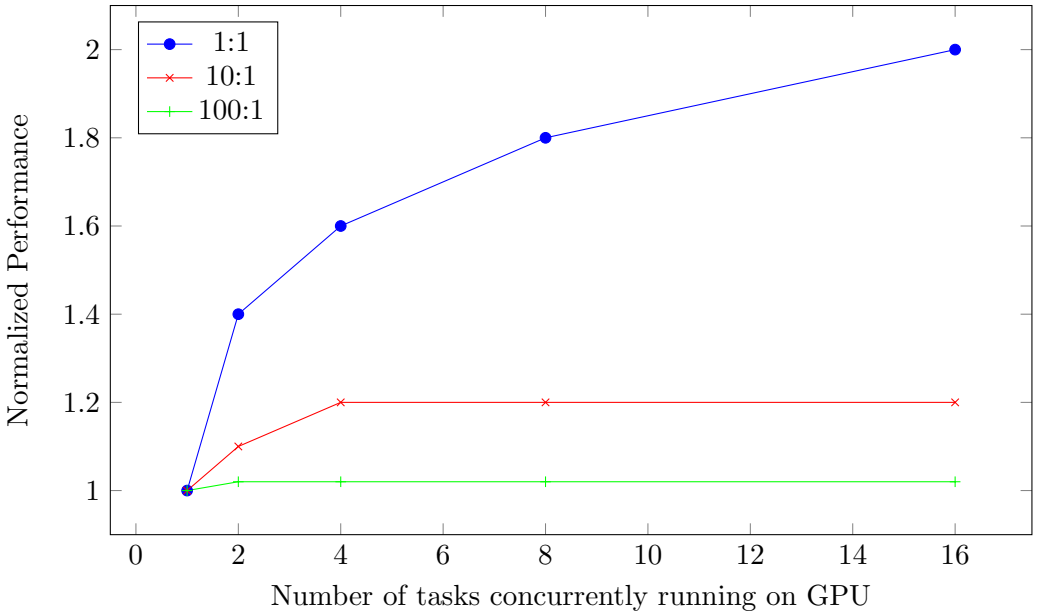

Figure 3.7: Effect of computation to PCIe transfer on the performance of GPU

Among these operations, only two combinations are executed serially. First, kernel executions can hardly be executed in parallel due to GPU's LEFTOVER resource management policy [34]. Second, PCIe data transfer in the same direction must be executed serially due to hardware limitations. All other combinations of operations can be overlapped and thus benefit from sharing GPUs.

GPU Memory Management

For a general purpose data analytics system such as Apache Spark, sharing GPUs is essential to speed up as many workloads as possible. The major obstacle for sharing GPUs in the cluster is that operating system and existing GPU drivers do not support virtual memory. As a result, when a GPU is shared by multiple tasks, GPU can run out of memory and task can crash. Some task crashes can be avoided by tuning the partition size

in the cluster, but the system needs to manage GPU's memory to concurrently execute GPU tasks. [44] Prior research such as [22, 44, 44] addresses the this problem. We implemented a user level GPU memory management library based in the framework proposed in [44] because GPU management in Apache Spark should be transparent to the GPU program and the operating system.

The library works as a layer between the GPU and the standard GPU library. It intercepts all GPU memory related system calls—GPU memory allocations, free, and kernel launches—from the task running on the node. The library creates *regions* to manage GPU memory. A *region* contains a GPU buffer and a CPU buffer. The CPU buffer is created lazily and stores the data in the GPU buffer when the library spills the data out of the GPU memory. When there is not enough GPU memory for new allocations, the library swap outs an existing region into the CPU memory.

Each worker node in the cluster needs to load the GPU memory management library when it starts. With the GPU memory management library, each GPU task has the illusion that it can use the whole GPU memory even though the GPU is shared.

GPU Task Scheduling

We modified Apache Spark to use and share available GPUs in the cluster. Our implementation abstracts each GPUs into multiple logical GPUs. Each logical GPU can run one GPU task at a time. GPU sharing granularity is configurable. Users need to explicitly set the number of available GPUs on each node. They also need to set how many tasks are allowed to be executed concurrently on each GPU provided that the total number of GPU tasks do not exceed the total number of CPU cores on the node.

Our implementation schedules tasks based on the RDD lineage graph. The scheduler checks if a stage contains any RDD that is created by GPU operators. If a stage contains

such operator, it will only be scheduled to nodes that have GPUs and that have at least one available CPU core.

Chapter 4: Experiments

In this chapter we evaluate the performance of our implementation to highlight the strengths and limitations of using GPUs for processing big-data. The experiments have demonstrated that:

- Batching data for GPUs incurs overhead that can be prohibitively high and diminish the benefits of offloading heavy tasks to GPUs.
- Performance of SQL queries improves by 2.23X and performance of data mining and statistical workloads improves by up to 12.28X when GPUs are utilized in the system.
- Sharing GPUs marginally improves the performance of data mining and statistical workloads, but it can improve the performance of SQL queries further by up to 1.53X.

We first describe the experimental environment and the workloads used in the experiments. Then we represent the experimental results.

4.1 Experimental Environment and Workloads

We conducted all the experiments on a cluster with 9 nodes on Amazon EC2 instance type of *g2.x2large*. Each node has 8 CPU cores, 15GB memory and one NVIDIA GPU with 4GB of GPU device memory. The OS on each node was Ubuntu 14.04, with NVIDIA driver

320.48 and CUDA 6.5. We configured the cluster to have one dedicated master node for Spark and HDFS. The remaining 8 nodes were configured as worker nodes.

