The PigMix Benchmark on Pig, MapReduce, and HPCC Systems

Keren Ouaknine[1], Michael Carey[2], Scott Kirkpatrick[1]
{ouaknine}@cs.huji.ac.il
[1]School of Engineering and Computer Science, The Hebrew University of Jerusalem
[2]Donald Bren School of Information and Computer Sciences, University of California, Irvine

***Abstract*—Soon after Google published MapReduce, their paradigm for processing large amounts of data, the open-source world followed with the Hadoop ecosystem. Later on, LexisNexis, the company behind the world's largest database of legal documents, open-sourced its Big Data processing platform, called the High-Performance Computing Cluster (HPCC). This paper makes three contributions. First, we describe our additions and improvements to the PigMix benchmark, the set of queries originally written for Apache Pig, and the porting of PigMix to HPCC. Second, we compare the performance of queries written in Pig, Java MapReduce, and ECL. Last, we draw conclusions and issue recommendations for future system benchmarks and large-scale data-processing platforms.**

***Index Terms*—PigMix, HPCC Systems, MapReduce, Benchmark, Big Data, Performance**

I. Introduction

Data production and consumption has dramatically increased in sectors ranging from healthcare to advertising, business intelligence, and social sciences. To address this demand, several large-scale data processing platforms have emerged. The most well-known one is Hadoop, inspired by Google's MapReduce programming model [1]. It has become popular very quickly and large companies such as Twitter, Facebook, and Linkedin are currently using it as their processing platforms to gain insights from their data. Another open-source data-processing platform is the High Performance Computing Cluster (HPCC) from LexisNexis, the company behind the largest legal database. The development of HPCC started around 2000, and was open-sourced in 2012. On top of these processing platforms, there is a need to abstract the programming model to allow analysts and data scientists to focus on analyzing data rather than writing and debugging complex code. To address this need, Big Data platforms have each adopted query languages. For Hadoop, a popular language is Pig Latin (commonly abbreviated as Pig), and for HPCC, it is the Enterprise Control Language (ECL). Queries are transparently compiled into jobs by a compiler: Pig for Hadoop, and ECLCCServer for HPCC. But which platform performs best? For which tasks? Can they be compared rigorously? PigMix [2] is a benchmark released by Yahoo! and Hortonworks, the main contributors to Pig. PigMix has been used in studies ranging from cloud computing [3] to user engagement modeling [4] and security [5]. Therefore, we have ported PigMix to ECL to run the benchmark queries on HPCC Systems. As shown in Figure 1, this paper uses PigMix to compare three data-processing approaches: 1) Hadoop MapReduce jobs handwritten in Java [1], 2) Pig Latin scripts, compiled into Java MapReduce jobs for Hadoop, and 3) ECL scripts, compiled into C++ jobs for HPCC.

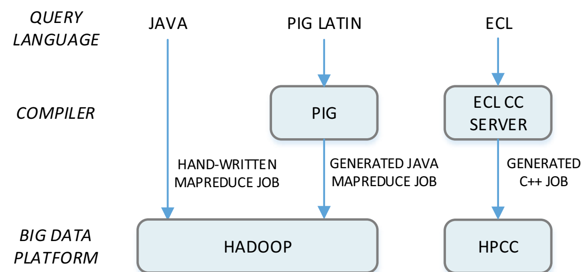

Fig. 1. Comparison of handwritten Java MapReduce (left), Pig scripts (middle), and ECL scripts compiled into C++ jobs for HPCC (right).

In the reminder of this paper, we provide some context for our study, detail our contributions to the PigMix benchmark, and provide performance results from running the benchmark at several scales with the three approaches of interest, and conclude with lessons learned.

II. Study Context

In this section, we first introduce Hadoop and MapReduce, then explain how Pig scripts run on Hadoop. We continue by describing the HPCC platform, and finish with an overview of the PigMix benchmark.

[1]The hand-written queries are intended to represent a reasonable form of coding of the queries in Java MapReduce, assuming a basic level of MapReduce programming expertise [2].

