Big Data Analytics on Cray XC Series DataWarp using Hadoop, Spark and Flink

Robert Schmidtke, Guido Laubender and Thomas Steinke
Zuse Institute Berlin (ZIB)
{schmidtke, laubender, steinke}@zib.de

***Abstract*—We currently explore the Big Data analytics capabilities of the Cray XC architectures to harness the computing power for increasingly common programming paradigms for handling large volumes of data. These include MapReduce and, more recently, in-memory data processing approaches such as Apache Spark and Apache Flink. We use our Cray XC Test and Development System (TDS) with 16 diskless compute nodes and eight DataWarp nodes. We use Hadoop, Spark and Flink implementations of selected benchmarks from the Intel HiBench micro benchmark suite and others to find suitable runtime configurations of these frameworks for the TDS hardware. Motivated by preliminary results in throughput per node in the popular Hadoop TeraSort benchmark we conduct a detailed scaling study and investigate resource utilization. Furthermore seek to evaluate scenarios where using DataWarp nodes might be advantageous to using Lustre as file system backends.**

***Index Terms*—DataWarp; big data; cluster compatibility mode**

I. Introduction

Big data analytics span a wide range of applications, from relational queries over machine learning to real-time stream processing. The introduction of the MapReduce [1] cluster programming paradigm in 2008 allowed such applications to be developed easily and deployed in a scalable and robust manner on commodity hardware clusters. Since then the amount of data to be processed has reached petabyte scales and the variety of applications does not only just include industrial problems, like recommendation engines or real-time bidding, but also scientific ones, such as image analysis and graph processing. As the scientific community has drawn heavily on high performance computing in the past, there is a deep understanding of creating very specific codes tailored to the underlying high performance computing hardware, which differs considerably from commodity cluster setups. HPC nodes are usually diskless and connected using high speed interconnects which allow remote direct memory access, whereas cluster nodes are equipped with a local disk and communicate over Ethernet using the TCP/IP stack and are usually programmed using higher level languages and frameworks.

In this paper we seek to explore the big data analytics capabilities of our Cray XC Test and Development System (TDS), comprised of 16 diskless compute nodes. Selected big data benchmarks are used to evaluate DataWarp as storage backend when processing large amounts of data during analytics tasks and to identify possible bottlenecks, both in the hardware and software setup. We contrast the results with Lustre as storage backend.

We will start by presenting the hardware and software setup we used to run the benchmarks. The following section describes the benchmarks in greater detail. In section IV we present the benchmark results along with a specification of configuration parameters, and provide an explanation as to why the TDS' performance is as observed with a special focus on the influence of the inclusion of DataWarp nodes. We conclude with a summary of our observations and comment on the suitability of the TDS as a big data analytics platform.

II. Hardware and software setup

All results presented have been generated on the XC Series Test and Development System (TDS) at ZIB which is a single cabinet system comprised of 16 compute nodes and eight DataWarp I/O nodes. The compute nodes are equipped with one Intel ten-core IvyBridge Xeon E5-2670v2 processor clocked at 2.5 GHz and 32 GiB of memory.To run Hadoop, Spark and Flink on the TDS, we made use of the Cluster Compatibility Mode (CCM).

We used eight DataWarp nodes over Aries, configured statically as scratch file system. Each DataWarp node contains two 1.6 TiB SanDisk Fusion SX300 SSDs, integrated into the XC Series using PCIe 3.0 to connect to the Cray Aries network, providing 3 GiB/s bandwidth per node. The compute nodes each have access to two Lustre file systems: one with 2.3 PiB capacity on 80 OSTs and a second with 1.4 PiB on 48 OSTs, delivering more than 30 GiB/s and 20 GiB/s I/O bandwidth, respectively.

The TDS was running Cray Linux Environment (CLE) version 5.2.UP04 with support for Cray DataWarp Phase 1. With that, a DataWarp Service (DWS) can be configured to allow dynamically allocating DataWarp capacity striped over all available DataWarp nodes to batch jobs on request (in contrast to Phase 0 where the SSD space of each DataWarp node could only be individually and statically mounted on compute nodes).

For the benchmarks a persistent DataWarp instance was created and configured as scratch space in striped access mode. The instance was activated as static single mount point on all compute nodes. Using this configuration writes are automatically striped over all DataWarp nodes in chunks of 8 MiB. The Lustre file system has been used with default values, which means files are not striped over multiple OSTs.

The Cray DataWarp Service uses the Cray Data Virtualization Service (DVS) to make the SSD space of the DataWarp nodes accessible on each compute node. DVS is a network service that transparently projects the file systems on the DataWarp service nodes to each compute node. Thus, the remote file system appears local to the compute nodes [2]. DVS is lightweight and scales up to tens of thousands of clients accessing the file system. The DVS provides statistics of file system accesses on each compute node (`/proc/fs/dvs`) as well as inter process communication statistics between the nodes which we will use to analyze how Hadoop, Spark and Flink access the file system underneath. For Lustre we use the `/proc/fs/lustre` entries to obtain similar insight. A more detailed discussion of the DataWarp installation at ZIB and results of synthetic I/O and application benchmarks with Cray DataWarp Phase 0 (so-called static DataWarp) was described in [3].

Since all frameworks used require a Java Virtual Machine (JVM) to be present and SSH for deploying services, we use the Cluster Compatibility Mode (CCM) 2.2.1 by Cray. We run all benchmarks on multiple data processing engines: Hadoop 2.7.1 [4], Spark 1.6.0 [5] and Flink 0.10.2 [6]. The benchmarks presented are executed using the Intel HiBench microbenchmark suite 5.0[1], with small adaptations to support newer framework version which we have contributed back to the project [7], along with some adaptations to be run on the TDS [8]. Because the HiBench suite does not include comparable Spark and Flink implementations for TeraSort, we have used a different implementation that uses the same partitioning as the Hadoop implementation from HiBench [9].

We now briefly introduce the main paradigms and frameworks used throughout this paper and the benchmarks.

1) MapReduce: MapReduce is a programming paradigm where computational problems are stated as a series of *map* and *reduce* steps. The mapper consumes arbitrary input and outputs key-value pairs. The values are grouped by key, and the reducer fetches and processes values of the same key[2], and outputs another value computed from the range of input values. Since this approach is inherently parallel, MapReduce can be scaled out easily to clusters consisting of thousands of nodes. The step where data is transferred from mapper to reducer is called *shuffle* and traditionally involves all mappers persisting their key-value pairs to disk, and the reducer fetching the appropriate data over the network. This allows for great robustness as map and reduce tasks can simply be restarted on failure. The downside is a large amount of I/O. The shuffle is part of the reduce, the time period where map tasks are active is called the map phase, and the time period where reduce tasks are active is called the reduce phase. Note that while logically separate, these phases can greatly overlap in practice.

The simplest MapReduce example is the one of counting words: map every $word$ from a text to a pair $(word/1)$, use the $word$ as a key during reduction, which sums up all the 1s per $word$ and outputs the final number of occurrences for each $word$. More complex programs can be specified as a series of map-reduce steps. Apache Hadoop is a popular open-source implementation of MapReduce and is widely used in the industry.

2) YARN: All frameworks support execution on the resource management and job scheduling framework YARN (part of Apache Hadoop), which is what we have used in all our benchmark runs as well. YARN consists of the *ResourceManager* which governs all resources available in the cluster (mainly number of CPU cores and main memory). A *NodeManager* is run once per node in the cluster and runs and monitors containers (dedicated spaces specified by number of CPU cores and main memory). The ResourceManager negotiates resources with the applications and then deploys tasks in containers on the NodeManagers (e.g. a map or reduce task with certain memory and CPU requirements). We dedicate one node in our setup to be the *ResourceManager* taking care of task placement on the remaining worker nodes and overall resource management and utilization. On each worker node (of which we use up to 14 of our 16-node TDS), a *NodeManager* is started, managing memory and cores to allocate for tasks to be executed on it.

