What can go wrong with in-memory computation frameworks

Ravi Tandon, Haoyu Zhang
Guide: Prof. Kai Li
COS 598D
Analytics and Systems of Big Data

Abstract

Whislt the advent of in-memory computation platforms for big-data applications has considerably improved application throughput, it has significantly increased reliance on the available memory resources within individual computing nodes. The current design of large scale data computation platforms makes use of commodity servers with few resources on individual nodes. We present an analysis of the impact of this tension that could possibly hamper future large scale computing platforms. We, therefore, present an analysis of the effects of memory pressure on big data applications running on the Spark runtime. Specifically, this work quantifies the impact of garbage collection done by storage management on the application throughput. Besides, we present results based on object level access patterns of an application running on Spark through a unique interception mechanism oblivious to applications. Additionally, we extend the abstraction of RDDs by providing indexing capabilities within key-valued RDDs. We show that simple range partitioning of real world log data can help reduce query time when using RDDs.

1 Introduction

With the advent of big data systems such as [3, 6, 10] large computations have been pushed within memory of the computing nodes. While reduction in the dependence on secondary storage for fault-tolerance can improve the performance throughput significantly, it increases the memory pressure on the application. Our belief is that as systems scale, memory would become a bottleneck for applications that rely heavily on memory. We, therefore, investigate the effect of memory pressure on a state of the art runtime engine (*Spark*) and point out the fundamental issues in extending current systems owing to specific access patterns of these applications. Additionally, we suggest extensions in the design of RDDs that can significantly reduce in-memory computation, thus lending to better scalability.

Spark is an in-memory runtime that supports large big data workloads. Spark introduces the concept of *Resilient Distributed Datasets (RDDs)* which are large data sets and can be partitioned across several nodes. RDDs use a global namespace and therefore are globally visible from every node within the cluster. Internally, Spark, models computations as graphs of tasks much like *Dryad* [7] and computes lineages based these computation graph models for resilience. Inherently, Spark pushes computations and the corresponding data in memory, unlike the MapReduce [5] framework that relies on intermediate persistence for fault tolerance. We posit that such frameworks will be bottlenecked by the available DRAMs as the applications scale. We identify two basic reasons for this. Firstly, commodity servers typically run with 4GB or 8GB of RAM and therefore processing data on the order of hundreds of gigabytes of data can require tens of server machines. This could not only increase the cost of the cluster, it would result in heavy overheads due to excessive network and disk bandwidth utilization. Scaling DRAM is not a viable option since the cost of DRAM ($/GB) goes exponentially higher as DRAM sizes increase beyond 64GB. [4] Secondly, runtimes such as Spark are built on managed runtimes such as Scala. Scala runs on top of Java Virtual Machine (JVM). JVM increases the overall overhead and results in significant reduction in application throughput due to memory management overheads. [9]

We therefore investigate the performance of a big data application over 5 different configurations and show that applications that use in-memory computation frameworks would perform better if memory per node can scale well. Horizontal scaling supports larger workloads at the cost of a reduction in the application throughput due to increased communication costs. In order to develop deeper insight into possible solutions to scale applications we have designed a unique interception mechanism within the Java Virtual Machine that profiles an applicationâĂŹs access patterns at the object level. We observe that applications have a heavy tailed access pattern which can be exploited to extend memory management systems to better scale applications by transparently and dynamically profiling the workload. In order to reduce computation and communication costs, we propose generic extensions to RDDs by range partitioning them. We refer to these RDDs as IRDDs *(Indexed RDDs)*. We find such range partitioning

schemes to be useful for running filtering queries on log based datasets.

The rest of the paper is organized as follows. Section 2 provides higher level ideas that motivated this work. Section 3 describes the design of our work. Section 4 discusses implementation details of I-RDDs. Section **??** describes some of the related work. Section **??** describes our evaluation. Section 7 provides a brief summary of our work.

2 Motivation

With the advent of Spark like Big data runtimes big data, workloads will create excessive pressure on the current memory subsystems. We believe that current memory subsystems are not well suited to handle spikes of memory requirement and will get severely affected by non-uniform memory requirement patterns. Fig 1 shows the overall time it takes for the Garbage Collector for different tasks when running a page ranking application. The initial tasks require a large amount of memory. We believe this could be because of the extra memory required when reading data from the secondary storage (due to deserialization of data). There is a subsequent drop in the throughput for these specific set of tasks. This was the primary motivation for studying the effect of memory pressure on different configurations. Besides, we believe, the answer to better memory management lies in understanding the underlying access patterns of different workloads and offloading low priority data on hybrid memory subsystems. Therefore, we performed experiments quantifying access patterns in baseline spark system.

