SparkTune: tuning Spark SQL through query cost modeling

Enrico Gallinucci
DISI - University of Bologna
Bologna, Italy
enrico.gallinucci@unibo.it

Matteo Golfarelli
DISI - University of Bologna
Bologna, Italy
matteo.golfarelli@unibo.it

ABSTRACT

We demonstrate SparkTune, a tool that supports the evaluation and tuning of Spark SQL workloads from multiple perspectives. Unlike Spark SQL's optimizer, which mainly relies on a rule-based model, SparkTune adopts a cost-based model for SQL queries; this enables the accurate estimation of execution times and the identification of cost and complexity factors in a user-defined workload. The estimate is based on the cluster configuration, the database statistics (both automatically retrieved by the tool) and the resources allocated to the workload. Thus, for any given cluster, database and workload, SparkTune is able to identify the best cluster configuration to run the workload, to estimate the price to run it on a cloud platform while evaluating the performance/price trade-off, and more. SparkTune turns the cluster tuning efforts from manual and qualitative to automatic, optimized and quantitative.

1 INTRODUCTION

Over the past years, Apache Hadoop has become the most popular framework for Big Data handling and analysis. On top of it, *SQL-on-Hadoop* solutions have been introduced to provide a relational abstraction layer to the data. Among them, one of the most popular is Apache Spark, whose SQL-based sub-system (i.e., Spark SQL [1]) enables SQL queries to be rewritten in terms of Spark commands and to be executed in parallel on a cluster[1].

Although these systems are largely adopted and quickly becoming more solid and mature, they are still limited in terms of cost modeling features. For instance, the module in charge of translating SQL queries to Spark commands (i.e., Catalyst [1]) mainly relies on a rule-based optimizer. The cost model introduced to Catalyst in its 2.3.0 release is still very simple (e.g., as concerns join ordering, the cost function estimates a logical cost in terms of number of returned rows). The need for robust cost-based models is increasing, since the possibility to evaluate a priori the execution cost of a workload is still lacking. Very little work has been done on this aspect: some optimizers propose cost-based features, but they evaluate only portions of a query [8].

Our system, *SparkTune*, is an application that builds on a cost model [3] for Spark SQL to support the user understanding the cost of a workload and gaining the required knowledge to properly optimize the execution. From a high-level perspective, SparkTune allows to: i) evaluate the duration of a workload; ii) see the cluster's size and potential first-hand, so as to properly optimize the allocation of resources; iii) evaluate the trade-off between the execution time and the price to execute it on a cloud platform. The latter point is achieved by extending the cost model with the pricing information of major cloud providers, which enables the translation of resource consumption into actual money. On a more technical side, SparkTune also: iv) enables the prior identification of stragglers (i.e., tasks that performs more poorly than similar ones due to insufficient assigned resources); v) if adopted by Spark SQL, it would enable the creation of a full cost-based optimizer, both static and dynamic.

In Section 2 we summarize the core aspects of the cost model (full details have been published in [3]), while Section 3 discusses the features of SparkTune; the demonstration proposal is finally given in Section 4.

2 COST MODEL OVERVIEW

The Spark architecture consists of a *driver* and a set of *executors*. The driver negotiates resources with the cluster resource manager (e.g. YARN) and distributes the computation across the executors, which are in charge of carrying out the operations on data. Data are organized in *Resilient Distributed Datasets* (RDDs), i.e., collections of immutable and distributed elements *partitions* that can be processed in parallel. Partitions can either come from a storage (e.g. HDFS) or be the result of a previous operation (i.e., held in memory). At the highest level of abstraction, a Spark computation is organized in *jobs*, which are composed of simpler logical units of execution called *stages*. The physical unit of work used to carry out a stage on each RDD partition is called *task*. Tasks are distributed over the cluster and executed in parallel.

In Spark SQL, a declarative SQL query is translated in a set of jobs by its optimizer, Catalyst, which carries out the typical optimization steps: analysis and validation, logical optimization, physical optimization, and code generation. Physical optimization creates one or more *physical plans* and then it selects the best one. At the time of writing, this phase is mainly rule-based: it exploits a simple cost function only for choosing among the available join algorithms [1].

Our cost model [3] computes the query execution time given the physical plan provided by Catalyst. We remark that it is *not* a cost-based optimizer, as the latter implements query plan transformations to optimize the original plan, and it does so without necessarily computing the whole cost of the query.