We allow YARN to allocate up to 70% of a node's physical memory (22.4 GiB) and up to a total of all (ten) cores per worker node for tasks to be executed on each worker. In Hadoop MapReduce jobs, each map task is granted up to 3 GiB and each reduce task is granted up to 4 GiB of memory per node, allowing for a sufficient degree of parallelism through concurrently running tasks per worker node (we have specified a default map and reduce parallelism of 4 times the number of worker nodes). If memory limits are reached, the tasks spill to disk, which we have redirected either to the DataWarp nodes or the Lustre file system.

3) HDFS: The Hadoop Distributed File System (HDFS) is also part of Apache Hadoop, and manages data in a distributed, non-POSIX fashion (e.g. not supporting concurrent writes, supporting appending). Data is split into large blocks (typically between 64 MiB and 1 GiB in size) and distributed over all *DataNode*s in the cluster. A *NameNode* manages file system metadata and keeps track of where individual blocks are stored. This is to allow data-local computation, where a task is placed on the node that hosts the task's input data, which greatly reduces network traffic in the cluster. Data is persisted into the underlying file system on each node and read/written upon request. We use HDFS for all our experiments, co-locating the HDFS NameNode with the YARN ResourceManager and the HDFS DataNodes with the YARN NodeManagers, resulting in an equal number of worker nodes, DataNodes and NodeManagers in the cluster. We use a HDFS block size of 256 MiB for all setups, meaning files written to HDFS are split into chunks of 256 MiB and striped over a corresponding number of DataNodes. We have disabled file replication for all our benchmarks, i.e. each file stored

[1]which in turn internally uses Hive 1.2.1, Kafka 0.8.1, Mahout 0.11.1 and Zookeeper 3.3.6

[2]or multiple keys, in which case the data is sorted on these keys before reducing

in HDFS has a replication factor of one. We run all our experiments once with DataWarp as HDFS' underlying file system, and once with the Lustre file system.

4) Spark and Flink: Apache Spark and Apache Flink have been conceived as extensions to the MapReduce paradigm, with the main advancements being reducing I/O by processing as much data as possible in memory, a wider array of programming primitives, and the ability to consume and produce streaming data. Programs for these data processing engines are specified as Directed Acyclic Graphs (DAGs) that specify a data flow from source(s) to sink(s) as a chain of transformation operators, such as map, filter or join.

The basic data abstraction in Spark (Flink) is the Resilient Distributed DataSet RDD (DataSet), which is basically a collection of objects distributed over many hosts.

Spark (Flink) starts an Executor (TaskManager) on each worker node, with a certain number of cores (slots) each that will do the actual processing. Operators in the DAG are placed on worker nodes with a certain degree of parallelism (usually the number of input splits that need to be processed, i.e. the number of blocks stored in HDFS that represent the input data). Since the location of the blocks is known via HDFS, the operators can be placed on the nodes hosting the actual data. Furthermore, operators can be fused where possible to reduce network traffic (e.g. a map and a filter operation usually only operate on local data and hence can be merged into one operation instead of splitting it over nodes). This way, the logical DAG is mapped to the physical cluster and data flows along it during program execution. We configured the number of Executors (TaskManagers) to be equal to the number of worker nodes, with each Executor (TaskManager) controlling four cores (slots) and a total of 20 GiB of memory. The Executors (TaskManagers) are co-located with the NameNodes and DataNodes to allow tasks placed on them to process local data.

Spark and Flink allocate a large portion of main memory where data is processed and kept, thus eliminating the need for disk I/O during the shuffle phase. Fault tolerance is achieved by the concept of lineage, where an RDD (DataSet) can be recomputed by following the chain of transformations backwards. Richer primitives speed up computation as the program does not need to be broken down into small map and reduce steps. If tasks require more memory than is available (e.g. because of a very expensive user defined function (UDF) in a map task), then graceful spilling to disk happens. We redirect this spillage to DataWarp and Lustre as well.

Notable differences between Spark and Flink are the support for iterations and true stream processing in Flink, whereas Spark emulates stream processing by windowing streams into micro-batches of configurable size.

5) Kafka: Apache Kafka is a persistent distributed commit log that can be used for messaging in a publish-subscribe fashion. Arbitrary messages can be published by producers under a certain topic, Kafka then persists them and allows consumers to subscribe to topics and process the data. Fault-tolerance for consumers is built-in through roll-back, and parallelism is achieved by splitting a topic into a number of partitions which can be consumed concurrently by many clients. In our setup we use Kafka in the streaming benchmark, where we feed data into Kafka from multiple producers, and let Spark and Flink consume the data in a streaming manner. We place Kafka on a dedicated set of four nodes, each operating with eight network threads. The persistent directories are placed on Lustre, as unfortunately DataWarp does not support read/write memory mapped files, which Kafka relies upon.[3]

6) Remark: All of the above frameworks benefit from one or more fast local disks to read data from and write data to, a large amount of main memory for reducing the need for disk I/O, many cores for concurrent processing and a network for fast TCP/IP, as this is the main method of communication between the distributed components. Our TDS differs significantly from these requirements, providing only remote storage, a moderate amount of main memory and CPU cores, as well as a highspeed Aries interconnect which cannot be harnessed by these frameworks without modifications. In the following we seek to explore the capabilities of our TDS for typical Big Data Analytics tasks.

III. Benchmarks

In order to evaluate the TDS' performance we have used three different benchmarks with different characteristics: TeraSort, Streaming and SQL. All benchmarks are executed using a fixed problem size and a varying number of compute nodes (three, seven, eleven and 15) to examine strong scaling, with either Lustre or DataWarp as storage backends. For each number of compute nodes we dedicate one to host various managing processes, such as the YARN ResourceManager and the HDFS NameNode, leaving two, six, 10 and 14 worker nodes for actual computation.

We will briefly present the benchmarks in the following sections. For TeraSort we have additionally added a weak scaling setup for Hadoop where the amount of work per node is kept constant as we increased the number of nodes. Because of the versatility and general applicability of the TeraSort benchmark, we focus primarily on this benchmark to analyze various performance observations, and only include a subset of information for the other two benchmarks.

A. TeraSort

The TeraSort benchmark is a popular benchmark for data processing engines and has been used widely on a variety of setups to assess a cluster's performance. The initial implementation is from Hadoop [10] and consists of sorting 1 TiB of data distributed over as many nodes as there are in the cluster. The data is organized in rows of 100 bytes each, with the first ten bytes being the key to sort on and the remaining 90 bytes forming the values that will be sorted.

The benchmark stresses HDFS, as both the input to be read and the output to be written amount to 1 TiB of data

[3]We received `mmap: Function not implemented.` when using `PROT_READ|PROT_WRITE` with `mmap` instead of just `PROT_READ`.

and additionally, for Hadoop, all data is directly written to and read from the underlying file system during the shuffle phase. Because a global ordering on the keys needs to be achieved, the network is stressed as well as all data needs to be shuffled to their corresponding correct output locations. Each reducer's CPU and memory is stressed during the local sorting of keys. Because of its versatility, this benchmark is useful for assessing how well the MapReduce framework is configured in terms of overall task parallelism and memory per task, and we used the configuration values determined with TeraSort for the subsequent benchmarks as well.

In order to reach the goal of sorting 1 TiB of data, we have generated 11 billion rows, of which each of the worker nodes received an equal share. The data is generated during the *TeraGen* step, which we have not analyzed in detail as it is constant for each benchmark execution.

First during the sort, each map task tokenizes the input records into the ten-byte-key and the 90-byte-value. Next, a custom partitioner assigns a partition number to each key produced by the map task to determine the shuffling, with as many partitions as there are reduce tasks in the job. This is done by sampling 100,000 keys and sorting them into a Trie structure to determine the key range for each reducer, such that for each reducer r and key k the following holds: $samples[r-1] <= k < samples[r]$. By looking up each key from the input data in the trie, the partition number for this key can be determined and thus assigned to a specific reducer. This guarantees that each reducer receives a distinct key range and, after merge-sorting the key-value pairs locally, a global sorting is achieved.