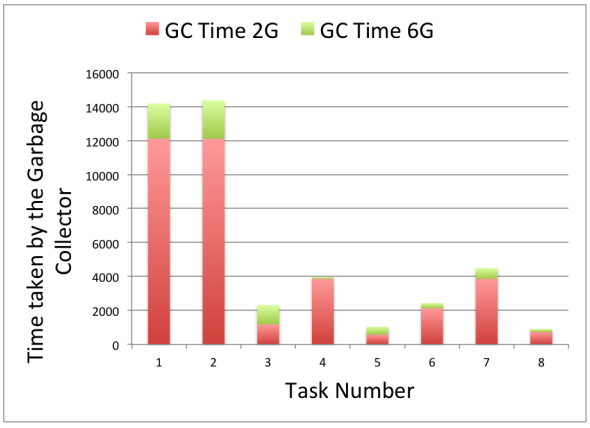

Figure 1: Overall time spent in garbage collection for tasks with 2GB and 6GB of memory per node

Another, observation we had is that the abstraction of RDDs is still inept when handling search queries on semi-structured datasets such as logs from applications, page views etc. When filtering a dataset Spark has to parse all the partitions within a dataset, while the actual results might be contained in a subset of the overall partitions. Fig 2 shows the comparison between Vanilla Spark's filtering query and an ideal implementation of a set of queries on a dump from Wikimedia [2] where we observe that on an average only 48% of the total partitions contained relevant results. This was the primary motivation behind building a range partitioning mechanism on top of partitions and thereby indexing the data stored in RDDs.

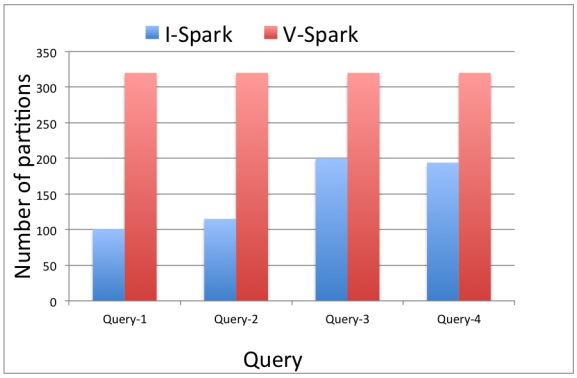

Figure 2: Comparison of query runtimes on Vanilla (V-Spark) and an ideal query engine (I-Spark)

3 Design

We first describe the basic design Spark. Spark abstracts data in the form of a globally addressable data structure called Resilient Distributed Dataset. The RDD is divided into smaller partitions. Each partition is an an array of the unit (the unit is class type of the data). In order to execute a job on spark, a driver program requests the master node to schedule a given task on the worker nodes. The job (in our case a filter query) is then executed in four basic steps:

1. The master node requests file metadata from the underlying hadoop file system. A lazy read on the file is performed, meaning the data is also read into the memory when an action has to be performed on the RDD. Based on the file metadata, the master node computes the appropriate partitions on the file size. (Spark uses a maximum size of *33MB* per partition).

2. The master then sends the jar file containing the job to the different worker nodes along with the set of partitions on which each of the worker node has to perform the job.

3. The worker nodes read the data from the distributed file system into their local memory. Thereafter, they execute the actions on the RDDs and return the result back to the master node.

4. The master node then performs necessary completion actions on the job (generally an aggregation of the results) and returns the result back to the driver.

Index Creation: In our design, driver program initially requests the creation of an index on the RDD given a specific key. The Spark runtime sends a job (the index creation) job on each of the worker node. Each of the worker node then reads the allocated partition and computes a pair of keys. This pair denotes the smallest and the largest key within the partition and essentially is the range of the partition. The master then retrieves the range for each partition and then aggregates them into a single in-memory array mapped to the file, key pair. Any subsequent query on the specified key is performed through a different sequence of steps. For any lookup query on the key, the master node computes the partitions which include the key within their range. The master node then runs the job on the filtered set of RDDs which may contain the key. The job is then sent to the different worker nodes. Each of the worker node performs the job on their respective of partitions and returns the result back to the master. The master then aggregates the result and sends it back to the driver.