The cost model covers a wide class of queries that composes three basic SQL operators: selection, join and generalized projection. The combination of these three operators determine GPSJ (Generalized Projection / Selection / Join) queries, which were first studied in [5] and which are the most common class of queries in OLAP applications. Each Spark physical plan modeling a GPSJ query can be represented as a tree whose nodes represent tasks; each task applies operations to one or more input tables, either physical or resulting from the operations carried out in its sub-tree. Our cost model relies on a limited number of *task types* (listed in Table 1) that, properly composed, form a GPSJ query. In particular, a feasible tree properly composes the following task types: *table scan* SC(), *table scan and broadcast* SB(), *shuffle join* SJ(), *broadcast join* BJ() and *group by* GB(). SC() and SB() are always leaf nodes of the execution tree since they deal with the

[1]Spark is not necessarily tied to Hadoop, although this is its most used architecture.

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

physical storage where the relational tables lie. SJ() and BJ() are inner nodes of the trees and can be composed to create left-deep execution trees; finally GB(), if present, is the latest task to be carried out.

The execution time is obtained by summing up the time needed to execute the tasks of the tree coding the physical plan of a query. In particular, the cost model is based on the disk access time and on the network time spent to transmit the data across the cluster nodes; CPU times for data serialization/deserialization and compression are implicitly counted by the disk throughput. This is consistent with [9], which clearly explains that *one-pass* workloads on Spark (e.g., SQL queries) are either network-bound or disk-bound, whereas CPU can become a bottleneck limitedly to serialization and compression costs. Also, the cost model assumes that data always fits the executors memory, so that data is never spilled to local disks.

Depending on its type, each task involves one or more basic operations (such as reading, writing, and shuffling) which we refer to as *basic bricks*; the cost of a task is calculated as the sum of the cost of the involved bricks (see 1 for the usage of bricks by the task types). In particular, each brick models the execution of an operation on a single RDD partition and considers the resource contentions given by parallel execution. Bricks are SQL-agnostic and require some parameters about Spark (e.g., network and disk read/write throughput) and about the cluster (e.g., number of executors per rack and number of cores per executor) to be known. As explained in Section 3, most of these parameters are automatically retrieved by SparkTune. In the following, we briefly discuss the nature of each basic brick.

- Read: consists in reading an RDD partition from disk. If the data does not reside on the executor's node, it must be read from another one, according to the locality principle. The cost model exploits the known cluster configuration to estimate the probability of such a case. When reading from another node, both the time to transfer the data over the network must be considered in addition to the time to read the data from disk. Since Spark enables in-memory pipelining of subsequent transformations that do not require shuffling (according to the Volcano model [4]), the overall time is computed as the maximum for disk reading and data transmission.
- Write: consists in writing an RDD partition to the local disk; no network data transfer is necessary.
- Shuffle read: consists in reading an RDD partition from disk and shuffling it through the network. Similarly to the Read brick, the overall time is computed as the maximum for disk reading and data transmission; in this case, however, disk reading only happens on the local disk.
- Broadcast: consists in loading the whole RDD on the application driver and in sending it to every executor. Since the two operations cannot be pipelined, the overall cost is determined as the sum of the two. The broadcast brick does not involve disk reading or writing, thus the cost only depends on network time.

Ultimately, we remark that – thanks to the known cluster configuration – the cost model is able to probabilistically estimate the amount of data to be read and/or transferred through the network for each brick.

Example 2.1. The following GPSJ query is taken from the TPC-H benchmark [7]; it computes the total income collected in a given period and for a specific market segment grouped by single orders and priority of shipping.

```
SELECT l_orderkey, o_orderdate, o_shippriority,sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND
c_custkey = o_custkey AND l_orderkey = o_orderkey AND
o_orderdate < date '1995-03-15' AND
l_shipdate > date '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
```

A graphical representation of the Spark physical plan chosen by Catalyst is reported in Figure 1.