We measured the overhead of batch processing and examined performance of our implementation for machine learning workloads. For machine learning workloads, we used K-Means ad logistic regression. The dataset of logistic regression had 2 million data points with 1024 features per data item. We implemented both K-Means and logistic regression using the GPU-RDD API. We also used SQL queries from the Star Schema Benchmark [32] and the TPC-H benchmark [11]. We set the scale factor to 50 for both benchmarks.

We cached the data in the cluster memory for all workloads. For each workload, we measured the performance on Spark 1.4.0, and with 1, 2, 4, and 8 concurrent task on a GPU. We repeated each experiment 5 times and report the median results.

4.2 Batching Overhead

Batching overhead stems from two parts: 1) batching objects and rows into `GpuRecord`s, 2) converting `GpuRecord`s back to rows and objects. To measure the batching overhead, we used a micro benchmark that batches sets of rows into `GpuRecord`s and then converts `GpuRecord`s back to rows. We set the total number of rows to 2 million and used fixed-length string columns of with length 16. We used the string type because creating string object in Java heap is expensive. We varied the number of columns and measured the execution time of the micro benchmark. For comparison, we also reported the execution time of directly projecting all the rows in Spark. The results are show in Figure 4.1.

The micro benchmark demonstrated that the batching operation is expensive. The overhead increases with the size of batched data. With 8 string columns in a row, the execution time of batching rows is 18.3X longer than projecting the rows. The execution

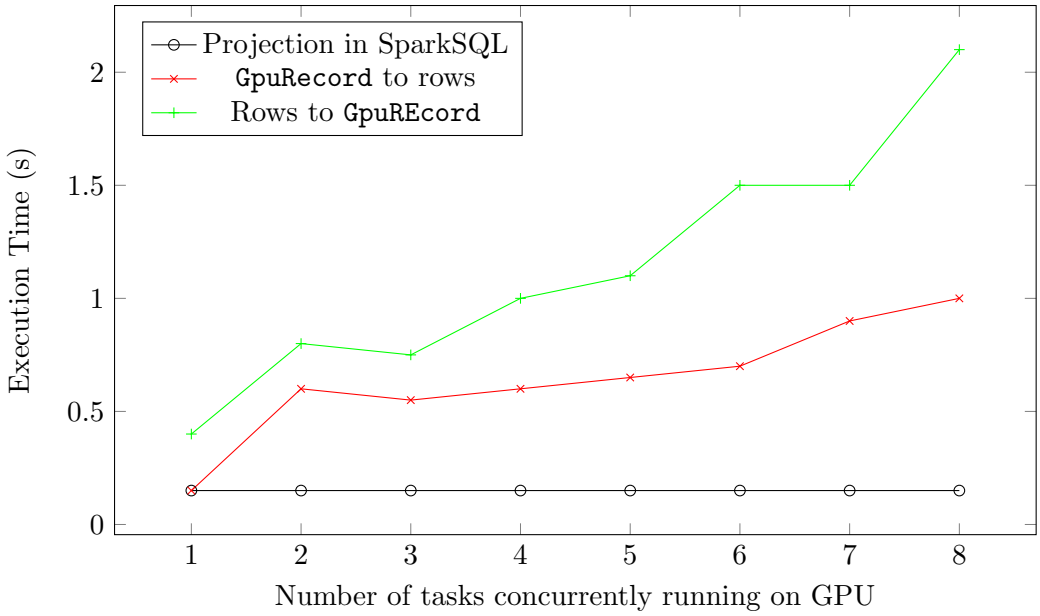

Figure 4.1: The overhead of batching data into columnar format

time of converting the `GpuRecord`s to rows is 10.5X longer. Batching overhead is high because a Java String Object cannot be directly copied to the native memory. The string object must first be converted to a byte array and then can be copied byte by byte to the native memory. This micro benchmark also corroborates that if an operation is not "computation-expensive", it should not be offloaded to the GPUs.

4.3 Data Mining

K-Means is a popular clustering algorithm. We implemented it both on Spark and Spark-GPU to study its performance. The algorithm has two steps: 1) find the closest center to each data point, 2) update the cluster centers using the data from step 1. We initially used the `GpuRecord` API to implement K-Means on GPU, but this implementation was inefficient because our dataset had a large number of columns (1024 columns in our

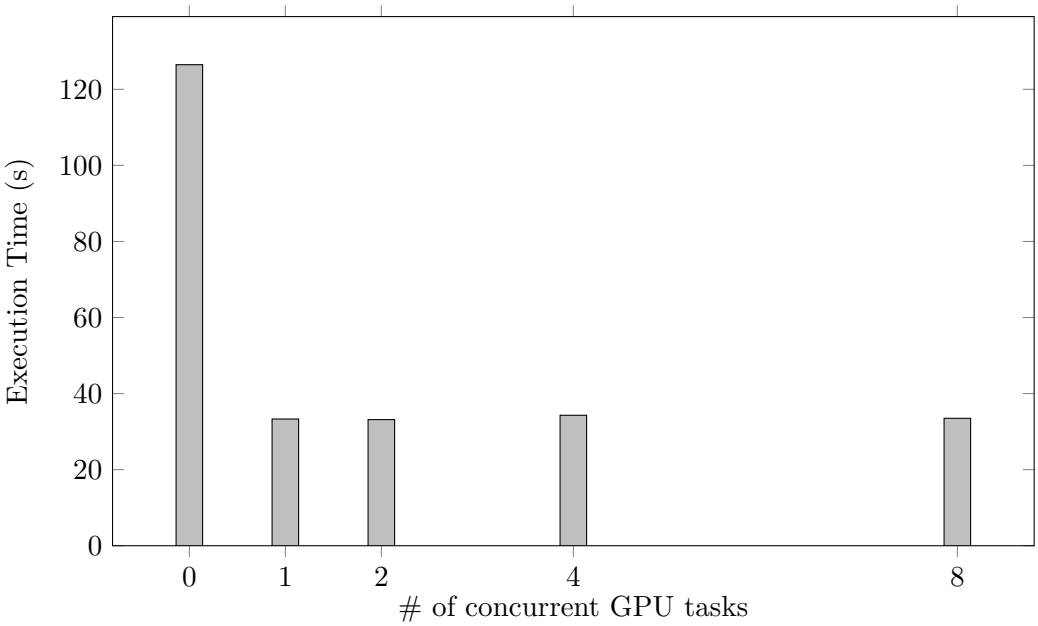

Figure 4.2: Performance results of executing one iteration of K-Means on Spark with different GPU sharing configurations.

