Is Memory Disaggregation Feasible? A Case Study with Spark SQL

Pramod Subba Rao and George Porter
UC San Diego

Abstract

This paper explores the feasibility of entirely disaggregated memory from compute and storage for a particular, widely deployed workload, Spark SQL [9] analytics queries. We measure the empirical rate at which records are processed and calculate the effective memory bandwidth utilized based on the sizes of the columns accessed in the query. Our findings contradict conventional wisdom: not only is memory disaggregation possible under this workload, but achievable with already available, commercial network technology. Beyond this finding, we also recommend changes that can be made to Spark SQL to improve its ability to support memory disaggregation.

1. INTRODUCTION

Achieving efficiency in data processing requires balanced computing, meaning that a system has the right mix of CPU, memory, storage IO, and network IO so that one part of the computation is not bottlenecked waiting for results from another part of the computation. Getting this balance just right is a moving target, since the input data, number and type of queries, presence of failures, and network conditions are in a constant state of flux. Correctly provisioning bare-metal servers is a challenge since one must determine their specific configuration at entirely the wrong timescales, well before they are put into production. While virtual machines play an important role in providing more flexibility in balancing resources, they are not enough. Even with VMs, you are limited to configurations implementable in a single server, you are subject to "fragmentation" of resources, and you are not able to upgrade individual components like CPU and memory independently of each other.

These challenges have led to disaggregated server designs, where individual components such as CPU, memory, and storage are interconnected over a network, rather than over a bus within a single chassis [12]. The advantages of disaggregation include more efficient utilization of resources, and the ability to independently upgrade different system components. The challenge for disaggregation is the "memory wall" [21]. Today storage is commonly disaggregated via SANs and other network-based file storage protocols, and Facebook has introduced a disaggregated system-on-chip (SoC) platform called Yosemite [5], which relies on networked storage. Yet there is a growing gap in the rate at which CPUs can execute instructions and the rate at which data can be fetched into the CPU from main memory. For this reason in Yosemite (as well as other designs), memory and CPU are still tightly integrated in the same chassis.

This paper puts aside the issue of disaggregating memory in general, and instead examines disaggregating memory for a common and increasingly deployed type of application: analytics queries. Using Spark SQL [9] as a motivating platform, we measure the actual rate at which threads of execution access memory and process records, and using these measurements, determine the feasibility of disaggregating memory. Spark is an example of a growing set of data-parallel frameworks which exhibit minimal data-dependent branches, and as such, can take advantage of significant amounts of pipelining. For this reason, they are largely latency insensitive, further enabling the use of disaggregated memory.

Our initial results show that even after significant optimization, Spark SQL analytics queries access memory an order of magnitude slower than the underlying components permit, opening up the possibility of disaggregating memory from compute. In fact, the requirements on the underlying network are modest, and can be met with existing commercial products such as 40- and 100-Gb/s NICs (e.g., the Mellanox ConnectX-4 NIC [17]). We conclude by recommending further changes that improve Spark SQL's ability to support memory disaggregation.

2. MOTIVATION

Two major reasons that server disaggregation attempts have avoided memory is that, at a component level, (1) the bandwidth required between memory and the CPU is too large to be supported by commercial network equipment, and (2) network latency is too high. In the first case, we demonstrate experimentally using microbenchmarks that memory bandwidth indeed exceeds network capabilities (in Section 2.1). Yet users do not run microbenchmarks, they run applications, which might not have such stringent requirements, including for memory latency. We explore one such application in Section 2.2, chosen because of its highly efficient use of memory, serving as a compelling motivating application.