Table 1: Task types characterization

Task Type	Additional params	Basic bricks
SC()	pred, cols, groups	Read, Write
SJ()	pred, cols, groups	Shuffle Read, Write
SB()	pred, cols	Read, Broadcast
BJ()	pred, cols, groups	Write
GB()	pred, cols, groups	Shuffle Read, Write

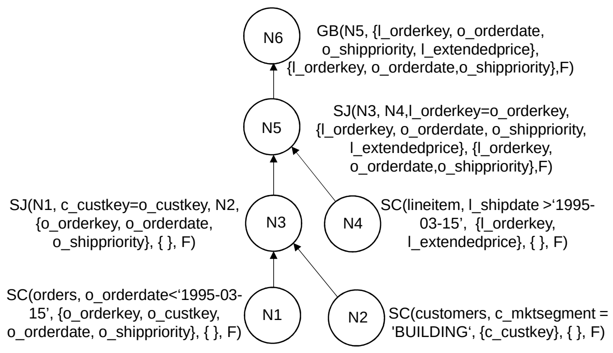

N6
GB(N5, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
N5
SJ(N3, N4,l_orderkey=o_orderkey, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
SJ(N1, c_custkey=o_custkey, N2, {o_orderkey, o_orderdate, o_shippriority}, { }, F)
N3
N4
SC(lineitem, l_shipdate >'1995-03-15', {l_orderkey, l_extendedprice}, { }, F)
SC(orders, o_orderdate<'1995-03-15', {o_orderkey, o_custkey, o_orderdate, o_shippriority}, { }, F)
N1
N2
SC(customers, c_mktsegment = 'BUILDING', {c_custkey}, { }, F)

Figure 1: GPSJ grammar derivation for the query in Example 2.1; node names substitute sub-expressions in the inner nodes

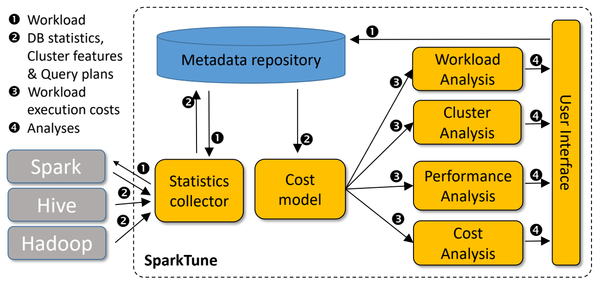


Figure 2: SparkTune's architecture and data flow

3 THE SYSTEM

SparkTune is implemented as a web application on a classic WAMP stack (Windows, Apache, MySQL, PHP) and it is publicly available at http://semantic.csr.unibo.it/sparktune. We plan to release SparkTune as a standalone application in the near future. The architecture of the system is sketched in Figure 2, where the flow of data is also indicated.

3.1 Environment setup

The system enables registered user to setup their own environment in order to define custom scenarios and run user-specific analyses. The setup of the environment can be done either manually or in an automatic fashion. The information required by the system is the following:

- Cluster configuration, such as the number of nodes and racks, the number of cores per node, HDFS's replication factor, hardware statistics. Automatic retrieval of these data is enabled by providing credentials for an SSH connection to one of the cluster's nodes. Most parameters are obtained through calls to the Hadoop's APIs, while Spark jobs are run to infer the read and write throughput of the disk, as well as the intra-rack and inter-rack network throughput. Both throughput values are inferred as a function of the number of concurrent processes that require disk access and network data transfer.
- Database statistics. Automatic retrieval of these data is enabled by providing credentials to access the Hive Metastore. For any of the available database, the user can trigger the retrieval of the name, size, and statistics for every table and attribute.
- One or more workloads, meant as sequences SQL queries to be manually provided by the user; then, the system automatically retrieves from Spark the physical plan.

These data are stored in the Metadata repository. Each user can setup multiple clusters, databases, and workloads. The versions of Spark currently supported are 1.5 and 2.2.

3.2 Analyses

The system provides four kinds of analysis and simulation. Each of them requires to define a specific environment (i.e., to select a cluster, a database, and workload among the ones provided in the setup), possibly requests additional parameters (e.g., the number of executors to be allocated for each node), and outputs a detailed report depending on the kind of analysis. Figure 3 shows examples of such reports, which are fully discussed in the following.