Design of Indexed RDDs: Here we discuss the design of our indexed RDDs. We support indexing on top for RDDs wherein each record can be structured as a key-value pair. We range-partition the RDDs. For each partition the highest and lowest key (according to an appropriate class comparison function) is computed. This is thereafter stored as the metadata for each partition. The range partitions are thereafter used for any subsequent queries to compute the possible subset of partitions within which the key can exist and the corresponding query is run on the RDD. In our current design, the range partitioning is done by the master and the range partition is cached in memory as a map of partition index as the key and range as the corresponding value. Besides, support for range partitioning, we are also support a faster indexing scheme if the underlying data is sorted on the key. The indexer stores the smallest value for each partition and caches it in memory. Since, the keys are sorted, any subsequent pair of keys can be used as a range for the preceding partition. Thereafter, the master can efficiently figure out the range of partitions within which a querying key would lie in.

Design of JVM prototype: For our virtual machine prototype, we use an open source Java Virtual Machine [1], *Oracle's OpenJDK framework*. A Java program is compiled into an intermediate bytecode representation. Object accesses are compiled into load and store instructions. We intercept the load and store instructions and put in extra instructions to that increment the count of an object on each access. The Java Virtual Machine abstracts each java object as class instance within the virtual machine. In order to achieve a low overhead on each object access we add an extra field within the object header and increment the count on every object access. This lets us monitor accesses on each object throughout the java program's execution.

4 Implementation

We here describe the implementation of our indexed RDDs. We create a separate derived class IndexedRDDKV, with the RDD class as its base class. We support the following interfaces on the RDD class:

1. *indexedKV():* The driver program can call indexedKV() on any RDD to transform the given RDD into an RDD of key, values which supports indexing. Each record must be a key-value pair. We expose the following interfaces on the IndexedRDDKV class:

2. *rangePartitions ():* The method creates a range partitioned index on the data in the RDDs. For each partition, the smallest and the largest value is stored as the range of the partition.

3. *searchByKeyRangePartitioned (key: String):* This method searches for the key within the range partitions. The master first filters the set of partitions within which the key can exist and then runs the query on those select partitions.

4. *indexPartitions():* The indexPartitions method is used to create a partition on the range of keys when the input data is already sorted.

5. *searchByKey(key: String):* The search by key function searches the given string within the RDD using the index created by the indexedPartitions function.

5 Evaluation

5.1 Experiment 1: Study of the effect of garbage collection on Spark runtime

Our first set of experiments were conducted to understand the effect of memory pressure on applications by quantifying the impact of garbage collection on a page ranking application written in Spark. The higher level questions that we wanted to answer are as follows:

1. What is a good strategy of scaling Big data applications ? (more cores, lesser memory per core versus lesser cores, more memory per core)

2. What is the impact of managed runtimes on application throughput ?

3. Is there a more fundamental problem with the way memory is managed by runtime engines ?

Experimental Setup: The experiments were conducted on machines with Intel(R) Xeon(R) running at clock frequency of 2.66GHz, with 4 processor cores, with 16 GB of DRAM size. Each machine runs Linux version 2.6.18 with L1 cache size of 32K.

Experiment 1: We conducted the experiments on 5 different configurations. The configurations were as follows:

- Configuration 1(15W, 1G): The configuration consists of 15 worker nodes, each with 1 GB of memory.
- Configuration 2(10W, 1G): The configuration consists of 10 worker nodes, each with 1 GB of memory.
- Configuration 3(5W, 2G): The configuration consists of 5 worker nodes, each with 2 GB of memory.
- Configuration 4(2W, 5G): The configuration consists of 2 worker nodes, each with 5 GB of memory.
- Configuration 5(1W, 7.5G):The configuration consists of 1 worker node, each with 7.5 GB of memory.

Table 1 lists out the running times, along with the overheads due to the garbage collector for different configurations.