ANCS '16, March 17 - 18, 2016, Santa Clara, CA, USA

ISBN 978-1-4503-4183-7/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2881025.2881030

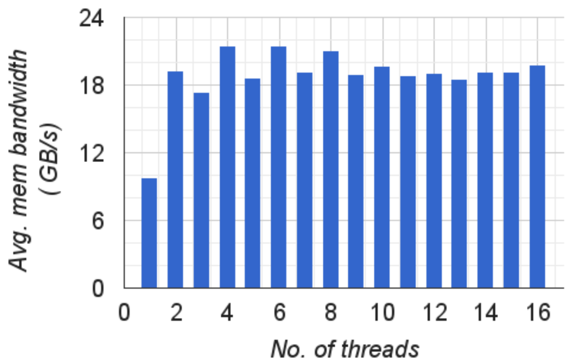

Figure 1: Aggregate memory bandwidth of STREAM benchmark

2.1 The memory wall: barrier or paper tiger?

We start by examining an upper-bound on the bandwidth requirements of a memory disaggregated system through the STREAM benchmark [16], which is a synthetic benchmark that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. A wrapper tool called stream-scaling [7] automates the process of executing STREAM over the various core counts is used in this study. We have deployed STREAM onto an 8-core Intel Xeon 2.27 GHz E5520 processor-based HP ProLiant DL380 G6 server. This machine has 24GB of DDR3 Synchronous RAM (1333 MHz), configured as 12x2GB banks in the Advanced Error Correction Code mode.

Figure 1 shows the results of the STREAM Copy benchmark on the experiment hardware, which copies data from one array of doubles to another. The total memory bandwidth of the system with all cores active is approximately 21 GB/sec (or 168 Gb/s), a result that matches the HP ProLiant datasheet [2]. Not only does 168 Gb/s exceed any commercially available network interface card (NIC), it exceeds the PCIe 3.0 bandwidth capacity (a x16 device is limited to 15.75 GB/s, of which approximately 14.2 GB/s are usable), meaning that modern servers are simply unable to keep up with such demand. Thus, absent additional constraints, memory disaggregation is not feasible. Lim et al. [14, 15] explore partial memory disaggregation, where memory is partitioned into local and remote blades. In this paper, we examine entirely remote disaggregated memory constrained to a specific, though popular and widely deployed, application.

2.2 Analytic queries with Spark SQL

Spark SQL [9] is a relational data processing system implemented in Scala and built on top of the functional programming API of Apache Spark [22]. It bridges the gap between traditional analytics queries and machine learning algorithms by offering both an SQL interface and a procedural programmatic interface.

Analytics queries are used to generate summary reports from large amounts of raw data in order to glean insights. They are characterized by: (1) accessing the source data in a read-only manner, (2) accessing a large number of rows from a table (frequently the entire table), but only for a small subset of all the available columns in the table, (3) performing aggregation operations (count, sum, average, group by) on one or more columns or tables, and (4) consisting of CPU-intensive operations for advanced analytics and machine learning algorithms. To this last point, Ousterhout et al. [18] have shown that for a large set of realistic workloads, Spark is CPU-bound, not memory, network, or storage bound. Due to the nature of aggregation-based queries, the size of the working set can decrease, in some cases significantly, after every stage of aggregation.

In this paper, we chose Spark SQL due to its highly-efficient use of memory, due in part to three major factors: (1) it stores data in a column oriented format, making it efficient to access all the rows of a column, (2) it generates Java bytecode directly for commonly used aggregation operations like *count*, *min*, and *max*, thus avoiding the overhead of multiple, and possibly virtual, function calls, and (3) it applies operators to entire data sets (called RDDs) in parallel, without data-dependent branching. This lack of branching reduces the impact of higher memory latency on overall application throughput. We deploy a series of analytic queries taken from the literature and published benchmarks, and for each thread of execution, we measure the rate at which records are processed and calculate the effective memory bandwidth based on the sizes of the columns accessed in the query multiplied by the number of threads. This results in the overall aggregate memory bandwidth.

Specifically, if there are *threadcount* threads accessing *size* bytes from each record, during a time interval of $AvgTime_{100000}$ between every 100,000 records (averaged across all the threads), the aggregate memory access rate is:

$$Mem\_access\_rate = \frac{(size \times 100000)}{AvgTime_{100000}} \times threadCount \quad (1)$$

We use this formula to calculate the actual memory access rate, rather than potential access rate, for sets of queries.