For benchmarking Hadoop we have used the implementation shipped with the HiBench suite. For Spark and Flink we have used a different implementation that uses the same partitioner as the Hadoop implementation for comparative reasons [9], because the Spark HiBench TeraSort implementation generated heavily skewed partitions.

B. Streaming

The streaming benchmark of the HiBench suite starts by generating a large number of d-dimensional double-precision floating-point vectors, around the centers of a fixed number of clusters. These are persisted into HDFS, where they are read from afterwards and published to Kafka, using four concurrent producing threads. For this benchmark we use the Flink and Spark frameworks only, as Hadoop does not support streaming data[4]. The frameworks subscribe to the Kafka topic the vectors are published under and aggregate statistics (*min*, *max*, *sum* and *count*) on one of the d dimensions.

This benchmark assesses the network stack in terms of throughput and latency for small messages of about 80 bytes and the frameworks' abilities to keep up with the incoming data as they are aggregating statistics, with the added constraint of maintaining them globally and not just locally for each worker.

We will use varying setups that trade off minimizing latency, where each record is published to Kafka individually and processed immediately by the frameworks (Flink supports per-record processing; in Spark this needs to be approximated by using a very small microbatch size, as Spark is an inherently batch-oriented framework), and maximizing throughput where records are processed in large windows by the frameworks.

C. SQL

The SQL benchmark included in the HiBench suite is based on a 2009 SIGMOD paper [11] and is comprised of multiple queries that operate on large data resembling HTML documents and web server log files. We have picked a join/aggregation query that outputs users in order of descending total ad revenue they have generated during all their page visits within a specific time period, transforming an input data set of 878 GiB to an output data set of 3.5 GiB.

This benchmark stresses the same features as the TeraSort benchmark, except the large amount of input data to be processed is more complex which means that a series of map and reduce phases needs to be executed (instead of just one for TeraSort), hence involving more load especially in the shuffle and reduce steps where joining and sorting are performed. As the output data is small compared to the input data set, the frameworks' abilities to push down operations in order to reduce network traffic early in the job is tested as well.

The relevant parts of the data for this query are:

- a table `Ranking` with a mapping from `url:String` to `rank:Integer`, and
- a table `UserVisit` with a mapping from `ip:String` to `url:String`, `date:Date` and `revenue:Double`.

The query first finds the average page rank of all pages visited within a specific period of time and total revenue generated for each user by calculating `avg(rank)` and `sum(revenue)` for each user's `ip` accordingly. This includes a join on `url` and a filter on `date`. Second, the query sorts the result in descending order on `sum(revenue)` per user to output the `ip`, `sum(revenue)` and `avg(rank)` for users ranked by the total revenue they generated within that period of time.

IV. Results

Because the benchmarks themselves took a great effort to configure and in some cases a long time to execute, the results presented depict the best run out of several we have conducted. The numbers presented are mostly taken from the performance counters of the corresponding execution engines, with Hadoop providing the most diverse set of statistics (we have added more custom file system counters), Spark providing access to these only via the Web UI, and Flink being the least verbose of all systems (we have added automatic dumping of job statistics after each run).

[4]*Hadoop Streaming* might suggest this feature, however it only means that standard UNIX streams are used to communicate with arbitrary mapper/reducer commands.

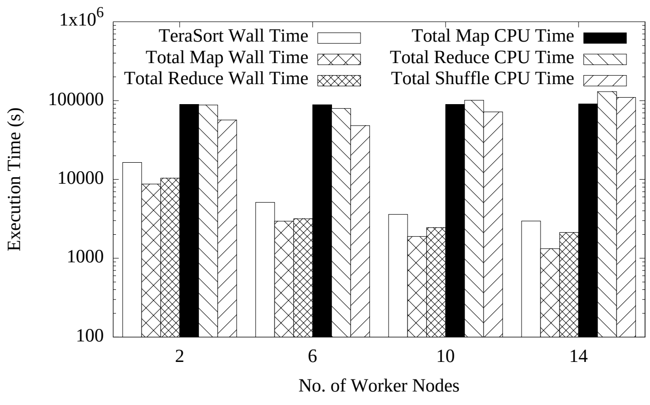

Fig. 1. Hadoop MapReduce TeraSort of 1 TiB on DataWarp

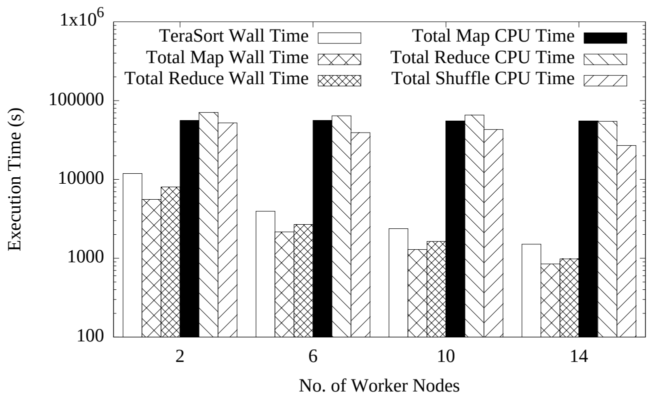

Fig. 2. Hadoop MapReduce TeraSort of 1 TiB on Lustre

Before each run, we set up everything from scratch, formatting all data in HDFS and the NameNode, and generate the input data anew and copy it to the appropriate input locations.

In the following analysis we will use the *throughput* for comparing results. We define throughput as the result set size divided the time it took to generate that result set. This is synonymous with the *cluster throughput*, and to obtain the *per-node throughput* we divide the cluster throughput by the number of worker nodes participating in the benchmark.

A. TeraSort: General observations

We have conducted the TeraSort benchmark for sorting 1 TiB of data on two, six, ten and 14 worker nodes with an additional node hosting the ResourceManager and the NameNode. Each of these setups was run once using the DataWarp service as storage backend for HDFS, and once using the Lustre file system. For each of these setups we have chosen Hadoop MapReduce, Spark and Flink as execution engines to compare performance of these different common analytics frameworks. The Hadoop, Spark and Flink implementations are comparable as they use the same partitioner determining the assignment from map output to reduce input.

For the Hadoop configurations we have included one plot per benchmark run, one depicting the execution times of different phases of the benchmark (Figures 1 and 2). Weak scaling using Hadoop on DataWarp is shown in Figure 3. For the Spark and Flink configurations we present one plot each, showing the execution times of different stages of the job (Figures 4 through 7). A detailed discussion of file system performance is presented in Section IV-B.

Because the underlying HDFS was configured to use blocks of 256 MiB, and one map task in the TeraSort benchmark works on one block of input data, the total number of map tasks in the Hadoop jobs was approximately constant over all benchmark runs, i.e. 1 TiB of input data divided by 256 MiB of work per map task equals 4096 map tasks to be executed in theory. In practice we have observed a total of 4200 map tasks executed, which can be explained by slightly imperfect balancing during the generation of the TeraSort data during the TeraGen step. For the reduce phase we have empirically determined a parallelism of 60 per worker node to work well, which means for two worker nodes, there are 120 reduce tasks, for six worker nodes there are 360 reduce tasks and so forth.[5]

We observe sublinear scaling when using both DataWarp and Lustre as file system backend with respect to the overall execution time *TeraSort Wall Time*, which includes the *Total Map Wall Time* and the *Total Reduce Wall Time*. From these three metrics we can see that the map and reduces phases have a large overlap.

With a growing number of worker nodes, almost linear scaling is observed as well when just examining the wall time spent on the map tasks, which indicates an appropriate degree of parallelism for the map phase of the benchmark. The overall work performed during the map phase remains constant, as shown by the constant cumulative amount of CPU time spent on all map tasks.

However, the amount of time spent during the reduce phase does not scale linearly[6], even when cleaned from bad reduce tasks. The reduce phase consists of the actual reducing and the shuffle phase, which is responsible for assigning each mapped key-value-pair from the map phase to the correct reducer. This involves an all-to-all communication step, which, as more and more workers are added, severely dominates the reduce phase: The *Total Shuffle CPU Time* is part of the *Total Reduce CPU Time* in Figures 1 and 2, hence the shrinking difference between the two is the actual reducing.