experiments). To copy each column to the GPU device memory, a separate GPU memory command copy is needed. Instead, we allocated one continuous buffer for all data points in columnar format in GPU. With a continuous buffer, only one GPU copy command is needed to transfer the data to the GPU device memory. For each iteration of the computation, we offload the operation of calculating the closest centers to the GPUs. A GPU kernel is launched to find the closest center for each data point in one partition and aggregates on the centers locally. Then a global aggregation updates the centers.

Figure 4.2 shows the performance results. K-Means on Spark-GPU significantly outperforms K-Means on Spark without GPU. The largest improvement is 3.82X, which is significant because K-Means is computation-intensive and it shuffles a limited amount of

data. It only shuffles centers in each iteration. The dominant operation in K-Means is finding the closest center for each point, which is suitable for running on the GPUs.

Sharing GPUs for K-Means only improves the overall performance by up to 7%. The reason is that the performance of K-Means is bounded by kernel execution—and GPU kernels rarely run concurrently.

4.4 Statistical Analysis

Logistic regression is a commonly used classification method in statistical analysis. The algorithm finds the hyper-plane p that divides a dataset the best. If the hypothesis for the classification is given with

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)} \tag{4.1}$$

where $x, \theta \in \mathbb{R}^{n+1}$, the normal vector for p has is proportional to θ. The objective in Logistic Regression is to find the best θ in formula 4.1. Even though there are exact mathematical solutions to find θ, they are too expensive to be used in practice. The most common way to find θ is through an iterative algorithm that incrementally drives θ to the best value. In each iteration of the computation, the gradient of the current hypothesis should be calculated to adjust the value for θ. Computing the gradient is expensive because it requires computing the lengthy sum

$$\nabla_\theta l(\theta) = \sum_{i=1}^{m} \left(y^i - h_\theta(x^{(i)}) \right) x_j^{(i)} \tag{4.2}$$

over all data points in the dataset. This part of the computation is CPU-thirsty and can be accelerated on the GPUs.

Similar to the K-Means algorithm, we stored all the data points in columnar format in one continuous buffer for each partition.

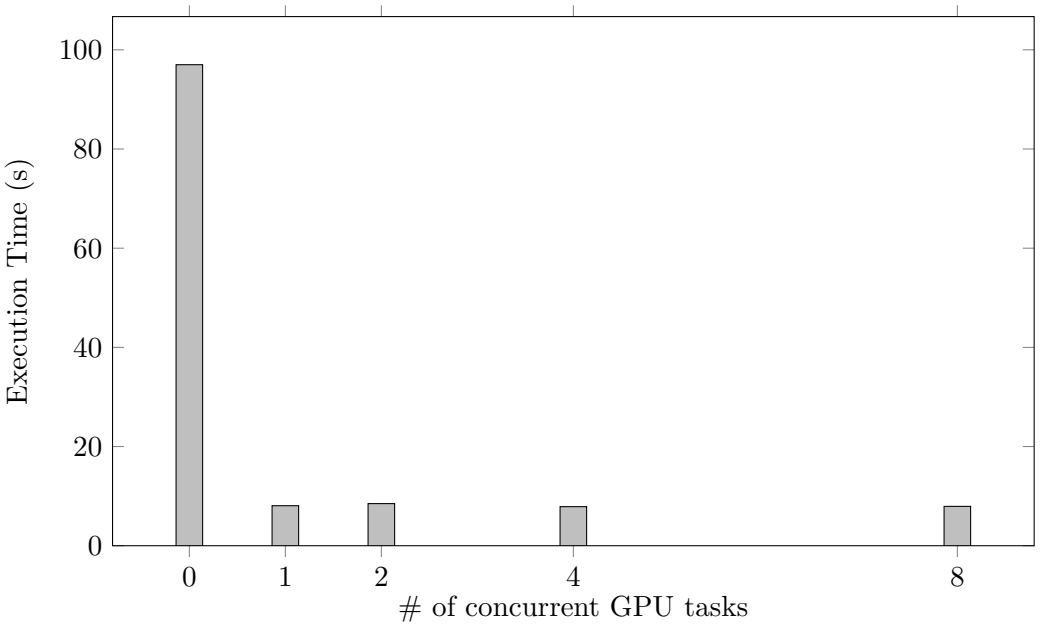

Figure 4.3: Performance results of executing one iteration of logistic regression on Spark with different GPU sharing configurations.

Figure 4.2 shows the performance results, which are similar to the results for K-Means. Offloading operations to GPU can improve performance of logistic regression on Spark up to 12.28X. The logistic regression algorithm needs to calculate the partial derivative The performance of logistic regression is bound by calculation in each data point, Sharing GPUs marginally improves the workload performance.