2.3 High-speed networking

Today 10- and 40-Gb/s Ethernet is commercially available and widely deployed within production data centers [11]. Commercial 100 Gb/s NICs and switches are now available from vendors such as Mellanox [17], and 400 Gb/s Ethernet is in the standardization process [10]. A key aspect of these new standards is that their high overall speeds are obtained by joining multiple, parallel, underlying links together. For example, 100 Gb/s Ethernet is largely 4 25 Gb/s lanes, and 400 Gb/s is currently 16 25 Gb/s lanes (eventually to be replaced with 4 100 Gb/s lanes when those become available). Simply put, the ability to increase a single lane of Ethernet cannot keep pace with overall bandwidth demands, and so parallelism is used in new standards.

3. EXPERIMENTAL SETUP

3.1 Hardware and software

The hardware used for the following experiments is a cluster of five nodes, consisting of a single master node and four workers. Each server is the same configuration as described above in Section 2.1, and they are interconnected with a 10 Gb/s network. We rely on the cluster to distribute jobs to servers, however our measurements are limited to a single server, and thus the network is not a bottleneck for profiling the bandwidth requirements of the memory system.

We use Apache Spark 1.3.0 [1], deployed in standalone mode on Ubuntu Linux 14.04.2. One executor process is run on each worker node and is allotted 18GB of RAM out of the 24GB. To read CSV files, we use the spark-csv library [6]. The Java Virtual Machine used is Java HotSpot 1.6.0_45-b06. No other software is running on this cluster apart from the default system processes.

3.2 Spark SQL

Apart from allocating ample memory (18 GB) to each node, we have set up Spark in a way that increases the demand on the memory subsystem compared to more general configurations, in an effort to provide a conservative upper-bound on the memory bandwidth requirements. Unless mentioned as follows, we maintain the default Spark configuration. We have modified the following settings:

1. We ensure all memory accesses are to local memory. In production workloads, off-node memory might be accessed as well.
2. *spark.storage.memoryFraction* is increased to 0.8 (default: 0.6). This is the fraction of Java heap to use for Spark's memory cache; increasing this value ensures that resilient distributed datasets (RDDs) are cached entirely in memory.
3. We set *spark.shuffle.spill* to false. This ensures that data does not spill over to disk during the reduce phase.
4. Memory compression is turned off (*spark.sql.inMemoryColumnarStorage.compressed*) to reduce extra overhead on the CPU during query processing.
5. Our instrumentation measures access times after every 100,000 records, and so we set *spark.sql.inMemoryColumnarStorage.batchSize* to 100001 (from its default of 1000) to ensure we have a sufficient number of records in each batch.
6. We turn on dynamic code generation (*spark.sql.codegen*). This optimization within Spark SQL generates Scala code at runtime which is specialized for the types and the number of expressions used in the query. It also avoids autoboxing overhead where primitive types are being used. In the absence of code generation, simple operations like extracting a column from a row, or adding two literals, can result in branching and virtual function calls. The code generation feature generates inline Scala code for the same and compiles them to JVM bytecode before execution.
7. To prevent Spark from writing data to disk during the shuffle phase, we ensure intermediate data is written to memory by setting the partition to a *tmpfs* filesystem.
8. We disable the OS swap partition (via *swapoff -a*).
9. Whenever measurements are needed for a specific number of threads, we achieve that by splitting up the data into an equivalent number of RDD partitions.

3.3 Workloads

We evaluate the feasibility of memory disaggregation using three workloads. The first is the STREAM benchmark, described previously, which measures the raw capacity of the memory subsystem, setting the upper bound on what application could obtain. The second is a microbenchmark of Spark SQL's memory access performance, achieved by measured a simple *COUNT(1)* query, which simply scans a synthetic RDD with rows and columns of different lengths, using a varying number of threads, all while incrementing a counter. This sets an upper-bound on the performance of Spark SQL, as it forms one of the simplest queries possible to express. Third, we evaluate a series of more complex queries drawn from the UC Berkeley AMPLab "Big Data" benchmark [3].