Configuration	Application Run Time (in ms)	% Overhead due to GC
15W, 1G	428,245	33%
10W, 1G	414,742	30.5%
5W, 2G	216,320	23%
2W, 5G	152159	15.29%
1W, 7.5G	164893	8.45%

Table 1: Comparison of the overhead in throughput due to garbage collection for different configurations

The application we ran was Vanilla Page Ranking algorithm provided with the Spark library. The dataset that we took describes a network which was collected by crawling Amazon website. It is based on *Customers Who Bought This Item Also Bought* feature of the Amazon website. If a product i is frequently co-purchased with product j, the graph contains a directed edge from i to j [**?**]. It describes a graph of *403394 nodes* with *3387388 nodes*. The data is around *50MB* in size. The page ranking algorithm was run for *10 iterations*.

Analysis of results: Fig. 3 shows a relative comparison of the overall throughput from the cluster under different configuration settings. Fig. 4 shows a relative comparison of the impact of garbage collection on the cluster under different configuration settings. We can observe that garbage collection has a detrimental effect on the overall running time of the application. For a configuration of 2 nodes with 5 GB of memory per server, the overall running time of the application increased by 16%, whereas for a setup of 15 workers with 1 GB of memory on each server the overall running time of the application was reduced by 33%.

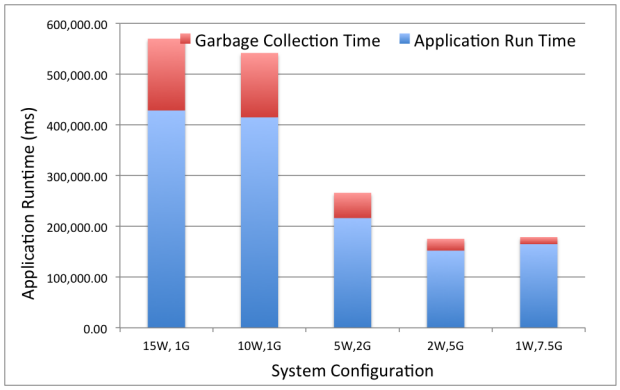

Figure 3: Comparison of application throughput for different configurations

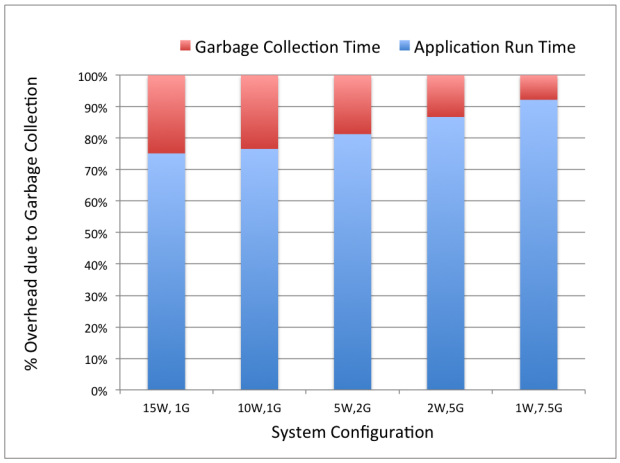

Figure 4: Comparison of relative affect of garbage collection on application runtime for different configurations

There are several reasons that explain the above behavior. Firstly, spark is an in-memory runtime application. It requires extensive amounts of memory for storing intermediate data such as when de-serializing data read from the storage layer, maintaining cache of RDDs, lineages of RDDs etc. Larger memory requirements creates extra memory pressure triggering frequent garbage collection. Therefore the overall throughput of the application decreases. Secondly, in order to scale the application, when the number of nodes in the cluster is increased, the synchronization costs due to communication over the network and secondary storage causes additional overhead.

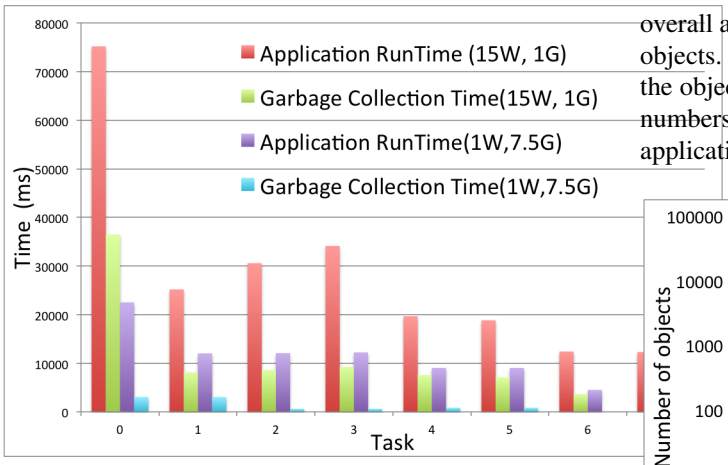

Figure 5: Comparison of application runtime for successive tasks for two different configurations

One of the intuitive conclusions one can derive from the above set of experiments is that in order to scale memory-intensive applications, horizontal scaling may not be as effective as scaling resources on individual nodes.