4.5 Shuffle-Rare Query

The Star Schema Benchmark (S.S.B.) has one fact table, *lineorder*, and four dimension tables, *date*, *part*, *supplier*, and *customer*. We used query 3.1 from the benchmark, which has the longest execution time, to study Spark performance with GPU.

```
1  SELECT c_nation,s_nation,d_year,SUM(lo_revenue) AS revenue
2  FROM customer,lineorder,supplier,ddate
3  WHERE lo_custkey = c_custkey
4  AND lo_suppkey = s_suppkey
5  AND lo_orderdate = d_datekey
6  AND c_region = 'ASIA'
7  AND s_region = 'ASIA'
8  AND d_year >=1992 AND d_year <= 1997
9  GROUP BY c_nation,s_nation,d_year
10 ORDER BY d_year ASC,revenue DESC;
```

Figure 4.4: S.S.B query 3.1

Query 3.1 joins three dimension tables with the fact table, groups by, and sorts the results. Since all dimension tables are much smaller than the fact table, the best way to execute the joins is using broadcast join. Since SparkSQL cannot accurately estimate the join selectivity, it uses shuffle hash join, which performs slower than broadcast join for query 3.1. We manually set the query plan to be broadcast join in the experiment. The overall join selectivity is 3.4%. The query execution time is dominated by the join operations. Figure 4.5 shows the physical plan for query 3.1.

Apache Spark performs better when the query is executed on GPUs. The performance was optimum with 2.28X speed up when GPUs were configured to run 4 concurrent tasks. GPUs exploit two characteristics of this query to gain significant performance benefits. First, a large portion of the query execution time is spent on the joins, which significantly benefits from running on the GPUs. This is despite the fact that shuffle operations, broadcast operations, global aggregations, and global sorts do not use the GPUs. Second, the output of each GPU operator can be used by the next GPU operator without converting the data back to row format, which eliminates overhead of converting intermediate data back and forth to columnar and row-wise format.

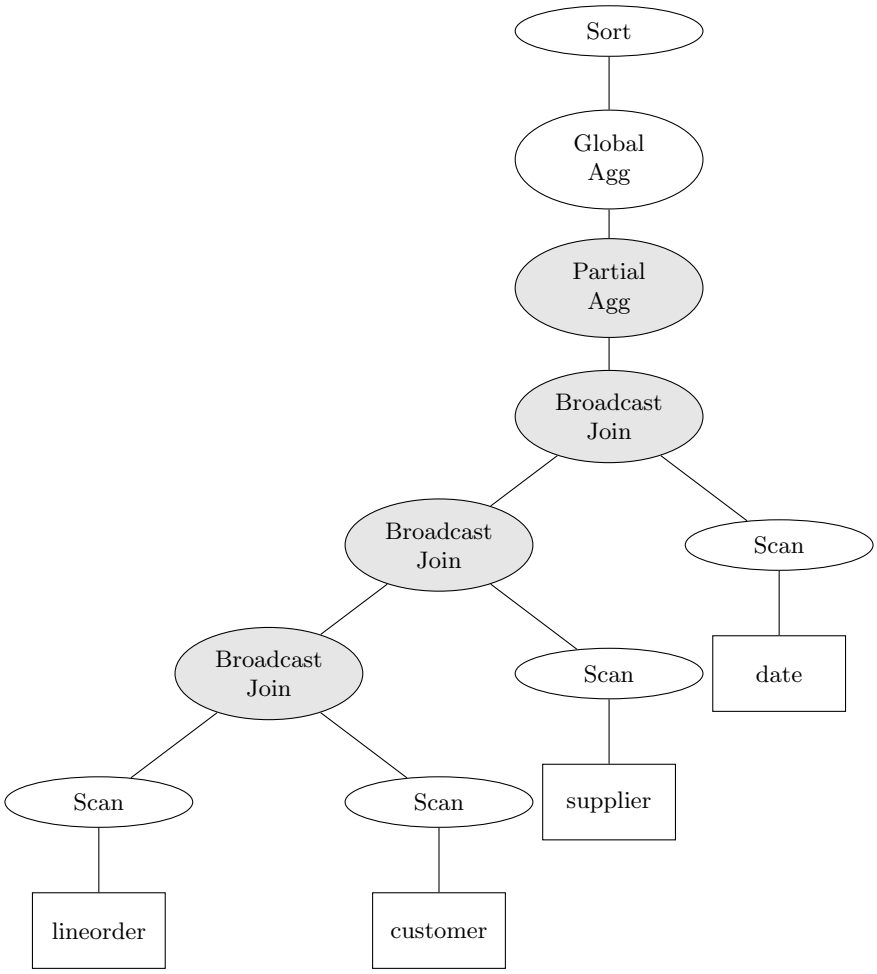

Figure 4.5: The physical execution plan for SSB 3.1 query. Rectangle nodes are input tables. White nodes are executed on the CPU and shaded nodes are executed on the GPU.