3.4 Measurement technique

We measure memory bandwidth at the application level by instrumenting Spark SQL, and validate these measurements by comparing to CPU-level performance counters.

Spark SQL instrumentation: Spark SQL loads data into an in-memory table accessed via the *InMemoryColumnarTableScan* class. Data for all rows of a column is stored in a Java byte array. The first 4 bytes of the array are used to specify the data type, and the rest contain the actual data. We request that the data for the query be cached in memory through Spark's *rdd.persist()* mechanism. When the query is executed for the first time, a CSV file is read to populate the in-memory table; subsequent executions of the query access only the in-memory representation. We log timing information within *InMemoryColumnarTableScan* during iterations over the table, at intervals of 100,000 records in each of the threads. Equation 1 is used to calculate the access rate to memory. All measurements are taken at one of the worker nodes in the cluster.

CPU performance counters: Intel processors provide a set of counters and associated monitoring software [4] to measure CPU utilization and bytes read/written from memory. We use these during query processing to validate the application-level measurements, and our findings (not shown) match those reported by the Spark-level instrumentation.

4. EXPERIMENTAL RESULTS

We examine the memory demands of analytical queries first by examining a trivial query that provides an upper-bound on the bandwidth that can be achieved by Spark SQL, and then by considering two more complex queries drawn from the AMPLab's Big Data Benchmark [8].

4.1 Microbenchmark queries

Query 1
```
SELECT COUNT(1) from SingleColumnTable;
```

Spark SQL implements Query 1 by fetching the data from the smallest column in the row and then incrementing a counter. If the row contains only one column, this is equivalent to fetching the entire row. We chose Query 1 as an example of a query with minimum CPU usage in order to measure baseline performance of the system. Based on the time taken to count every 100,000 records, the memory access rate is calculated according to Equation 1.

Columns of data, of various lengths, were generated and mapped to different numbers of partitions in order to create

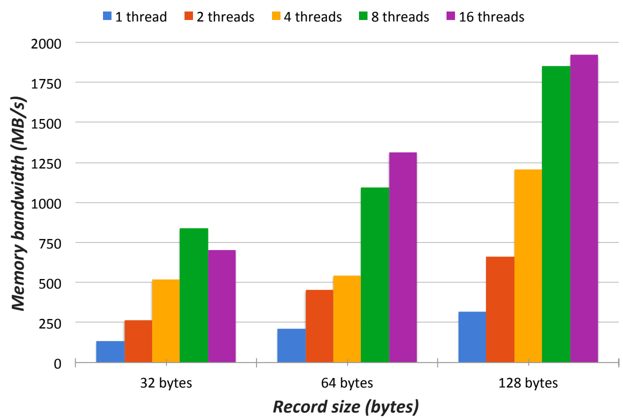

Figure 2: *Select count(1)* **query**

the appropriate number of threads. Figure 2 shows that a maximum average throughput of 1.9 GB/sec is seen when running with 16 threads on 128 bytes of data, all from the in-memory cache. While the throughput increases with the number of threads, it tapers off as it reaches 16 threads. Since the hardware has only 16-cores, running more than 16 threads is not representative of the CPU intensive nature of analytics queries.