5.2 Experiment 2: Study of object access level patterns within Spark

In order to answer the second question, we took a dummy application in Spark that calculates the value of Pi on a local machine.

Objective: The objective of the experiment was to measure the spectrum of object level accesses when running a baseline spark application that does minimal work.

Experimental Setup: The experiments were conducted on Intel(R) Xeon(R) CPU E3-1230 V2 machines running at a clock frequency of 3.30GHz with 8 cores with L1 cache size of 32K. For the experiment we used 512MB of memory.

Analysis of results: The results we report are for a specific period of time for which the application runs, since our interception mechanism 3 can approximate object accesses within a single evacuation [**?**] cycle of G1 garbage collection. Fig. **??** shows a histogram based on the object level accesses of different component objects within the spark application. The results indicate a very heavy tailed object level access pattern. Within the specified period, around 45,000 objects were live within the young generation of the application. Out of these, around 1000 objects (nearly 2%) had an access count of over 1000 per object. We label these objects as hot objects. They account for nearly 58% of the overall accesses for the given period within the survivor and eden regions. Nearly, 17% of the overall accesses derive from 0.02% of the total number of objects. Nearly, 3% of the overall accesses are for 50% of the objects in the specified regions of the memory. These numbers clearly indicate that object access patterns within applications are extremely heavy tailed.

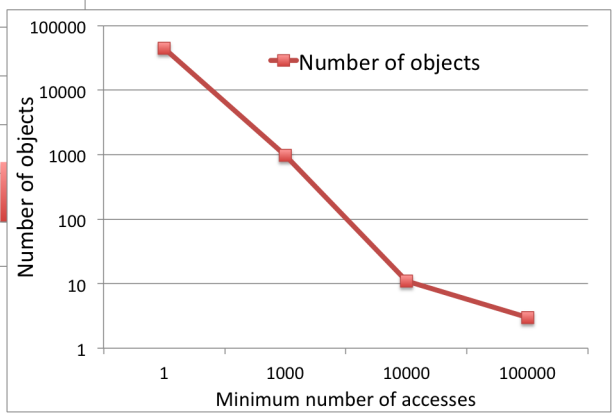

Figure 6: Histogram of object accesses for different objects for a Spark Application

Most of the data objects that were accessed more frequently were arrays of utility functions such as worker queue, XML document handlers, symbol table entry handlers, loggers etc. The objects that were less frequently accessed were temporary objects such as method objects, strings, classes for caching root names, object for looking up the mime types (used for inspecting header of objects), objects created for storing symbol tables for compilers, parsers etc. There are three reasons explaining the heavy tail in the specific distribution pattern of object level accesses. Firstly, accesses to large arrays of objects (such as RDDs) will have specifically high values since each access of an object within an array is also an access to the array (understood as a container of objects). Such arrays can be considered hotspots within the object space. Secondly, instances of objects that contain utility functions such as compression of data, methods for polling the network interfaces, DOM parsers etc. have large access counts. These objects are accessed more ubiquitously due to their generic functions and therefore are hot in their access patterns. Thirdly, a very large collection of objects (nearly half of all the objects) such as classes for caching root names might get accessed once and be used only at certain specific intervals, such as that at a task start up for looking up the addresses of the different workers. These "cold" objects show low usage access pattern.

The intuition that one can develop from this experiment that access patterns within memory intensive workloads (such as those used by Spark) have extremely long tails. Such access patterns unearth a potential problem within

contemporary memory management system designs. Current virtual memory systems have a paged based abstraction for managing application data. Our intuition is that data residing on similar pages might not always have very good "access similarity". Paging, therefore, by virtue of its underlying design cannot capture the access level semantics of applications at finer granularity. This experiment provides us with useful insights and helps further our understanding of the underlying fundamental problems within current memory management system designs.