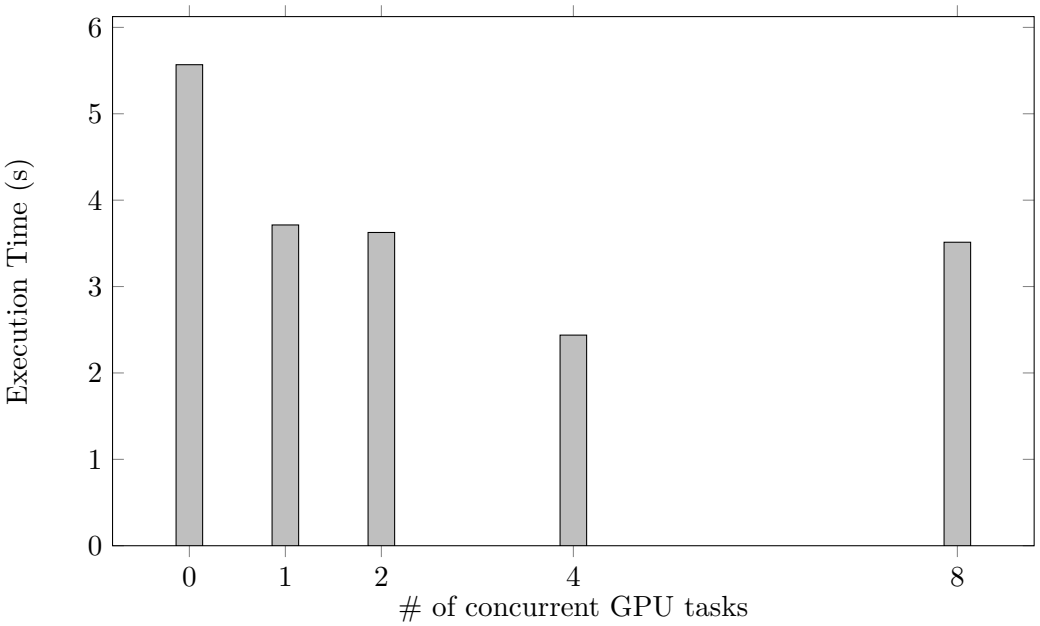

Figure 4.6: Performance results of executing SSB query 3.1 on Spark with different GPU sharing configurations.

The experiment also demonstrates that sharing GPUs can improve the performance of GPU workloads. Compared to running one concurrent GPU task, running 4 concurrent GPU tasks increases the performance by 1.53X. But when a GPU is shared among 8 concurrent tasks, the performance degrades to the level of using only 1 concurrent GPU task. This degradation is to be blamed on current GPU drivers, which manage GPU resources poorly. This results shows that we should avoid oversubscribing GPU resources.

4.6 Shuffle-Intensive Query

The TPC-H benchmark has a snowflake schema. Queries in the benchmark have more complex behavior than the star schema queries. We used query 3 from the benchmark to examine the performance of Spark with GPU.

```
1  SELECT
2          l_orderkey,
3          SUM(l_extendedprice * (1 - l_discount)) AS revenue,
4          o_orderdate,
5          o_shippriority
6  FROM
7          customer,
8          orders,
9          lineitem
10 WHERE
11         c_mktsegment = 'MACHINERY'
12         AND c_custkey = o_custkey
13         AND l_orderkey = o_orderkey
14         AND o_orderdate < '1995-03-26'
15         AND l_shipdate > '1995-03-26'
16 GROUP BY
17         l_orderkey,
18         o_orderdate,
19         o_shippriority
20 ORDER BY
21         revenue DESC,
22         o_orderdate
```

Figure 4.7: TPC-H query 3

Query 3 joins three tables, *orders, customer*, and *lineitem*, and aggregates and sorts the join results. Since none of these table are small enough for broadcast joins, we used shuffle join. Figure 4.8 shows the queries physical plan. When executed on Spark-GPU, joins and the aggregation were offloaded to GPUs. The scans were executed on CPUs.

Figure 4.9 shows the results. Using GPU can improve the query performance for all GPU configurations, with the biggest improvement at 1.23X with 8 concurrent GPU tasks. The performance benefits of the TPC-H query are less compared to the SSB query because the data shuffle operation in the shuffle join dominates the query execution time. A data shuffle operation writes the data to disks before data shuffling and reads from the disk before joins. Disk I/O is expensive and cannot be accelerated by GPUs.

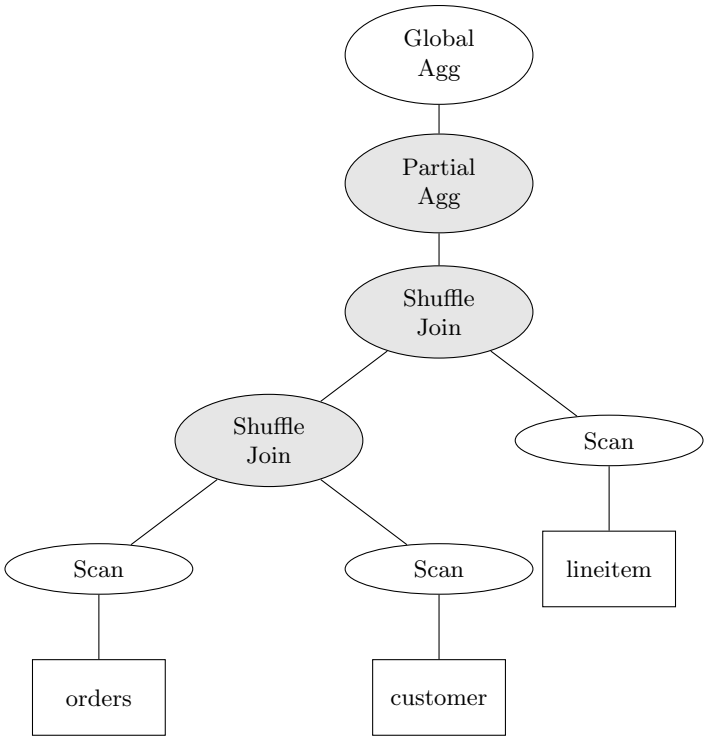

Figure 4.8: The physical execution plan for TPC-H 3 query. Rectangle nodes are input tables. White nodes are executed on the CPU and shaded nodes are executed on the GPU.

Sharing GPU's does not help improve the query performance. All GPU sharing configurations have similar performance since GPU operations do not dominate the execution.