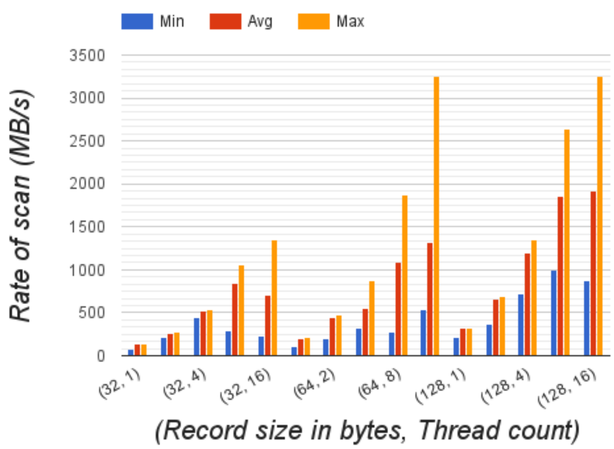

Figure 3: Summary statistics for Query 1

Figure 3 shows the average, max and minimum rates of memory access for Query 1. It can be seen that a maximum throughput of 3.3 GB/sec is observed while scanning records of size 128 bytes in 16 threads. A distribution of this data is shown in Figure 4. The key takeaway from this result is that while under optimal conditions, Spark SQL is capable of driving, in aggregate, an impressive 26.4 Gb/s of network bandwidth on our hardware, it requires 16 independent threads to do so.

4.2 AMPLab Benchmark queries

The previous section has looked at microbenchmark queries, which are significantly memory-intensive, and found that they are satisfiable with 40 or 100 Gb/s Ethernet devices currently available from vendors such as Mellanox [17]. We now turn our attention to more realistic queries, provided

Figure 4: CDF of memory access rates for Query 1 (60M records, 64 bytes, 16 threads)

Column name	Column size	Comments
sourceIp	19 bytes	4 byte length; 15 byte IP
adRevenue	4 bytes	sizeof(FLOAT)
Total	23 bytes	

Table 1: Data accessed per row for Query 2

by the AMPLab "Big Data" benchmark suite [8].

4.2.1 "Group by" query

Query 2

```
SELECT sourceIp, SUM(adRevenue) FROM
 uservisits GROUP BY sourceIp
```

We next look at an Aggregation Query from the Big Data Benchmark [3], shown as Query 2. It shows the advertisement revenue obtained from each end user IP address based on all the sites visited by that IP address and grouping the total revenue obtained from each of those addresses. The *uservisits* table has 10 million entries. For each row, the following columns are accessed: sourceIp and adRevenue. Table 1 shows that the data accessed per row of this query is 23 bytes.

Spark SQL creates 8 threads to process this data on each node. The data accessed per row is 23 bytes, and the average time interval between every 100,000 records is 63.7 ms, and the minimum is 51.0 ms. Based on Equation 1, this translates to an access rate of 289 MB/s (2.3 Gb/s), and a maximum rate of 361 MB/s (2.9 Gb/s). Per thread, however, the demands are a relatively paltry 289 Mb/s on average, 361 Mb/s max.

Effect of code generation: Since Query 2 is more resource intensive (in terms of both CPU and memory) than Query 1, it is instructive to look at the performance of the query without bytecode generation. Figure 5 shows the time spent in various phases of Query 2 in the absence of code generation. Due to the creation of a large number of temporary helper objects for aggregation, and the accompanying garbage collections, the time spent for iterating over data is larger than needed, showing that to get high memory utilization, it is essential to run with code generation enabled. Even with this optimization, it is still practical to support this result with existing network technology.

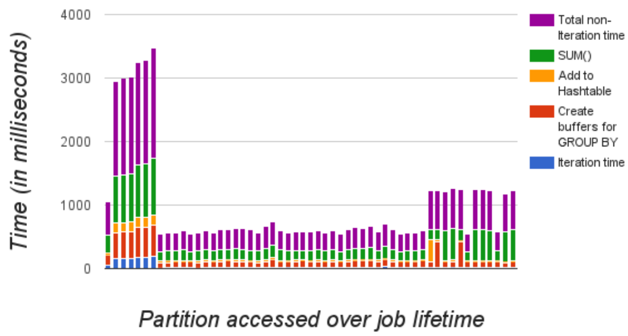

Figure 5: Code generation optimizes memory access rates for Query 2

Column name	Column size	Comments
url	59 bytes	4 byte length; 55 bytes data
pageRank	4 bytes	sizeof(FLOAT)
Total	63 bytes	