978-1-4673-7278-7/15 $31.00 © 2015 IEEE
DOI 10.1109/BigDataCongress.2015.99

IEEE computer society

A. Hadoop

Hadoop is designed for large-scale analytics on a distributed cluster of commodity computers. One of its major advantages is its ability to scale to thousands of nodes. It is designed to efficiently distribute and handle large amounts of work across a set of machines. The Hadoop Distributed File System (HDFS) splits data into smaller parts (i.e blocks) and distributes each part redundantly (by default) across multiple nodes. A MapReduce job [1] processes the blocks by running tasks on the cluster nodes in parallel. The input and the output of the job are stored on HDFS. Among other things, Hadoop takes care of the scheduling, monitoring and re-execution of failed tasks [6]. Hundreds of companies rely on Hadoop to store and analyze their data sources. To name a few: EBay, Hulu, Riot Games, Twitter, etc

B. Pig

Apache Pig is used to express and process complex MapReduce transformations using a simple declarative query language [2] [7]. Pig Latin (the query language) defines a set of operations on sets of tuples, such as aggregate, join and sort. Via a query compiler, Pig translates the Pig Latin script into MapReduce so that it can be executed on Hadoop. Pig has thousands of users and contributors. Among the most prominent ones are Netflix, LinkedIn, and Twitter. Yahoo! uses Pig to express more than 60% of all its Hadoop jobs [8]. Pig jobs are compiled into one or more MapReduce job(s), each reading data from and writing data to HDFS. Pig shares a few similarities with a traditional RDBMS: a query parser, a type checker, an optimizer, and operators (inspired by the relational algebra) that perform the data processing. The parser transforms a Pig Latin script into a logical plan. Semantic checks and optimizations [9] are applied to the logical plan, which is then optimized to a physical plan. The physical plan is then compiled into a MapReduce plan, i.e, a chain of MapReduce jobs, by the MapReduce compiler. This MapReduce plan is then further optimized (combiners are used where possible, jobs that scan the same input data are combined, etc). Finally MapReduce jobs are generated by the job control compiler. These are submitted to Hadoop and monitored by the MapReduce launcher.

C. HPCC with ECL

From a high level, the HPCC architecture [10] appears similar to that of Hadoop. They both run on a shared nothing cluster and aim to use a software infrastructure layer to coordinate the processing of Big Data in parallel across commodity nodes with local storage. Many of the components of Hadoop have corresponding components in HPCC, although there is not always a one-to-one mapping. ECL is the native operational language of the HPCC platform. It is also a declarative programming language and operates at a similar level of abstraction to Pig Latin. Work is submitted to the HPCC system by sending ECL programs to a component called the ECL-CC server, which generates C++ code for each logical operation (e.g. sort) that the HPCC engine will execute. This is then compiled down to an executable. That executable is then passed to an ECL agent, that is responsible for the work flow for the particular job [11]. The ECL agent cooperates with the ECL System to schedule the execution of the job across one or more Thor processing clusters. The Thor processing cluster is structured to have a master and some number of slaves. The executable contains code for each node to execute, and also contains code to allow the master to coordinate the execution of the slaves. Thor is the ”data refinery” cluster and is responsible for all ETL operations. The ECL optimizer is a key component of the platform. The optimizer selects the most efficient components that produce the end result the user requested; In general, this will not be the same as what the user specified. The optimizer partitions the execution of the job graph to manage the consumption of machine resources at each stage. In HPCC, data structures are datasets of records (much like tuples in Pig) which may contain child datasets nested to any level. One difference from Pig's data model is that the Pig map has no equivalent data structure in ECL (hence the issue described later with Q1 querying on a map).

D. Main differences

Babu and Herodotou have offered a list of criteria to compare massively parallel databases [12]. We use some of these criteria in Table I to highlight the key conceptual differences between Hadoop and HPCC. The design of Hadoop is oriented to provide reliability at the tradeoff of performance. Intermediate results are written to disk between each mapper and reducer and to HDFS between one MapReduce job and another in a multi-job query. Therefore, when several MapReduce jobs are chained to execute a query, we can expect a slowdown in the execution. This allows Hadoop jobs (and Pig) to tolerate failures. In contrast, HPCC will fail an entire job upon a task failure unless the user explicitly requests a checkpoint. Hadoop also implements some functionality that speeds up its jobs, such as speculation. Speculation runs ”forks” of the same task on several nodes and returns the first one to complete. This can shorten long execution tails due to a slow machine (but not due to

	Hadoop	**HPCC**
Processing paradigm	MapReduce programming model	Flexible
Task uniformity	The same map/reduce task on all nodes	Tasks can vary from one node to another
Internode communication	Never directly (i.e., always file based)	Sometimes
Data partitioning	Only between the mapper and reducer phase	At any stage
Reliability	Configurable through HDFS	One or two replicas
Fault tolerance	High granularity at the task level	Entire job fails w/o predefined checkpoints

Table I: Conceptual differences between Hadoop and HPCC.

skew). Also, Hadoop can reschedule a task on another machine transparently.