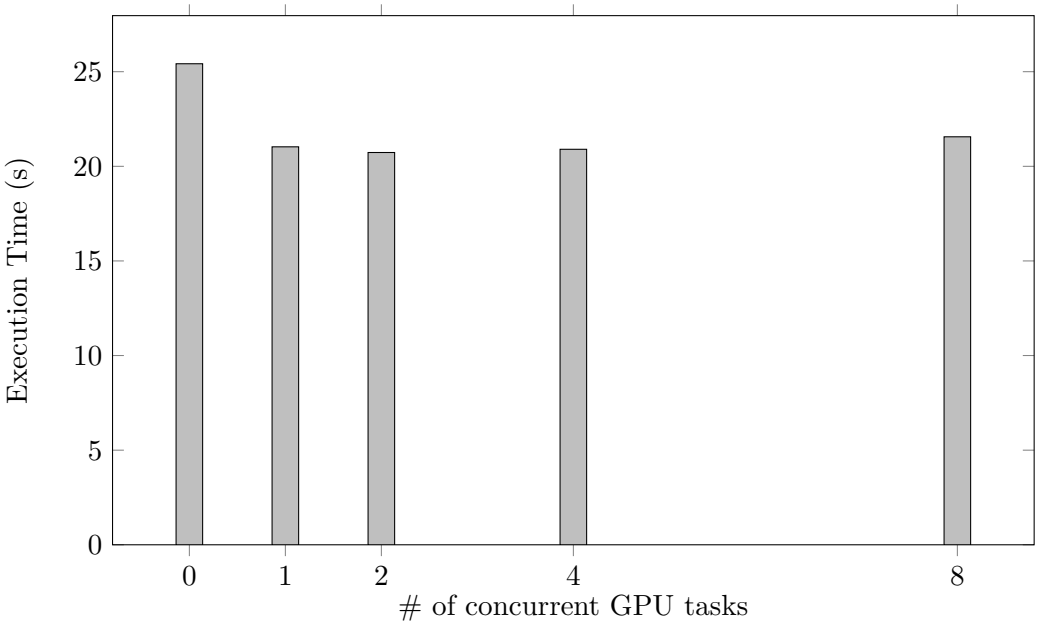

Figure 4.9: Performance results of executing TPC-H query 3 on Spark with different GPU sharing configurations.

Chapter 5: Related Work

In this chapter we examine two bodies of related work: work on building scalable, fault-tolerant data analytics systems, and work on using GPUs to speed up applications.

5.1 Large Scala Systems

The body of research on making large-scale systems is both broad and deep. Any aspect of a large-scale system, such as security, check-pointing, load balancing, scheduling, voting, fault-tolerance, is a challenge and has been studied extensively. In this report we are interested in two aspects of a large-scale system: the storage layer, and the execution engine. The storage layer interests us because we have expanded the storage format of Apache Spark to add columnar format support. Section 5.1.1 discusses large-scale storage systems.

The large-scale execution engines interest us because we have extended the execution engine of Apache Spark to offload computation-intensive queries to the GPUs. We are particularly interested in execution engines that support SQL or RDBMS like operations. Section 5.1.2 discusses large-scale data analytics systems.

5.1.1 Large-Scale Storage Systems

Google File System [15] started a paradigm shift in the designing large-scale systems. It relied on using inexpensive commodity hardware to build high throughput systems. In

this new paradigm, component failure is not rare, it is the norm and should be addressed in the system design. Google File System avoided complicated fault tolerance solutions and opted for a simple and straightforward design. In their design, they replicated data on three nodes. If a task fails on a node, they restart the task on a replicate node. This core design decision was followed in the MapReduce model [12], and many other large-scale data analytics systems. We will discuss some of these systems in this section.

BigTable [9], is another large-scale storage system from Google, which is built on top of Google File System. It is designed to serve both batch-processing jobs and real time jobs. BigTable users can request the data to be stored either on the disk or tin the main memory. All entries in BigTable are addressed with a row name, a column name, and a timestamp. The timestamp is to keep multiple versions of the data. To achieve low latency, BigTable provides strong atomic and consistency guarantees at the row level and weaker guarantees across rows, which makes it ill-suited for applications with strong consistency requirements. It also keeps the latest event logs in the main memory, which it calls a memtable, and writes it to the disk periodically. The persisted event logs on the disk are compacted and merge to clean up disk and to make recovery from failure quick.

Megastore [4] extends the BigTable guarantees to provide ACID semantics within fine-grained partitions of data but weaker guarantees across partitions. If users select partitions carefully, the weak cross-partition guarantees of Megastore will be enough for most applications even though the read throughput of Megastore is not comparable to BigTable.

Spanner [10] is another extension to BigTable that provides low-latency reads and external consistency across multiple datacenters. Spanner uses the wall clock to timestamp

writes and orders the commits based on the timestamps, which makes providing external consistency across datacenters practical. Using timestamps that represent actual time, Spanner can order commits and events without resorting to complex solutions.

Amazon uses Dynamo [13] to store users shopping data. Dynamo is a key-value storage system and guarantees consistency at key-value pair level, but it relaxes consistency guarantees across pairs to achieve high availability. Facebook's Cassandra [27] is based on Dynamo with added features, such as memtables and compaction, from BigTable [9].

5.1.2 Large-Scale Data Analytics Systems

MapReduce [12] was the first large-scale framework made of commodity hardware. Each job in MapReduce is made of a *map* phase followed by a *reduce* phase. In the map phase, each input element is transformed into a `key-value` pair. In the reduce phase, all the values with the same key are combined to get the final results. More than one set of map reduce phases can be chained together to do more complex computations. The MapReduce framework writes the output of each phase to the disk to make recovery from failures easy, but it makes MapReduce slow compared to other large-scale data analytics systems.

The design decisions behind MapReduce made it affordable for many organization. For example, New York Times scanned its 1851–1980 archive into PDF files using Hadoop [46], the most popular implementation of MapReduce, in less than 24 hours. The cost of scanning all the archive would have been too high on any system prior to MapReduce.