Workload analysis. The goal is to provide an in-depth analysis of the complexity and cost of executing the workload on a cluster with a specific configuration (i.e., the number of executors and the number of cores per executor must be defined by the user). First of all, SparkTune runs each query's physical plan in the workload on the cost model to estimate the total execution time of the workload. Secondly, it provides full details about every task of every query (including the amount of data read/written, the time to read/write data from disk or to transfer it across the network). On the one hand, this greatly helps the identification of stragglers: Figure 3.1 shows the report of a single query, which presents the tasks in the execution tree colored on a red gradient (the harder the red, the higher the percentage of time required by the task w.r.t. to the query). On the other hand, it helps understanding which factors mainly impact on the cost of each task/query: below the execution plan in Figure 3.1, the total estimated time of the query is split among disk read time, disk write time, and network time; then, by clicking on a task in the tree, a pop-up as the one in Figure 3.2 shows the same data for each task — together with other detailed information (e.g., the amount of data read/written/transferred, the cardinality of the involved table(s), etc.).

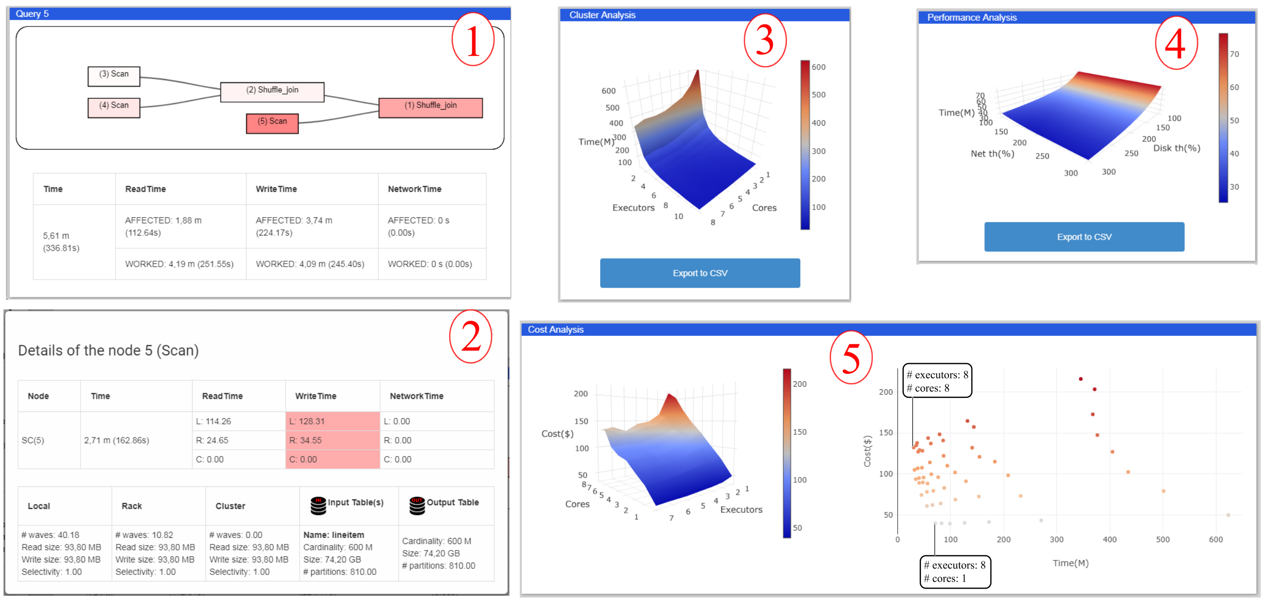
Cluster analysis. The goal is to optimize the allocation of resources by understanding the impact of the configuration parameters on the performance. The total execution time of the workload is calculated for every configuration potentially available within the boundaries set in the cluster setup[2]. The results are presented in a 3D surface graph (Figure 3.3), showing the execution time by the number of executors and the number of cores. This graph emphasizes how the increase of executors and cores progressively reduces the execution time; more specifically, it shows which of the two factors has the greatest impact in the time reduction for the given workload.

Performance analysis. The goal is to provide a what-if analysis that shows the potential impact of enhancing the performance of the cluster (in terms of disk throughput and network throughput) on the execution time of a workload. Given a specific cluster and its configuration, the system estimates the execution time by progressively improving the disk and/or network throughput in steps of 20% (i.e., 120% w.r.t. to the current throughput, 140%, etc.). The results are presented in a 3D surface graph (Figure 3.4), which helps understanding which factor (disk or network) is most critical in determining the execution time. Noticeably, the increase in disk throughput is assumed in the same measure for both reading and writing.