Another observation that draws immediate attention is the fact that the execution times of the benchmarks vary by up to a factor of two between using DataWarp and Lustre as storage backends, resulting in total cluster throughputs from 64 MiB/s to 352 MiB/s for Hadoop on DataWarp, and 88 MiB/s to 690 MiB/s on Lustre. The per-node throughputs follow a more interesting pattern: on DataWarp it is highest (34 MiB/s) when using 6 worker nodes, and decreases down to 25 MiB/s with different numbers of worker nodes. On Lustre the per-node throughput is highest with 14 worker nodes (50 MiB/s) and at

[5]We will comment on the implications this has for spilling and briefly mention an alternative below.

[6]which includes the time taken by failed and restarted tasks, of which there are approximately 1%

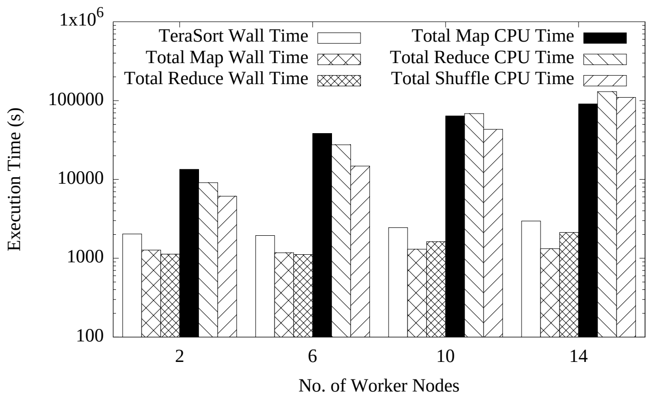

Fig. 3. Hadoop MapReduce TeraSort of 1/14 TiB per Worker on DataWarp

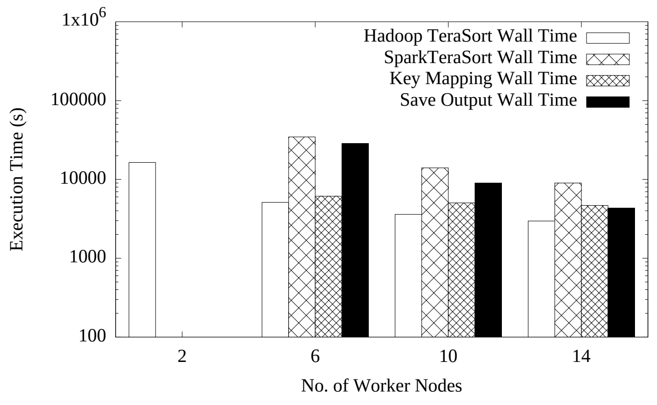

Fig. 4. Spark TeraSort of 1 TiB on DataWarp

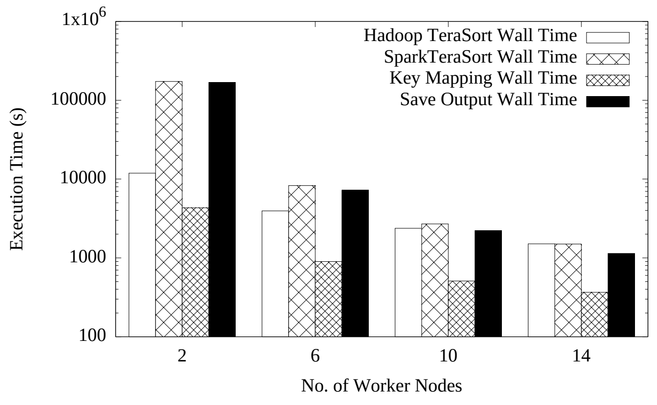

Fig. 5. Spark TeraSort of 1 TiB on Lustre

a constant 44 MiB/s with all other configurations. Furthermore we observe an increasing speedup of Hadoop on Lustre over Hadoop on DataWarp of 1.3 for two nodes to 2.0 for 14 nodes. We will discuss this difference later on in more detail when correlating performance with file system metrics.

The observed per-node throughputs are moderate, in particular when comparing to TeraSort records recently set by the industry. In 2014, MapR achieved around 19 MiB/s per node in a cluster of 1003 virtual nodes, and 101 MiB/s per node in a cluster of 21 nodes using a tuned version of Hadoop that incorporates many natively implemented improvements [12]. Cisco reports a per-node throughput of 220 MiB/s in a cluster of 16 nodes in 2013 [13]. It is important to note that both companies used clusters with an aggregate memory multiple times the size of the input data (2 TiB and 4 TiB), many cores (2x16 per node) and local disks, whereas our TDS only has an aggregate 512 GiB of memory[7], ten cores per node and access to remote DataWarp and remote Lustre file systems. The first popular records were set by Yahoo! in 2007 and 2008 on a cluster of 910 nodes with 8 cores, 8 GiB of memory and four SATA disks per node, sorting 1 TiB of data in 297 and 209 seconds [14], yielding a per-node throughput of 3.9 and 5.5 MiB/s.

In Figure 3 we show the weak scaling properties of Hadoop TeraSort on DataWarp where we have kept the amount of data to be sorted at a constant 1/14th of 1 TiB per worker node. We observe an increase in *TeraSort Wall Time* which is entirely due to the corresponding increase in *Total Reduce Wall Time*, which in turn is caused by the increased amount of time spent during the shuffle phase. This can be seen when looking at the corresponding *Total CPU Time*s, where the time spent on reducing increases over-proportionally due to the corresponding increase in shuffle time.

We now turn our attention to the TeraSort benchmark runs on Spark, where we present results in Figures 4 and 5 for configurations similar to Hadoop. Spark job essentially consists of three major steps: reading the input and partitioning according to the keys, mapping the keys and values to a serializable and thus transmittable data type, and sorting and writing the output to HDFS. These steps are less separated from each other compared to Hadoop, because Spark uses fusion of tasks where possible. As the plots show, sorting and saving the output are fused together, as they take up almost the entire *SparkTeraSort Wall Time*, with the key mapping and partitioning running concurrently as well. Note that the Spark TeraSort benchmark using two worker nodes on DataWarp consistently failed with an `InvalidStreamHeaderException` in the partitioning step which we were not able to mitigate, and hence had to exclude results for this configuration.

We have included the *Hadoop TeraSort Wall Time* to ease comparison between the two frameworks. While Spark scales super-linearly both on DataWarp and Lustre, we observe that Spark runs between a factor of 6.8 (six nodes) and 3.0 (14 nodes) longer on DataWarp, and between a factor of 14.6 (two nodes) and 1.0 (14 nodes) longer on Lustre than the corresponding Hadoop configuration in our array of benchmarks. Note that on DataWarp we do not have Spark results for two worker nodes, but extrapolating the scaling behavior and considering the results on Lustre, the slowdown would likely have been between one and two orders of magnitude. We furthermore observe sublinear scaling behavior of the key

[7]for 16 nodes, however we never use more than 14 workers, giving an effective aggregate memory of 448 GiB