Table 2: Data accessed per row for Query 3 (Rankings table)

4.2.2 *"Join" query*

Query 3
```
SELECT sourceIp, AVG(pageRank) as avgPageRank,
  SUM(adRevenue) as totalRevenue
          FROM
              Rankings AS R, Uservisits AS UV
         WHERE
             R.pageUrl = UV.destinationUrl AND
            UV.visitDate > '1980-01-01' AND
            UV.visitDate < '2015-05-20'
         GROUP BY
             UV.sourceIp
        ORDER BY totalRevenue DESC LIMIT 1
```

We next examine a *join* query taken from the Big Data Benchmark [3], shown here as Query 3. Along with the advertisement revenue obtained from each end user IP address, it also displays the average rank of the pages visited by that IP address, by joining with a Rankings table. The Uservisits table has 10 million rows as earlier, and the Rankings query also has 10 million rows. To analyze this query it is necessary to inspect the in memory representation more closely (shown in Tables 2 and 3). Since the data for Rankings is split into 2 partitions on the given node and the data for Uservisits is split into 7 partitions, the number of threads operating upon the two tables are also respectively 2 and 7.

For the *Rankings* table, the data accessed per row is 63 bytes, and the time interval between every 100,000 records was an average of 131.6 ms, and a minimum of 67 ms. Based on Equation 1, this translates to an average access rate of 45.1 MB/s (0.4 Gb/s) and a maximum access rate of 94.0 MB/s (0.8 Gb/s). For the *Uservisits* table, the data accessed per row is 86 bytes, and the time interval between every 100,000 records was an average of 504.2 ms, with a minimum of 285.0 ms, translating into an average access rate of 17.1 MB/s (0.1 Gb/s), and a maximum average rate of 30.2 MB/s (0.2 Gb/s).

Column name	Column size	Comments
adRevenue	4 bytes	sizeof(FLOAT)
destinationUrl	59 bytes	4 byte length; 55 bytes data
visitDate	4 bytes	sizeof(FLOAT)
sourceIp	19 bytes	4 byte length; 15 byte IP
Total	86 bytes	

Table 3: Data accessed per row for Query 3 (Uservisits table)

The relatively lower speed of access on the *Uservisits* table can be explained by the need to filter each row based on the range condition given in the query. Since the above two operations happen in parallel, the total average memory throughput during this phase is $45.1 + 17.1 = 62.2$ MB/sec (0.5 Gb/s). Maximum throughput, assuming both tables are scanned together in an optimal way, is $94.0 + 30.2 = 124.2$ MB/s (1.0 Gb/s). With 7 threads, that's approximately a total memory throughput of 7 Gb/s.

5. FEASIBILITY

We now discuss the potential feasibility of disaggregating memory for analytical queries.

5.1 Reasons to be optimistic

We have purposefully chosen settings and workloads to increase the overall memory access rates of Spark SQL to the extent possible with our hardware. For this reason we are optimistic in the above results, since in real-world deployments, these results are likely to be an upper bound on Spark SQL's potential performance. Additional reasons for this are:

1. Since Spark SQL runs on the Java Virtual Machine, garbage collection pauses - major collections in particular - can interfere with system throughput. We mitigate this by allocating a large amount of heap memory relative to the size of the data set and avoiding major collections entirely, though in a deployed system GC events would reduce memory demand.

2. Concurrency is achieved by partitioning the data and processing different subsets of the data in parallel. Concurrency does not exist within the context of a partition. If a series of operations have to be performed on a row (e.g,. filter, compute an expression on the value, then aggregate), they are performed in sequence and only then is the next row in the partition accessed.

3. It is a new framework, and possibly lacks advanced query optimization features. Since the queries run during this experiment do not benefit from such optimization, their absence should not affect the results.