E. *PigMix*

Recently a few efforts in the area of big data benchmarks emerged [13], [14]. The PigMix benchmark [15] was developed to test and track the performance of the Pig query processor from version to version. It was released in 2009 by Hortonworks. It consists of 17 queries operating on 8 tables testing a variety of operators and features such as sorts, aggregations, and joins. The benchmark includes, in addition to the Pig Latin queries, their equivalents in Java to run directly as MapReduce jobs as a baseline to test the performance gap with Pig. The Pig and MapReduce versions of all PigMix queries are part of the Apache Pig codebase. The equivalent queries for HPCC, in ECL, can be found in [15]. The 8 tables and respective schemas of PigMix model logs from users requesting Yahoo! pages. The main table, called page_views, contains fields such as the query terms that brought the user to the page or the time that the user spent on the page. The actual data of PigMix is randomly generated using a generator that is part of the codebase for Pig. One can vary the amount of data to generate by specifying the number of page_views rows. Our input sizes started at 10 million rows, i.e 16 GB, which is the scale at which Hortonworks experiments with the benchmark. We then scaled page_views up to 270 million rows, which is 432 GB, partitioned over ten nodes. Full details on all eight tables can be found on the PigMix webpage [16].

III. Benchmark contributions

In the course of setting up, running, and analyzing PigMix in our environment, we ended up making a set of changes and contributions to PigMix itself. We summarize each of them below.

A. *Correctness patch for the Java MapReduce queries*

As we first began to compare the results of the queries in Java and Pig when running PigMix on our cluster, we noticed that their outputs were different. Half of the handwritten Java queries were actually incorrect (Q5-10, Q12, Q15). For example, some queries were joining on a wrong field, or wrongly tokenized the records, hence outputting wrong results. (We found this to be surprising.) We fixed these queries to output correct results and submitted a patch that is now part of Apache Pig master branch (JIRA 3915). Interestingly (and luckily for the Pig community), our contributions related to the correctness of these queries had little impact on the performance of the queries.

B. *Performance patch for the Java MapReduce queries*

1) Overuse of memory: Upon scaling the input sizes for the PigMix experiments to several times the available main memory, some of the MapReduce queries failed due to out of memory errors (Q3, Q9-10, Q13-14). The queries were naively buffering the two inputs for the joins into two Java List objects prior joining them. Rather, a join on two large tables should be implemented as without buffering the lists and as described in the MapReduce literature [17]. From then on, the queries that were previously failing completed successfully.

2) Optimization of join queries: The PigMix benchmark has four large joins, and each of these joins in MapReduce was originally implemented as a chain of three MapReduce jobs: a separate job for reading each of the branches (left and right inputs), and a third job chained to the previous two for executing the actual join operator using an identity mapper. There is a simpler way to express a join, with one MapReduce job rather than three. Chaining MapReduce jobs has significant cost implications, i.e., writing the outputs to HDFS is prolonging the execution of the queries. The improved join implementation reads both inputs in the map phase and joins them in the reduce phase [17]. Figure 2 shows the execution time of the four PigMix joins on 270 million rows before and after the code change. This optimization is part of our patch JIRA-4004.

3) Update APIs: Following the changes above, the MapReduce queries were outputting the right results and

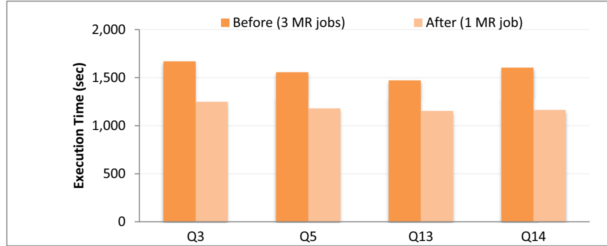

Fig. 2. Optimization of the hand written MapReduce join queries.

executing successfully at scale. However, they had been left unchanged since their publication in 2009, and the *mapred* API used in the Java queries is now deprecated. Pig's compiler is using the new API called *mapreduce*. It seemed to us a fairer comparison to update the MapReduce queries as well. This significantly reduced and simplified the code of the MapReduce queries (by about half the lines). The patch, which has significant performance contributions, has not been integrated yet into the main Apache Pig branch, but we used it in our experiments, as the changes were necessary to successfully collect results at scale. Also, as the PigMix benchmark has been the basis for research work and numerous publications in the past few years, it seemed important to publish results for our scaled, optimized, and up-to-date MapReduce queries as the comparison set for the Pig versions.