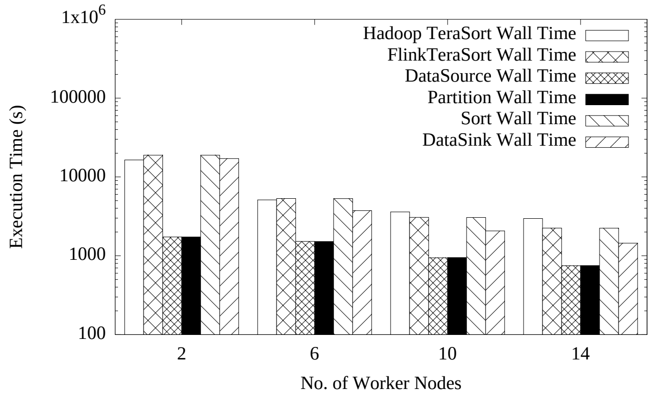

Fig. 6. Flink TeraSort of 1 TiB on DataWarp

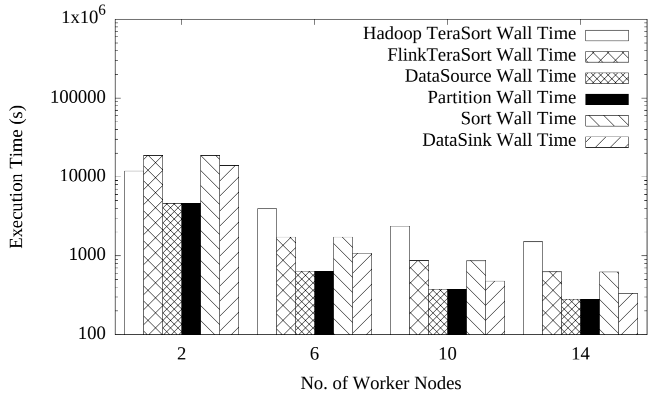

Fig. 7. Flink TeraSort of 1 TiB on Lustre

mapping phase on DataWarp, whereas this phase scales linearly on Lustre. Saving the output scales super-linearly on both file systems. We will again keep this in mind when discussing file system counters collected by DataWarp and Lustre, which give an indication on why Spark's I/O performance is so unfortunate, especially with reads on DataWarp.

As a consequence of this, Spark's per-node throughput for the TeraSort benchmark is lower than Hadoop's by the same factors we observed when discussing execution time with 5 to 8 MiB/s on DataWarp, and 3 to 50 MiB/s on Lustre, with only the 14-node configuration achieving similar performance. Note that in contrast to Hadoop, the per-node throughput increases with an increasing number of worker nodes under Spark. This demonstrates the in-memory characteristics of Spark, especially during the shuffle phase which is a lot less expensive. It is unfortunate that we could not scale out further, as Spark's scaling behavior promises a better performance gain than Hadoop's scaling behavior.

Furthermore we observe that with an increasing number of nodes, the speedup of Spark on Lustre over Spark on DataWarp increases from 4.2 for six nodes to 6.0 for 14 nodes.

Finally, we ran the TeraSort configurations on Flink as well, with results depicted in Figures 6 and 7. Similar to Spark, Flink aims at processing as much data in memory as possible,

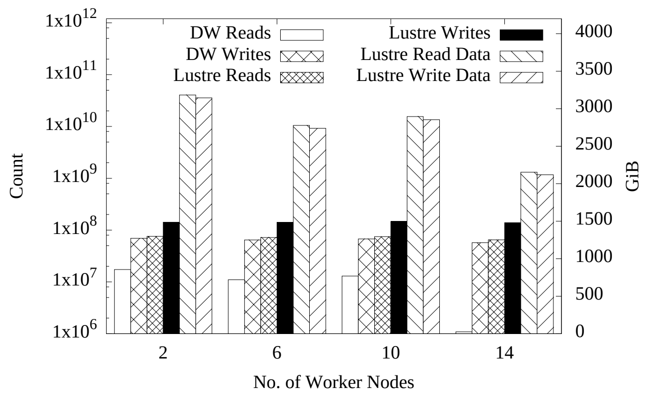

Fig. 8. Aggregate DVS/Lustre file system counters for the Sort phase of TeraSort on Hadoop

with detailed counters on I/O not being available in the Flink release we used for this performance comparison. We observe that Flink's execution times are almost always as good as or better as the corresponding Hadoop runs with stronger scaling, especially on Lustre. Consequently, Flink outperforms Spark for every configuration of the TeraSort run we conducted, by factor of 6.5 (six nodes) to 4 (14 nodes) on DataWarp (where Spark results were available), and by a factor of 9.3 (two nodes) to 2.4 (14 nodes) on Lustre. Per-node throughputs range accordingly from 28 to 33 MiB/s on DataWarp and from 28 to 119 MiB/s on Lustre, again with increasing per-node throughput as the number of worker nodes increases.

Similar to Spark, reading and partitioning the keys are fused together (the *Partition Wall Time* is part of the *DataSource Wall Time*), and sorting and saving the output as well (the *DataSink Wall Time* is part of the *Sort Wall Time*). The *DataSink Wall Time* decreases faster than the *Sort Wall Time*, indicating an increasing efficiency with more worker nodes, as relatively less time is spent on writing the output, compared to actually sorting it, as reflected by the increasing per-node throughput.

We observe an increasing speedup of Flink on Lustre over Flink on DataWarp from 1.0 for two nodes to 3.6 for 14 nodes.

B. *TeraSort: File system observations*

Following the observations from the previous section, we will now look into the file system counters collected by DVS and Lustre during the TeraSort benchmark on Hadoop, Spark and Flink over two, six, ten and 14 worker nodes, shown in Figures 8, 9 and 10. These counters, as opposed to the counters that each framework provides, capture all output and are therefore more reliable and complete. We aim to clarify three issues which we have briefly touched upon above:

1) How much data is written to/read from the file system exactly?
2) How does read I/O on DataWarp differ from Lustre under Spark?
3) How does I/O differ with varying number of nodes as we observe an increasing speedup of Lustre over DataWarp?

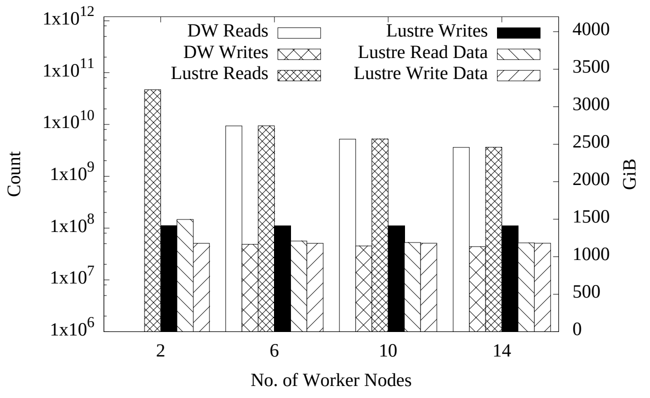

Fig. 9. Aggregate DVS/Lustre file system counters for the Sort phase of TeraSort on Spark

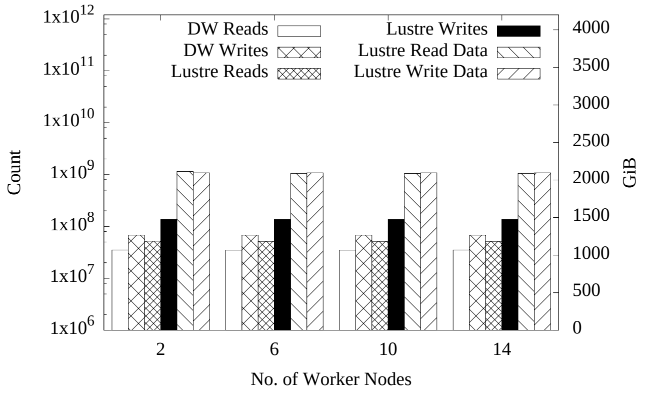

Fig. 10. Aggregate DVS/Lustre file system counters for the Sort phase of TeraSort on Flink

The plots depict the total number of read and write operations on DataWarp and Lustre, and the total number of bytes written to/read from Lustre, as we were unable to determine the number of bytes written to/read from DataWarp from the DVS counters. We will therefore assume similar values for DataWarp, as the Hadoop file system counters indicate as well. Keep in mind that for TeraSort on Spark and DataWarp with two worker nodes, we have no results because of an exception at runtime. Note that we have started all of the frameworks used on Lustre, which is why the written data reported can slightly exceed expectations because of logging activities of these frameworks.

For Hadoop we observe total data read/written of around 3 TiB for two, six and ten worker nodes, as we have expected: the entire data set is read once at the beginning, once during shuffle, and once in total because of spillage. The same holds true for the written data. On 14 nodes we observe I/O of only 2 TiB. This is because there are sufficiently many reducers such that the amount of data to be sorted by each of them fits in their memory, and therefore no spilling to disk is necessary. In effect, 2 TiB is the lower boundary for disk I/O when sorting 1 TiB of data using Hadoop.