Dremel [29] is an interactive data analytics system for read-only data. It can extract the schema for complex objects with nested data structures and convert them to columnar format. Its query execution engine schedules tasks based on the locality of the data. Unlike Apache Hive [43] and Latin Pig [31], it executes native queries without translating them into

MapReduce jobs, which are slow because they access disk at each Map/Reduce operation. Dremel can achieve low latency performance by avoiding unnecessary accesses to the disk.

F1 [40] is a data analytics system with strong schema enforcement, high availability, and strong consistency. F1 is build on top of Spanner[10] and has a high write latency, but batching reads/writes and using hierarchical schemas can hide most of the latencies. F1 servers are spread geographically across datacenters and they communicate with each other but no F1 server communicates with the worker nodes in another datacenter. To achieve high availability and quick recovery from failure, F1 servers are mostly stateless. The query execution engine of F1 runs the queries either in a centralized form or a distributed form. F1 distributes query executions only when the estimated increase in parallelism yields more benefits than the latency of distributed execution.

Impala [14] is also an SQL query engine that translates queries to native code using LLVM [28]. Impala's I/O sub-system and pipelined query execution are highly optimized for running queries on columnar storage format. Moreover it does not pay the scheduling overhead and data materialization overhead that MapReduce based systems impose. For the I/O bound workloads, Impala is 3X faster than Hive. Queries that need multiple MapReduce phases in Hive are 45X faster in Impala. Impala's performance benefits over Hive widens to to 90X when the workloads working set fits in the main memory.

Apache Spark [50] is a general purpose data analytics framework that, unlike its predecessors, support iterative dataflow models. It keeps datasets in the main memory and reuses them for efficiency. Apache Spark has separate components to process SQL queries, graphs, and machine learning, but the core component of Spark can run any application written to manipulate data. We discussed Apache Spark in detail in chapter 2.

5.2 GPUs in Non-Graphic Applications

In the previous section, we discussed some of the most important large-scale data analytics systems, some of which are used at web-scale by big companies. In this section we will discuss work that uses GPUs to accelerate applications. GPUs have been used to accelerate many applications and to survey all of them is beyond the scope of this report, but we are particularly interested in using GPUs for accelerating query processing and data analytics systems.

5.2.1 GPUs in Query Processing Systems

Database management systems have studied and used GPUs to accelerate query execution. To accelerate join operations, Kaldewey et al. [23] have used zero-copy accesses and saturated the PCIe bandwidth. Their join implementation gains maximum PCIe bandwidth that is within 2% of the theoretical limit. Bingsheng He et al. [19] have implemented a set of highly tuned and aggressively optimized primitives and used them to accelerated joins on GPUs. They achieved 2X-7X speed up over optimized CPU-based join implementations.

Satish et al. [37] have exploited the high bandwidth of GPU and have improved the performance of both comparison and non-comparison based sorts. They have implemented two GPU-based sorts: a radix sort that performs well for smaller sets and a merge sort that performs better for larger sets. TeraSort [16] have used the numerous GPU cores to implement a parallel bitonic sort. Their TeraSort sort on mid-range GPUs is competitive with sort algorithms on high-end CPUs and handles datasets that do not fit into the GPU memory or the main memory and scales to billions of records. Other studies have shown that performance of aggregation and selection operations can be improved using GPUs [24].

Finding the optimal query plan is an NP-Hard problem and is a new bottleneck in real-time data analytics systems. Sen et al. have used GPUs to speed up query planning [38].

Wu et al. [47] have used a technique called kernel fusion that reduces data movements between the main memory and the GPU memory. They have identified a set of common patterns in GPU kernels which can be fused together. They have proposed a compiler framework that automatically fuses relational algebra operators and eliminates redundant data movements. Using their compiler framework, they gained more than 2X speed up in kernel execution and PCIe transfer time.

All this body of research accelerates individual operators using GPUs, but they all focus on improving individual computational tasks. Yuan et al. [49] have integrated GPUs to fully functional RDBMS and have studied the performance benefits of using GPUs under different architectures and system designs. They have studied various optimization techniques, hardware architectures, and compression algorithms, and how they impact the performance of the RDBMS.

Wang et al. [45] have studied the benefits and limitations of sharing GPUs among concurrent tasks for query processing systems. In a different work [44], they have proposed a user user library for automatically managing GPU memory among concurrent tasks.

None of the related work so far is applicable in large-scale data analytics systems. Dandelion is a system that aims at running large-scale data analytics systems on GPUs. It is a compiler and runtime system that generates code to run on both CPUs and GPUs [35]. Programs in Dandelion are written in LINQ [42] and are translated to CUDA code for a subset of C# programs. Dandelion's CUDA code generation is restricted to method that do not dynamically allocated memory.

Facebook's Torch [1] is a large-scale system for machine learning. Developers script their programs in LuaJIT for Torch, which will be automatically translated to C/CUDA and will be executed on the underlying cluster. Unlike any other system that we have discussed so far, which all are query processing engines, Torch mainly targets machine learning algorithms such as neural networks. The main downside with torch is the sharp learning curve for LuaJIT.

Chapter 6: Conclusion and Future Work

In this report, we discussed the limitations of large-scale systems. We showed that current trends in workload features and hardware capabilities are making CPUs the bottleneck in the system. We implemented a system on top of Apache Spark that wisely uses the columnar storage format to reduce the unnecessary work assigned to the CPU cores. We also used the same columnar format to integrate GPUs as a computational resource to Apache Spark and discussed its challenges.

We evaluated our implementation for 4 different workloads and assessed suitability of our implementation in each workload. We used a query processing workload that mostly accesses data locally on the worker nodes and a query processing workload that needs to accessed data on other worker nodes. We also used two machine learning workloads that are computationally intensive and CPU-thirsty. We showed that for all the workloads, columnar storage and GPUs can improve the performance. The performance improvement is significant for machine learning workloads, but it is marginal for query processing workloads.