C. Adapting ECL

The HPCC team converted the PigMix data to CSV format and ported the PigMix queries to ECL using an internal tool for the translation [18]. However, nested data cannot be represented in CSV format, so Q1 could not be ported to HPCC Systems this way, and only 16 of the 17 queries were compared [19]. To fix this, we parsed the PigMix-generated input data to XML (another format HPCC can read) and rewrote the queries to handle the XML input. The execution time for XML was then 14 hours (!) to merely count the XML records, as is described in HPCC track system online, issue HPCC-12504. To solve this issue, we read the XML input files but then converted them to HPCC's internal format. From then on, we could successfully run all 17 queries on all systems. Additionally, the scale at which the previous experiments were conducted at HPCC was too small to be conclusive. Therefore we used larger storage in our experiments to scale-up to 270 page_views million rows.

IV. Software and Hardware Configuration

The experiments reported in this paper were conducted on a cluster of 10 dual-socket, dual-core AMD Opteron servers. Each server has 8 GB of memory and dual 1 TB 7200 RPM SATA disks. The servers were all connected via 1 GB Ethernet on the same switch. We used Pig version 0.13.1 and Hadoop version 1.2.1. For HPCC, we used the community version 5.0.2-1. During the experiments phase for Hadoop, we ran two concurrent mappers and reducers per node. We set the HDFS block size to 256MB.

V. Performance results

We now present the results of running PigMix on the aforementioned systems. We define the **multiplier** for two systems as the ratio of their running times.

A. Default benchmark

The scale factor used by Yahoo! and Hortonworks is 10 million rows of page_views (16GB). Over their ten-node cluster, this is 1.6GB of input data per machine. At this scale, all tuples fit in-memory. Since it is clearly difficult to reach conclusions relevant to Big Data if the input size of the experiments is that small, we opted in section V-B to scale the input size to a factor of several times the memory size of the nodes to better obtain better Big Data results.

1) Pig execution time compared to Java MapReduce on 10 million rows: Similar to the results published by Hortonworks [16], the multipliers in Figure 3 range mostly between 0.8 and 1.2 with an average multiplier of 1.09. Therefore, the execution times for Pig and MapReduce is close on small input sizes. The only query which has a low multiplier compared to the others is Q12. This query splits page_views into three subsets and writes them to a file. The filter condition for the three outputs has a common conditional factor. Pig detects that factor and calculates the common subset only once, whereas MapReduce calculates the common subset three times.

2) HPCC execution time compared to Pig on 10 million rows: HPCC ran faster than Pig on nearly all queries for the 10 million row input size, except for the sort queries, as shown in Figure 4. In HPCC, prior to a sort operator, the master node collects statistics from the slaves and predicts an optimal partitioning for the sort operation [20]. Each node then redistributes data from its data portion that falls outside its assignment to the appropriate nodes. Upon receipt of this data, each node can perform a merge sort to add the received data to the previously sorted data portion. If the data is skewed, so is the processing and the time for each

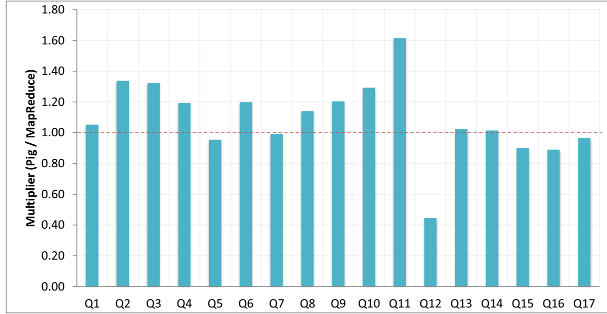

Fig. 3. The multiplier comparing Pig to MapReduce. A multiplier lower than 1.0 means that Pig completed before MapReduce.