5.2 Reasons to be pessimistic

Our study is still preliminary, and faces a number of limitations. We focus only on one kind of query (analytical) and restricts its measurement to the rate of consumption of input data. Of course other workloads may result in different bottlenecks and need a different model for analysis, and even within this approach, we have limited our analysis to a subset of published benchmark queries. Considering the limitations of Spark SQL listed in above, the viability of this approach needs to be tested using other frameworks like Impala [13], Redshift [19], and Tez [20].

Finally, we are limited in our hardware in terms of the number of CPU cores that are available. To this last point, a major research question addressing the feasibility of memory disaggregation is to understand the scaling behaviors of the hardware, as well as the query engine. In particular, if the bandwidth available at the NIC grows at a rate comparable to the number of threads available to Spark SQL, then our results will hold up in the future. However if the growth of CPU threads dedicated to query processing grows faster than the aggregate bandwidth of the NIC, then disaggregation will not be feasible without affecting query performance. Such a limitation might not rule out disaggregated designs, however, since it provides a number of other benefits (such as easier management, and incremental upgrades of individual components).

5.3 Suggested improvements

Based on the above experiments and a study of the Spark SQL source code, some improvements to the software architecture present themselves in the context of disaggregated memory. First, the data storage system should provide the ability to address and serve specific data items such as columns, partitions of rows, etc. This will keep aggregate bandwidth requirement to a minimum. Second, prefetching of rows should be implemented for queries which are known to scan all or most of the data set. Pipelining of different phases of a row (as explained above) can help towards this.

For the queries we analyzed, the per-thread memory access rates were all below 25 Gb/s (in some cases much lower). As new Ethernet standards make their way into the market, based on aggregating multiple, parallel, underlying lanes, it would be advantageous to match up these per-thread bandwidth demands with the lanes. For example, 4 Spark SQL threads would match well to a four-lane 100 Gb/s NIC.

6. CONCLUSION

This paper has described a preliminary approach to evaluating the feasibility of disaggregated memory. The approach consists of measuring memory access rates of analytics queries based on the amount of input data accessed per row of the query. A few sample queries drawn from the Big Data Benchmark [3] were used to benchmark this metric for Spark SQL. The results show that it is possible to disaggregate the memory for such workloads using currently available network hardware. Improvements in software architecture can help in performing better in a disaggregated memory environment. While care has been taken to set up Spark SQL so as to get a conservative set of results, more extensive testing needs to be done using different queries and configurations.

Acknowledgements

This work was sponsored in part by the National Science Foundation (CNS-1314921).

7. REFERENCES

[1] Apache Spark version 1.3.0. https://github.com/apache/spark/tree/branch-1.3.

[2] Best Practice Guidelines for ProLiant Servers with the Intel Xeon 5500 processor series Engineering Whitepaper, 1st Edition, Figure 6. ftp://ftp.hp.com/pub/c-products/servers/options/Memory-Config\--Recommendations\--for-Intel-Xeon\--5500-Series-Servers-Rev1.pdf.

[3] Big Data Benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[4] Intel Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor/.

[5] Introducing Yosemite: the first open source modular chassis for high-powered microservers. https://code.facebook.com/posts/1616052405274961/introducing-yosemite-\-the-first-open-source\--modular-chassis-for\--high-powered-microservers-/.

[6] Spark CSV. https://github.com/databricks/spark-csv.

[7] stream-scaling. https://github.com/gregs1104/stream-scaling.

[8] Amplab big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[10] IEEE P802.3bs 400 Gb/s Ethernet Task Force. http://www.ieee802.org/3/bs/index.html.

[11] Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center\--fabric-the-next-generation-\-facebook-data-center-network/.

[12] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

[13] Impala. http://impala.io/.

[14] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.

[15] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[16] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/, 1995.

[17] ConnectX-4 single/dual-port adapter supporting 100 Gb/s with VPI. http://www.mellanox.com/page/products_dyn?product_family=201&mtag=connectx_4_vpi_card.

[18] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. 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.

[19] Amazon AWS RedShift. http://aws.amazon.com/redshift/.

[20] Apache Tez. https://tez.apache.org/.

[21] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.