5.3 Study of indexing within RDDs:

Objective: We observe that while the design of RDDs supports caching and faster recovery through lineages, it lacks certain capabilities. Besides, RDDs inherently support operations on data sets which consist of pairs of key and values. We posit a natural extension to the design of RDDs using range partitioned indexes on keys. This experiment answers the following three questions:

1. Is there more efficient way of running queries on semi-structured data stored in files (database logs, page visit logs) ?
2. What are better schemes of indexing larger datasets using RDDs ?
3. How well can a generic indexing scheme based on range partitioning scheme scale ?

Experimental Setup: The experiments were conducted on machines with Intel(R) Xeon(R) CPU E3-1230 V2 running at a clock frequency of 3.30GHz, with 8 processor cores, with 32 GB of DRAM size. Each machine runs Linux version 3.11.10 with L1 cache size of 32K. The experiments were run on a cluster of 3 nodes connected with 1 Gb/s NICs arranged on a single rack. The Spark setup consisted of 6 workers (2 on each machine) and 1 master. Each of the workers was configured with 8GB of memory and the master had 2 GB of memory. The master was set up on the same machine as one of the workers. The tests were run using using the Spark shell interface. The experiments were conducted on Hadoop file system. The version of Hadoop was 2.2.0. The block size was set to default 256MB. The Hadoop master and data node were set up on the same machine.

Benchmarking DataSet: For the experiments we used page view logs from Wikimedia dumps [2]. Wikipedia publishes page view statistics every month. The following is an example of the data: *fr.b Special:Recherche/Achille_Baraguey_d%5C%27Hilliers 1 624*. In the above example, the first column ("fr.b") denotes the project name. The second column is the title of the page retrieved, the third column is the number of requests, and the fourth column is the size of the content returned. In our experiments, we use the project name as the key and the rest of the line as the value that needs to be retrieved from the underlying distributed file system. We use two different collection of data (10 GB, 25 GB) of data, approximating 5 months and 1 years of data respectively, with nearly 300 million and 730 million records.

Experiment 3: For the experiments we run five different queries and measure the overall time it takes for the queries to complete. Fig. 7 shows a comparison of three different configurations when running the queries on 10GB file. VSpark denotes queries on vanilla spark using unindexed RDDs, ISpark_C (Indexed Spark Coarse Grained Partitioned) denotes RDDs with 320 partitions and ISpark_F (Indexed Spark Fine Grained) denotes RDDs with 1280 partitions. While VSpark takes 87 seconds to complete , the average time for a positive query (a query that matched keys in the file) was around 37 seconds. Using, more fine grained partitioning, the average time to 11 seconds. Another interesting observation to note is that for queries that miss, the result is returned in under a second. The underlying reason for the improvement in performance is that the range partitioning mechanism can pre-determine the set of RDDs on which the query needs to be run. This, reduces the overall computation that needs to done for filtering records. Besides, the communication cost over the network is reduced. This results in good enhancement in the performance. For queries, that result in a miss, the master can instantaneously return a null result, since it can ensure that key is outside the range of all the partitions.

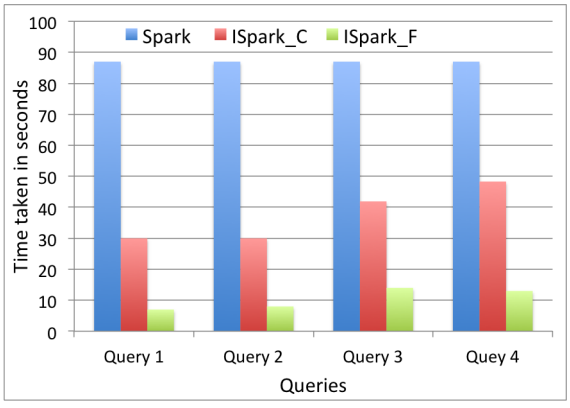

Figure 7: Comparison of query completion times between Spark, Spark-Coarse grained and Spark-Fine grained partitioned for a 10 GB file

Fig. 8 shows a comparison of three different configurations when running the queries on 25 GB file. The number of partitions for ISpark_C were 1572 and that for

ISpark_F were 10000. While VSpark takes 220 seconds to complete , the average time for a positive query (a query that matched keys in the file) was around 49 seconds. Using, more fine grained partitioning, the average time to 9 seconds. We can observe, that while VSpark scales poorly as the size of the data increases, indexes helps the filtering query to scale in a more scalable manner. With more fine grained partitioning, we were able to reduce the overall query time for a 25GB over 10GB. However, we cannot increase the number of partitions since the master node would have limited amount of memory. In order to scale up, we can maintain a separate indexing server that can cache indexes and return the set of RDDs on which a query has to be run.