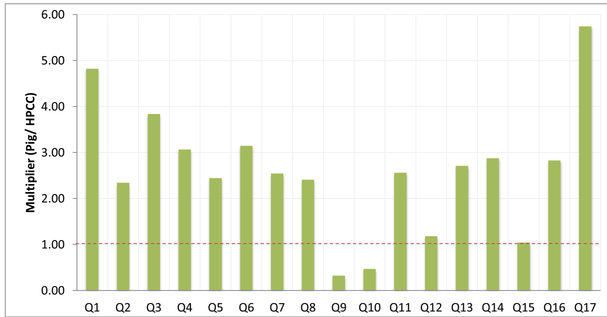

Fig. 4. The multiplier comparing HPCC to Pig. A multiplier higher than 1.0 means that HPCC completed faster than Pig.

partition to process, impacting the overall time, so this is an important optimization. For Pig, the optimization for the sort operator also balances the output across reducers. It does this by first sampling the input of the order statement to get an estimate of the key distribution. For that, it requires an additional MapReduce job to do the sampling; this takes generally less than 5% of the total job time. Based on this sample, it then builds a partitioner that produces a balanced total order. Unlike HPCC, a value can be sent to several reducers in Pig, which allows the partitioner to distribute the key more evenly. This appears to be a reason for Pig's superiority on queries Q9-10.

B. Scaling to Big Data

In this section, we describe PigMix query results on experiments scaling up to four times the overall memory size, i.e., 432 GB for the main table page views.

1) HPCC multiplier decreases as the input size increases: In the previous section, on small input sizes, HPCC ran faster on most queries. However, upon sizing-up our experiments, we observed the opposite trend. Size-up experiments are used to increase the input size while keeping the same hardware [21]; one can then observe how well a system operates with a higher data load. In Figure 5 we compare HPCC and Pig on 10

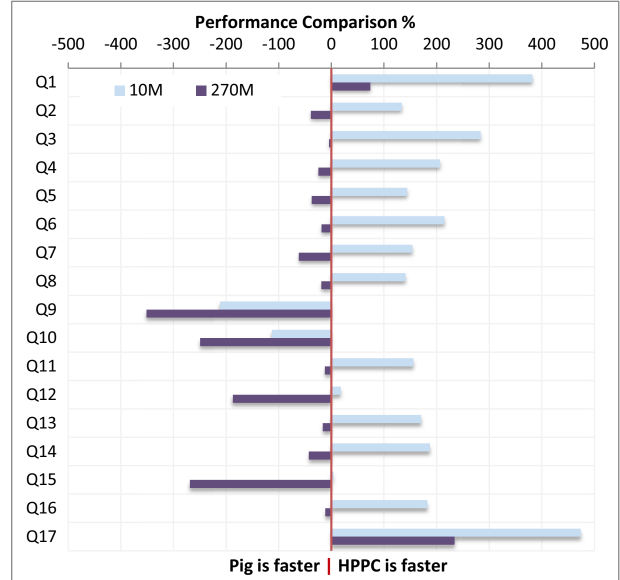

Fig. 5. Comparison of performance across queries between Pig and HPCC. A performance comparison of -200% means that HPCC is two times slower than Pig.

million and 270 million page_views rows. On the right hand side of the graph, the 10 million row queries executed faster on HPCC (as discussed in the previous section). On the left hand side, Pig completed faster than HPCC for most queries. In other words, scaling up reduces the advantage of HPCC running time as we increased the input size. (This is also consistent with experiments that we ran on intermediate sizes.) Space precludes detailed discussion of performance differences and their reasons. Full experimental results are available online[2].

VI. Lessons Learned

A. Indexing

Besides Thor, HPCC also has a real-time data delivery engine (Roxie) that uses indexing to be able to process thousands of requests per node per second. We also tried Roxie, but it did not improve the running time of HPCC over the Pigmix benchmark, as the queries are converted into full scans anyway due to low selectivity predicates for the PigMix select and join operators. For future work, it would be interesting to add to the Pigmix queries a set of more selective queries on indexed data and compare the execution times.

B. Hardware configuration

It is important to compare how different systems take advantage of the available hardware resources. In a first

[2]http://www.kereno.com/pigmix

round of experiments, we compared the three systems on a ten-node cluster where each node had multiple directly accessible disks. As the cluster was also used for Hadoop/HDFS, the disks were mounted as JBOD; i.e., they were mounted as separate Linux volumes and were not utilizing any sort of abstraction between the OS and disk besides the file system itself (i.e., no RAID or LVM). For many Big Data systems, this is one of their acceptable hardware configurations, with this type of configuration being geared towards newer systems that take on the tasks of replication and failover in software. However, for HPCC Systems, one must have one (and only one) location allocated for the data per node, homogeneously across the entire cluster [22]. Using RAID was not an option for us, as the cluster's hardware and OS were shared with other users. Therefore we had to limit our experiments to use only one disk for a fair comparison of Hadoop and HPCC.