While it is not optimal to observe this large amount of spilling to disk, we have to re-iterate that common Hadoop deployments run on clusters with more nodes and more memory than what we have at our disposal, and therefore these observations should present an edge-case of Hadoop usage. Seeing that our unaltered configuration is able to achieve zero-spill execution when scaled to 14 nodes confirms this.[8]

The total number of read and write operations on DataWarp and Lustre do not vary by much between the runs on two, six and ten nodes, however on 14 nodes we see a drop of an order of magnitude in reads from DataWarp. This is surprising, as we have to expect that the amount of data processed only shrank by 1/3, and the number of write operations on DataWarp and read/write operations on Lustre have not decreased.

This is an issue we have to leave open for future investigation, especially since the resulting average read sizes on DataWarp would range from 192 KiB to 2 MiB, while the DVS counters reported maximum read and write sizes of only 64 KiB. The average read and write sizes on DataWarp range from 39 to 47 KiB, and average read and write sizes on Lustre range from 16 to 44 KiB, which is in line with the maximum read size of 730 KiB reported by Lustre.

The observed increasing speedup of Lustre over DataWarp does not show in the total number of read or write operations. When using DataWarp as file system backend for the DataNodes, we placed them in the statically configured DVS mount point that automatically distributes data over our eight DataWarp nodes with two SSDs each. Client-side caching is not supported on DataWarp yet and the layer is currently under development, potentially speeding up I/O patterns as observed in our benchmarks, mainly many small reads and writes [15].

We have examined the Hadoop framework's file system counters as well and recorded overall data read and written as well as the corresponding times to do so, as the DVS counters do not provide timings of read and write operations. From these values we observe a read rate on DataWarp of about 240 MiB/s and a write reate of 155 MiB/s, independent of the number of nodes participating in the benchmark. When running an IOR on DataWarp with block sizes of 64 KiB to simulate Hadoop's I/O patterns, we observe read and write rates of 40, 70, 140 and 270 MiB/s for 8, 16, 32 and 64 parallel threads. These results need to be reconciled with the measurements from Hadoop, of which we have yet to determine their accuracy. Unfortunately, these counters are not available in Spark and Flink.

Spark shows different behavior than Hadoop: Overall a lot less data is written to/read from disk which is due to the in-memory characteristics of Spark. We observe an amount of data read decreasing over the runs and approaching the amount of data written, which is in the order of 1 TiB each, as is the lower bound for sorting 1 TiB of data in general as the input needs to be read once, and the output needs to be written once.

[8]We ran Hadoop TeraSort using ten nodes on DataWarp again with 1024 instead of 600 reducers, and while indeed the spill to disk reduced to zero, the overall execution time increased by more than 10% with an otherwise unaltered configuration.

Spark reports a shuffle size of 128 GiB per run, which needs to be read and written as well, explaining the total amount of data read and written above 1 TiB.

Spark issues two to three orders of magnitude more reads to DataWarp and Lustre compared to Hadoop, which, in conjunction with overall reduced amount of data read and written, reduces the average read operation to at most 340 bytes, severely limiting overall performance as discussed earlier. The number of reads scales anti-proportionally with the number of nodes, with a constant number of writes that is slightly smaller compared to the number of writes Hadoop issues.

Flink virtually generates the same I/O for every configuration we have used. While reading and writing almost twice the amount of data from/to the file system than Spark, Flink does so with two to three orders of magnitude less operations for the read case, achieving read sizes similar to Hadoop for Lustre.[9] This very balanced I/O profile is likely due to Flink's rigorous memory management off the JVM heap and explicit controlling of spill.

It is interesting to note that for all cases, the number of read and write operations performed on Lustre is larger than the number of read and write operations performed on DataWarp for the corresponding configurations. This will be explored in depth in future experiments we plan.

C. Streaming

We ran the streaming benchmark for collecting statistics on 100 million records of about 80 bytes each on two, six and ten worker nodes using Spark and Flink, because Hadoop does not support streaming applications. An additional four nodes are dedicated for Kafka, each of which runs eight network processing threads, which is why we cannot allocate 14 worker nodes as in the previous benchmark, because our TDS is comprised of 16 nodes in total. Four of the worker nodes run an additional producer thread, which reads the vectors from HDFS and publishes them to Kafka, achieving an aggregate rate of about 200,000 records published to Kafka per second. The topic the vectors are published under has 16 partitions to allow for sufficient consumer parallelism.

The HDFS setup is as previously described, with storage directories placed on either DataWarp or Lustre. The Kafka log directories where the actual data is stored are placed on Lustre as well, as they are memory mapped files, which are currently not supported for read/write access in DataWarp.

We use Spark and Flink for this benchmark with varying sizes of processing windows in which the vectors are collected and then processed as a batch, as well as one-record-"windows" in Flink (which supports true stream processing). Larger window sizes trade off latency for throughput, smaller windows trade off throughput for latency.

We tried windows of 1 and 10 milliseconds in Spark, these jobs however failed with prematurely aborting tasks due to an `InterruptedException`, so we could not simulate "true" stream processing in Spark. Using windows of 100 milliseconds, Spark finished successfully in only two cases (with six nodes on DataWarp and ten nodes on Lustre), which is why we present results only for 1000 millisecond windows in Spark, as the results for 10,000 millisecond windows are identical except for a ten-fold increase in average latency (as is expected). Flink does not support windows smaller than 50 milliseconds, except true stream processing one record at a time, which is why we present the stream processing results and the 1000 millisecond window results. Flink managed to run successfully with windows of 100 and 10,000 milliseconds as well, but again the results are identical except an according decrease/increase in average latency.

Execution time, average latency and throughput for Spark using 1000 millisecond windows on DataWarp and Lustre are presented in Figures 11 and 12 alongside the appropriate Flink results, and Figures 13 and 14 show the Flink results for one-record-at-a-time processing.

For Spark in the 1000 millisecond window we see total execution times between 541 and 568 seconds and throughputs between 176,000 and 185,000 records per second over all configurations, both on DataWarp and on Lustre. These values indicate no dependence on either the number of nodes involved, nor the file system used underneath. However, ex-

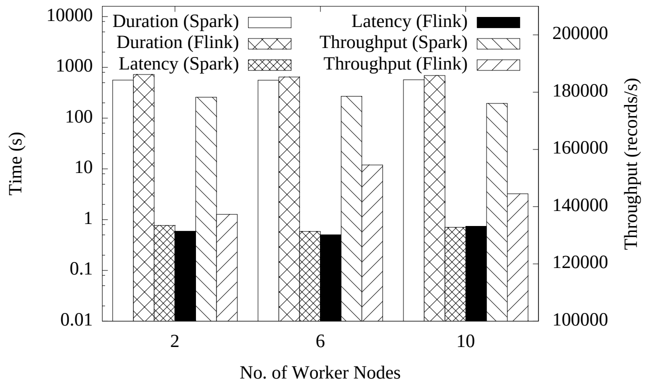

Fig. 11. Streaming on DataWarp with windows of 1000ms

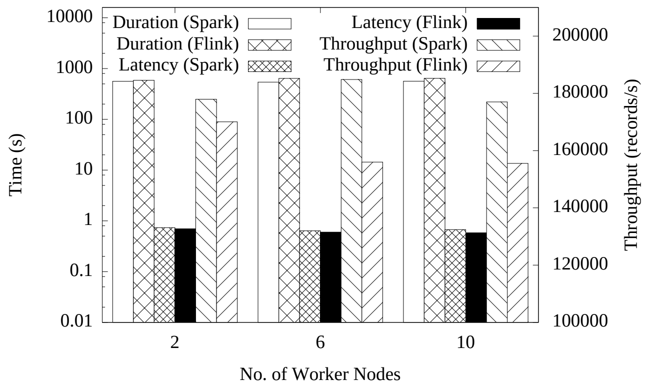

Fig. 12. Streaming on Lustre with windows of 1000ms

[9]As to why the amount of data read and written is always 2 TiB each we can only speculate, as Flink does not provide such detailed statistics in the release we have used.

amining the latencies, we observe higher variance, as they range between 586 and 768 milliseconds on DataWarp, and between 637 and 738 milliseconds on Lustre. Given that the average latency within a 1000 millisecond window should be 500 milliseconds for steady ingestion rates, these latencies indicate some processing delay per batch, which however can be compensated before the next batch starts. The sizes of the batches Spark processes range between 175,000 and 200,000 records, with an average of 192,000 records. That is, Spark can ingest records almost as fast as they are produced.