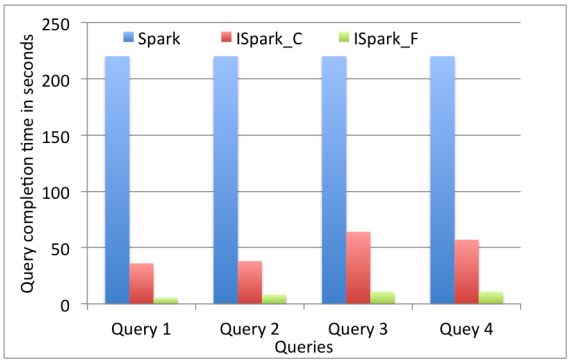

Figure 8: Comparison of query completion times between Spark, Spark-Coarse grained and Spark-Fine grained partitioned for a 25 GB file

Fig. 9 shows the overall time it takes for the indexing mechanism to set up the partitions on the range of RDDs. The partitioning scheming scales in accordance with the filtering mechanism since the all the set of records have to be filtered. However, since, this is a one time cost, the cost can be amortized with a batch of queries. Besides, as the number of partitions increases, the time it takes to build an index does not increase by a large amount. Therefore, fine grained indexing can be very useful as long the master has sufficient memory to hold the index.

6 Related Work

We build our design on top of the Spark[10] runtime engine. While, Spark provides useful abstractions for performing distributed computations on top of RDDs, it lacks good support for performing queries (such as filter, count etc. on pre-specified keys). Other frameworks such as Shark [6], Blink-DB [3] have been used for performing queries on top of Spark. However, they require data to be completely structured as data within the mysql database. We have designed our system for semi-structured data, which consists of key, value pairs in files.

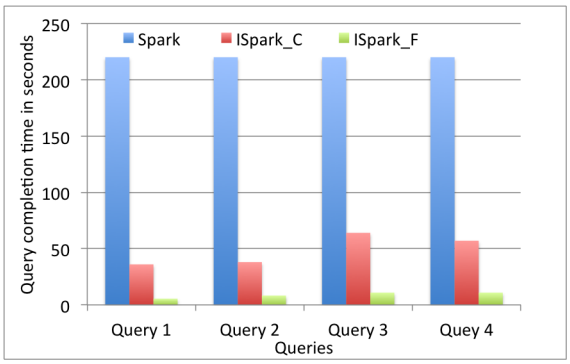

Figure 9: Comparison of index creation time

Systems such as MANIMAL [8] perform automatic optimizations on top of MapReduce programs. They can provide similar optimizations by building B+ trees on data already stored within the HDFS. However, our design works for an in-memory runtime engine, though fundamentally it is not very different.

Systems such as SSDAlloc[4] have looked at detecting object level accesses within native language programs. The system relies on page protection mechanism which can be expensive. Our system intercepts object level accesses within a managed runtime and is completely transparent and therefore addresses a different problem subspace.

7 Conclusion

We have presented an evaluation of the impact of memory pressure on the large in-memory computations. By quantifying, the impact of decrement in throughput due to garbage collection, we provide useful insights into managing cluster level applications. We posit that scaling horizontally by deploying more nodes may not scale applications as well as supporting individual nodes with larger memory. We have also performed experiments and measured the spectrum of object usage for a regular Spark application. We find that the object access patterns are heavy tailed and can be utilized to manage memory better. Lastly, we explore further improvements in the RDDs, by building indexes on top of partitions and show that we can perform more efficient range queries with such a design.

References

[1] URL http://openjdk.java.net/.

[2] URL http://dumps.wikimedia.org/.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[4] A. Badam and V. S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 16–16. USENIX Association, 2011.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 689–692. ACM, 2012.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[8] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[9] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116. USENIX Association, 2006.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.