For future work, we plan to integrate GPUs into more components of the large-scale systems. Many large-scale systems, including Apache Spark and Hadoop, use common "primitives" during job execution. One such primitive is sort-based shuffle. We want to extend our implementation to speed up such common "primitives".

Our Reflection-Based-API automatically infers the columns for an RDD based on the RDD type, but the developer needs to explicitly specify the RDD type in many cases. Specifying the type of an RDD can become very tedious and error-prone, particularly for RDDs that use complex expressions. For example joining two RDDs on a key can easily take 50 lines of Scala code. We want to extend our Reflection-Based-API to infer the RDD type automatically based on the RDD's parents and expressions. Scala has extensive support for type inference at compile time, but the type inference system falls short for more complex types. We want to use the more advanced features of Scala, such as virtual Scala and macros, to add automatic type inference to our Reflection-Based-API to make it more usable.

Bibliography

[1] Torch: http://torch.ch.

[2] Apache Parquet: https://parquet.apache.org, 2015.

[3] B Alpern, S Augart, S M Blackburn, M Butrico, A Cocchi, P Cheng, J Dolby, S Fink, D Grove, M Hind, K S McKinley, M Mergen, J E B Moss, T Ngo, and V Sarkar. The Jikes Research Virtual Machine Project: Building an Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, jan 2005.

[4] Jason Baker, Chris Bond, James C Corbett, J J Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, volume 11, pages 223–234, 2011.

[5] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1021–1032. VLDB Endowment, 2004.

[6] Michael D Bond, Milind Kulkarni, Man Cao, Meisam Fathi Salmi, and Jipeng Huang. Efficient Deterministic Replay of Multithreaded Executions in a Managed Language Virtual Machine. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 90–101, New York, NY, USA, 2015. ACM.

[7] George Candea, Neoklis Polyzotis, and Radek Vingralek. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *Proceedings of the VLDB Endowment*, 2(1):277–288, aug 2009.

[8] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1—-4:26, jun 2008.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[11] TPC Council. TPC Benchmark H (Decision Support) Standard Specification Revision 2.14.4: http://www.tpc.org, 2012.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. *Proc. VLDB Endow.*, 7(12):1295–1306, aug 2014.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[16] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, SIGMOD '06, page 325, New York, New York, USA, 2006. ACM Press.

[17] Tim Harris. Hardware Trends: Challenges and Opportunities in Distributed Computing. *SIGACT News*, 46(2):89–95, jun 2015.

[18] Bingsheng He, Weibin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce framework on graphics processors. *International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[19] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, SIGMOD '08, page 511, New York, New York, USA, 2008. ACM Press.

[20] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.

[21] Johan Henriksson. Java Bindings for OpenCL: http://www.jocl.org, 2015.

[22] Feng Ji, Heshan Lin, and Xiaosong Ma. RSVM: A Region-based Software Virtual Memory for GPU. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Piscataway, NJ, USA, 2013. IEEE Press.

[23] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware - DaMoN '12*, DaMoN '12, pages 55–62, New York, New York, USA, 2012. ACM Press.

[24] Tomas Karnagel, Rene Mueller, and Guy M Lohman. Optimizing GPU-accelerated Group-By and Aggregation. In *Sixth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 13—-24, Kohala Coast, Hawaii, 2015.

[25] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 130–141, New York, NY, USA, 2002. ACM.

[26] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1—-7:32, may 2008.

[27] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[28] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium*

on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75—-, Washington, DC, USA, 2004. IEEE Computer Society.

[29] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, sep 2010.

[30] Aaftab Munshi and Others. The OpenCL Specification. *Khronos OpenCL Working Group*, 1, 2009.

[31] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[32] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. Performance Evaluation and Benchmarking. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, chapter The Star S, pages 237—-252. Springer-Verlag, Berlin, Heidelberg, 2009.

[33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[34] Sreepathi Pai, Matthew J Thazhuthaveetil, and R Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. ACM.

[35] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, SOSP '13, pages 49–68, New York, New York, USA, 2013. ACM Press.

[36] Kenneth Russell and David Detlefs. Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272, New York, NY, USA, 2006. ACM.

[37] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[38] Rajkumar Sen, Jack Chen, and Nika Jimsheleishvilli. Query Optimization Time: The New Bottleneck in Real-time Analytics. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, IMDM '15, pages 8:1—-8:6, New York, NY, USA, 2015. ACM.

[39] Rifat Shahriyar, Stephen M Blackburn, and Kathryn S McKinley. Fast Conservative Garbage Collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 121–139, New York, NY, USA, 2014. ACM.

[40] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.*, 6(11):1068–1079, aug 2013.

[41] Danny Sullivan. Google Search Statistics: http://www.internetlivestats.com/google-search-statistics, 2012.

[42] LINQ Team. LINQ (Language-Integrated Query): http://msdn.microsoft.com/en-us/netframework/aa904594.aspx.

[43] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, aug 2009.

[44] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. GDM: Device Memory Management for Gpgpu Computing. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 533–545, New York, NY, USA, 2014. ACM.

[45] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proceedings of the VLDB Endowment*, 7(11):1011–1022, jul 2014.

[46] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[47] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.

[48] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data Mining with Big Data. *IEEE Trans. on Knowl. and Data Eng.*, 26(1):97–107, jan 2014.

[49] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, aug 2013.

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

[51] Paul C. Zikopoulos, Dirk DeRoos, Krishnan Parasuraman, Thomas Deutsch, David Corrigan, and James Giles. *Harness the Power of Big Data*. McGraw Hill Professional, 2012.