In order to push latencies over 1000 milliseconds, we would need to produce a lot more records concurrently, which our TDS does not have the resources for. In fact, we ran the streaming benchmark once with eight Kafka nodes and eight concurrent producers emitting 400,000 records per second, with just six worker nodes dedicated to running Spark on DataWarp. Spark achieved 305,000 records per second of throughput and an average latency of 670 milliseconds, which is worse than the corresponding run using six worker nodes on DataWarp and just four Kafka nodes and four concurrent producers (see Figure 11), but still far from overloading the framework.

For Flink in the 1000 millisecond window we see total execution times between 588 and 724 seconds and throughputs between 137,000 and 170,000 records per second over all configurations, both on DataWarp and on Lustre. Both metrics are worse than Spark's results, as the minimum execution time for Flink is higher than the maximum execution time for Spark, and the maximum throughput for Flink is lower than the minimum throughput for Spark. Furthermore we observe decreasing throughput on Lustre and constantly worse throughput on DataWarp, with the differences between DataWarp and Lustre being higher and more erratic compared to Spark. However, Flink achieves latencies between 498 and 793 milliseconds on DataWarp, and between 576 and 700 milliseconds on Lustre. With the exception of ten nodes on DataWarp, Flink always has the lower latency compared to Spark for the corresponding configuration. For Flink we do not have metrics detailed enough to comment on the average batch size. However, given the lower throughput, we assume it to be less than what Spark can process.

Finally, we ran the Streaming benchmark on Flink using true stream processing, resulting in sub-second latencies between 36 and 56 milliseconds as can be seen in Figures 13 and 14. The overall execution time matches the corresponding times on DataWarp and Lustre with a window size of 1000 milliseconds. Again we observe only moderate throughput, with DataWarp being the worse one, and with decreasing throughput similar to the execution with a 1000 millisecond window size on Lustre.

These results are not intuitive, as Flink's architecture is of inherent streaming nature, and thus we need to dedicate separate studies to examining Flink's configuration.

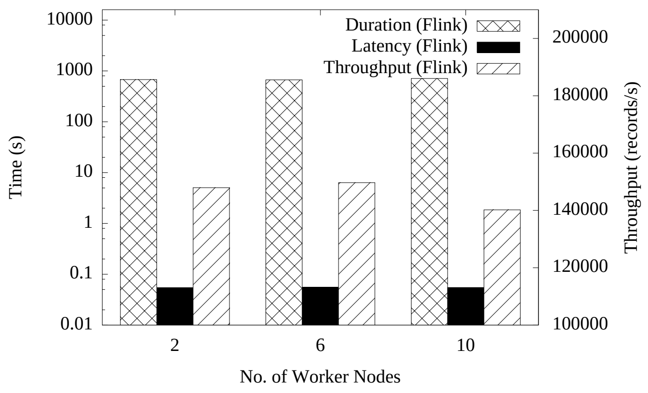

Fig. 13. Streaming on DataWarp with true stream processing (only Flink data available)

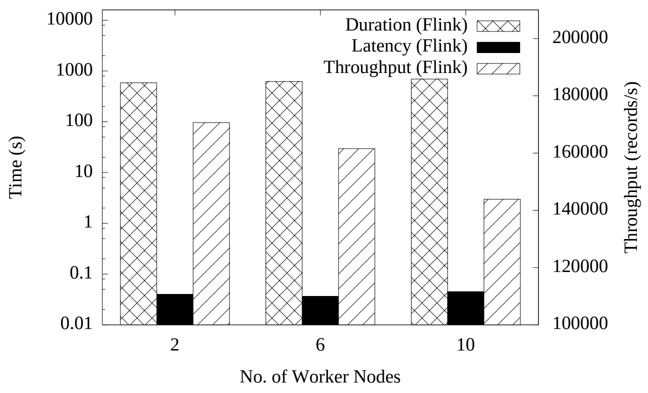

Fig. 14. Streaming on Lustre with true stream processing (only Flink data available)

D. SQL

We ran the SQL benchmark for ordering users by the total ad revenue they generated during their page visits within a specific time period on a data set comprised of $1.2 * 10^8$ rankings and $5 * 10^9$ user visits. The resulting set contains $1.2 * 10^8$ rows. This amounts to 878 GiB of input data and 3.5 GiB of output data. We used two, six, ten and 14 worker nodes, with the HDFS setup as previously described and placed on either DataWarp or Lustre. Cluster throughput is reported in terms of output data. Hadoop and Spark both execute the same Hive SQL file in which the query is specified; in Flink we programmed the query using Flink's Table API for SQL-like queries. The results of all three engines match.

We report execution time and throughput for Hadoop, Spark and Flink on DataWarp and Lustre in Figures 15 and 16. Note that Spark failed to run successfully on two nodes using DataWarp with an `EOFException` which we could not rectify and hence had to exclude results for this run.

We observe linear scaling behavior with Spark, except for 14 nodes, sublinear scaling behavior for Flink and sublinear scaling behavior for Hadoop except for 14 nodes. Hadoop's performance is the worst of the three engines across all runs,

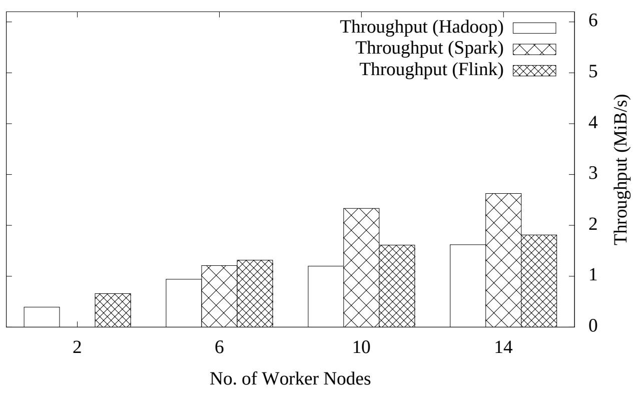

Fig. 15. SQL on DataWarp using Hadoop, Spark and Flink

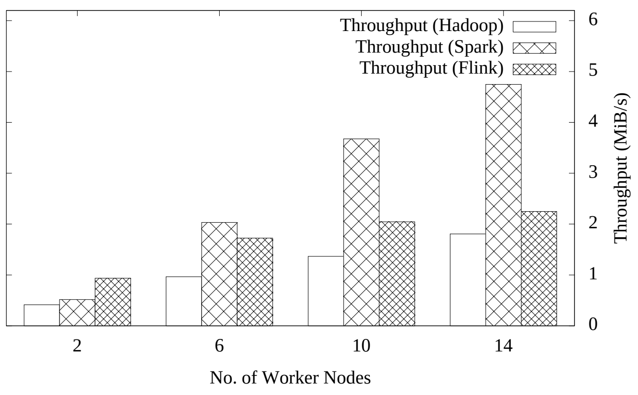

Fig. 16. SQL on Lustre using Hadoop, Spark and Flink

approaching Flink's performance with increasing number of nodes. For six nodes on DataWarp and two nodes on Lustre, Flink is the fastest execution engine for this benchmark, in all other cases Spark's performance is the best with an increasing gap as per the observed scaling behaviors of all three engines.

Given the fact that Hadoop and Spark execute the same Hive queries it is worth emphasizing the differences in execution time between these two engines, with Spark being between 1.3 and 2.7 times faster than Hadoop. This is likely due to the reduced disk I/O during the shuffle phase, as we will explain shortly. We also observe that the overall execution times are lower, and the cluster throughputs therefore higher on Lustre than on DataWarp for all frameworks and their corresponding configurations. This hints at the fact that I/O is indeed one of the limiting factors here for all execution engines.