C. System tuning

As mentioned in [23], it is crucial to fine tune each systems' configuration settings to improve the performance. The open source community blogs provide useful information for these tasks. Additionally, one should observe the distribution of the files over the cluster to make sure there are no anomalies at the file system level, as data distribution is critical for performance.

D. Dataset size matters

Pig and HPCC consumed the same input records and produced exactly the same output, but the execution flow of these two systems and their components are intrinsically different. The small size jobs on Pig definitely suffered from the overhead to launch them due to JVM setup, shipping jars to execute, setup the distributed cache, etc. Another slow-down is due to the fact that reducers don't start before mappers are globally finished. For certain tasks such as filtering or distinct, this processing paradigm can slow down execution. At the larger size, however, we observed that Pig outperformed HPCC on most queries, i.e., these overheads were no longer significant.

VII. Conclusions and Future Work

We ran the PigMix benchmark on Pig and MapReduce, and in the process we contributed to its correctness, scaling, and performance. These additions are now part of the Pig master branch, or available on JIRA as a patch to the Pig codebase. Additionally, we ported the PigMix benchmark to HPCC Systems and compared the execution time on various sizes. Small input sizes showed an advantage for Pig over MapReduce and for HPCC over Pig. However, as we increased the input sizes, the performance differences reduced. The Hadoop startup overheads encountered on small sizes were nullified when running at scale, i.e., with an input of four times the memory size. The average multiplier for Pig and MapReduce at scale was very close, 0.98. For future work, it would be interesting to experiment with additional queries such as shorter (more selective) queries, or multi-join queries needing multiple MapReduce jobs and intermediate outputs. Additionally, it would be interesting to port the benchmark to other Big Data systems and to the cloud.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[2] Gates, A., et al. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *VLDB Endowment*.

[3] J. Ortiz, V. De Almeida, and M. Balazinska. A vision for Personalized Service Level Agreements in the Cloud. In *2nd Workshop on Data Analytics in the Cloud*. ACM, 2013.

[4] M. Lalmas. User Engagement in the Digital World. *Disponible en*, 2012.

[5] James Stephen, J., et al. Program analysis for secure big data processing. In *ACM/IEEE Conf. on Automated Software Eng.*, 2014.

[6] The Apache Mapreduce Tutorial. http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.

[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. In *ACM SIGMOD Conference on Management of Data*, 2008.

[8] Christopher Olston. Web Scale Data Processing. http://infolab.stanford.edu/~olston/pig.pdf, 2009.

[9] A. Gates, J. Dai, and T. Nair. Apache Pig's Optimizer. *IEEE Data Eng. Bull.*, 2013.

[10] HPCC Systems website. http://hpccsystems.com/why-hpcc.

[11] Bayliss, D., et al. Method and System for Parallel Processing of Database Queries, 2005. US Patent 6,968,335.

[12] S. Babu and H. Herodotou. Massively Parallel Databases and MapReduce Systems. *Foundations and Trends in Databases*, 2013.

[13] Ghazal, A., et al. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *2013 ACM SIGMOD inter. Conf. on Management of Data*.

[14] T Graves. Graysort and Minutesort at Yahoo! on Hadoop 0.23. *http://sortbenchmark. org/Yahoo2013Sort. pdf*, 2013.

[15] Queries in Pig and ECL. http://hpccsystems.com/Why-HPCC/HPCC-vs-Hadoop/pigmix_ecl.

[16] Apache Pigmix website. http://cwiki.apache.org/confluence/display/PIG/PigMix.

[17] White T. *Hadoop The Definitive Guide*. O'Reilly, 2012.

[18] Bacon. http://hpccsystems.com/bb/viewtopic.php?f=10&t=111.

[19] Nested data. http://hpccsystems.com/Why-HPCC/HPCC-vs-Hadoop/pigmix_ecl#L1.

[20] Bayliss, D., et al. Method for Sorting and Distributing Data among a Plurality of Nodes, 2007. US Patent 7,293,024.

[21] D. DeWitt, J. Naughton, and D. Schneider. Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting. In *Parallel and Distributed Information Systems*, 1991.

[22] https://track.hpccsystems.com/browse/HPCC-12356.

[23] Michael J Carey. BDMS Performance Evaluation: Practices, Pitfalls, and Possibilities. In *Selected Topics in Performance Evaluation and Benchmarking*. Springer, 2013.