Despite being an in-memory data processing engine as well, Flink fails to keep up with Spark's performance as the number of worker nodes increases. This may be due that we had to make use of Flink's Table API, as Flink does not support executing Hive queries in release 0.10.2, which is the release we used. Furthermore, the API is still in beta and does not support an `orderBy` statement, which is why we had to resort to Flink's traditional DataSet API to implement this function as a sort on exactly one node, whereas Spark seems to be able to

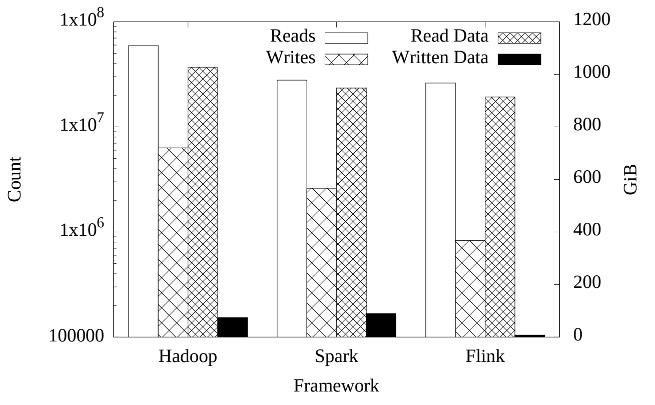

Fig. 17. Aggregated Lustre file system counters for Join on 14 nodes for Hadoop, Spark and Flink

achieve local sorting of partitions and then merging them into the final output, as the list of resulting files indicates. It is likely that this sort with a parallelism of one, independent of number of worker nodes, is the source of Flink's performance.[10]

In an attempt to clarify the runtime differences between the frameworks, we present file system counters for the SQL benchmark run on 14 nodes using Lustre, as Lustre provides all counters necessary. The results can be seen in Figure 17. Even though we have observed Spark to issue many operations to the file system, especially reads, we observe that Hadoop issues an order of magnitude more reads and writes than Spark in the SQL benchmark. This indicates many shuffle phases, which however appear to filter data early on, as Hadoop reads a total of 1025 GiB, just 146 GiB more than the input data set, and Spark reads 60 GiB more than the input data set. Flink reads just 36 GiB more.

Hadoop writes 70 GiB more than the output data set, Spark writes 85 GiB more and Flink writes just 4 GiB more. For Hadoop the overhead is explained by shuffle data, Spark reports a total of 11 GiB of shuffle data, however produces not just one result file, but one for each partition (56 in this case), each locally sorted, which are then merged into the result file. It is these partitions that make up the additional data. For Flink, again, we cannot be sure when the additional data has been read or written.

From these statistics it is clear why Hadoop performs worse than Spark and Flink, however as to why Flink performs much worse than Spark, we still cannot be certain, except that we now know it is not additional I/O that slows Flink down. In fact, seeing this little amount of data written indicates that the result indeed is processed largely on one node, and Flinks performance therefore is limited by the processing power of that one node. We will investigate how Flink's Table API has evolved since the release we are using in future experiments.

[10]It is worth noting that an `orderBy` has been added to the Table API in Flink 1.1.0: https://issues.apache.org/jira/browse/FLINK-2946. However, we could not incorporate results from this API because of time limitations.

V. Conclusion & Future Work

The performance of the three data analytics frameworks Hadoop, Spark and Flink was evaluated on a small Cray TDS with DataWarp and Lustre as filesystem backends using selected benchmarks with Big Data Analytics characteristics.

The configuration of the frameworks were not streamlined for specific benchmarks, but instead we determined working configuration parameters using the versatile TeraSort benchmark, and used these configurations for the other benchmarks as well.

The overall performance experiences show that for our small setup with at most 16 nodes, the DataWarp based file system (Stage 1) is in no case beneficial for the selected benchmarks TeraSort, Streaming, and an SQL benchmark compared to Lustre as storage backend, and displays worse scaling behavior. A recent study by D. Bard et al. [15] revealed that reading and writing small files (or small I/O transfers) to DataWarp is problematic in some cases and that generally in many cases the DataWarp performance is worse than the Lustre file system, the latter also being due to the client-side caching in Lustre, which is not yet available in DataWarp. Thus, this observation is in line with ours where a typical I/O data block is at most 64 KiB small.

Spark and Flink can take advantage of larger memory configurations per node. With our configuration of 32 GiB per node we see our benchmark results as cases where the file system backends are more stressed than would be necessary for larger memory configurations, as the frameworks have to spill to disk more often.

Profiling these data analytics workflows becomes challenging. The Hadoop framework provides a rich set of internal counters, which however are not always exhaustive. Spark and Flink provide fewer counters and metrics at the moment, most of them being only accessible conveniently via a provided web frontend. This is why we used counters provided by DVS and Lustre to assess actual I/O performance. An important area of future investigation is the reconciliation of read and write rates observed in the frameworks with the ones from IOR for similar I/O patterns. For profiling tasks using DVS, additional counters providing the total number of bytes read and written would be helpful. Support for read/write memory mapped files would be beneficial for certain frameworks as well.

For application performance profiling, the Cray performance analysis tools cannot be used out-of-the box due to the missing Java support. Here, a Cray Java version might be beneficial for the identification of bottlenecks and simplified instrumentation.

For the interpretation of performance results, details of the DVS software layer might better clarify the observations. For example, the degree of parallelism in the I/O queue handled by the DVS layer would be interesting in the context of scalability tests.

This study has still a work-in-progress character due to the late availability of the DataWarp software stack, and the still evolving state of the frameworks used, especially Flink and, to a certain extent, Spark as well. There are still some open questions for which we seek solutions together with the Cray experts within our joint project, including correctly collecting and interpreting DVS statistics.

For future work, the impact of different stripe sizes other than 8 MiB on the DataWarp might be reveal better settings for the chosen analytics workloads.

Acknowledgment

This work was partly funded by the German Federal Ministry of Education and Research (BMBF) with the two Big Data projects BBDC (grant number 01IS14013B) and GeoMultiSens (grant number 01IS14010C). The work is also part of our collaboration with Cray, Inc., on technology evaluation.

References

[1] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] S. Sugiyama and D. Wallace, “Cray DVS: Data Virtualization Service,” in *Cray User Group*, 2008.

[3] S. Andersson, S. Sachs, C. Tuma, and T. Schütt, “DataWarp: First Experiences,” in *Cray User Group*, 2015.

[4] (2016) The Apache Hadoop website. [Online]. Available: http://hadoop.apache.org/

[5] (2016) The Apache Spark website. [Online]. Available: http://spark.apache.org/

[6] (2016) The Apache Flink website. [Online]. Available: http://flink.apache.org/

[7] (2016) The HiBench microbenchmark suite. [Online]. Available: https://github.com/intel-hadoop/HiBench

[8] (2016) Fork of the HiBench microbenchmark suite. [Online]. Available: https://github.com/robert-schmidtke/HiBench/tree/custom

[9] (2016) Fork of the TeraSort benchmark. [Online]. Available: https://github.com/robert-schmidtke/terasort

[10] (2015) Apache Hadoop TeraSort. [Online]. Available: https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/examples/terasort/package-summary.html

[11] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 165–178.

[12] (2014) TeraSort benchmark comparison for YARN. [Online]. Available: https://www.mapr.com/sites/default/files/terasort-comparison-yarn.pdf

[13] (2013) Cisco UCS demonstrates leading TeraSort benchmark performance. [Online]. Available: http://unleashingit.com/docs/B13/Cisco\%20UCS/le_tera.pdf

[14] (2008) Apache Hadoop wins TeraSort benchmark. [Online]. Available: https://developer.yahoo.com/blogs/hadoop/apache-hadoop-wins-terabyte-sort-benchmark-408.html

[15] (2016) Cori Phase 1 Burst Buffer. [Online]. Available: {http://www.nersc.gov/assets/NUG-2016-business-day/Burst-Buffer.pdf}