Cost analysis. The goal is to evaluate the price of executing a workload on a cloud infrastructure. To make such an estimate, we extended our cost model with the pricing information obtained from cloud providers Amazon AWS (http://aws.amazon.com) and Google Cloud (http://cloud.google.com); this enables the translation of the time usage estimates of every core and every executor to the amount of USD (United States Dollars) to be paid to the provider for the execution of the workload. Similarly to the Cluster analysis, the system exhaustively calculates the total execution time of the workload in every possible configuration (within the upper bounds set in the cluster setup), but it is eventually translated to USD. Ultimately, two results are presented. The first one (left-hand side of Figure 3.5) is a mere cost analysis, showing the price obtained with the different cluster configurations by means of a 3D surface graph; this enables the identification of the cheapest cluster configuration, as well as the understanding of which factor (i.e., number of executors or number of costs) is the most expensive. The second one (right-hand side of Figure 3.5) is an evaluation of the trade-off between execution time and price; it is shown as a 2D chart, pinpointing each cluster configuration in the 2D surface represented by the execution time and the price. This chart allows an immediate identification of the ideal cluster configurations, which minimize both the time spent and the money to be paid. For the sake of completeness, we note that the price estimate only considers the consumption of the cluster's resources for the estimated time; it does not consider the price of the disk, which is required to store the database.

Example 3.1. The reports in Figure 3 pertain to a workload of 8 queries from the TPC-H benchmark on our 11-node cluster. Workload analysis. Figure 3.1 represents the execution tree of the query in Example 2.1 and shows that the overall time (5.61 min) is mostly due to two tasks (nodes 1 and 5). Given Spark's in-memory pipelining, the "affected time" voice indicates the portion of the time spent reading/writing/transferring data (i.e.,

[2]We adopt this naive approach for the purpose of showing a full report to the user. A cost-based optimizer that only looks for an optimal configuration could apply some heuristics to avoid an extensive research [2].



Time	Read Time	Write Time	Network Time
5,61 m (336.81s)	AFFECTED: 1,88 m (112.64s)	AFFECTED: 3,74 m (224.17s)	AFFECTED: 0 s (0.00s)
	WORKED: 4,19 m (251.55s)	WORKED: 4,09 m (245.40s)	WORKED: 0 s (0.00s)

Details of the node 5 (Scan)

Node	Time	Read Time	Write Time	Network Time
SC(5)	2,71 m (162.86s)	L: 114.26	L: 128.31	L: 0.00
		R: 24.65	R: 34.55	R: 0.00
		C: 0.00	C: 0.00	C: 0.00

Local	Rack	Cluster	Input Table(s)	Output Table
# waves: 40.18 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	# waves: 10.82 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	# waves: 0.00 Read size: 93,80 MB Write size: 93,80 MB Selectivity: 1.00	**Name: lineitem** Cardinality: 600 M Size: 74,20 GB # partitions: 810.00	Cardinality: 600 M Size: 74,20 GB # partitions: 810.00

Figure 3: Screenshots of the various functionalities of SparkTune, discussed in Section 3

"worked time") that actually contributed to determining the overall query time. Figure 3.2 shows the details of node 5 in Figure 3.1; it is the scan of the Lineitem table and it shows that the task is mostly disk-bound. Cluster analysis. Figure 3.3 clearly shows that, for the given workload, the performance gain due to increasing the number of executors is superior to the one due to increasing the number of cores per executor. Performance analysis. Figure 3.4 further confirms that the workload is disk-bound; indeed, improving the network throughput would have little to no impact on the execution time, as opposed to the adoption of more performing disks. Cost analysis. The prices shown in Figure 3.5 refer to to Amazon AWS; interestingly, the left chart shows that the cheapest configurations are those with a low number of cores. This indicates that the higher core-power would not be adequately put to use, as it does not correspond to a significant reduction in time (as the Cluster analysis also anticipated). Ultimately, this is confirmed by the right chart, which shows a high disproportion between the price required to reduce the execution time of the cheapest configured and the time actually saved.

4 DEMO PROPOSAL

In the demonstration we will show how cluster tuning can be automatized and optimized adopting a cost-based approach. First, we will demonstrate the automatic retrieval of data for the environment setup, either on our own cluster or on a cluster owned by someone in the audience (provided their willingness to grant its access). Then, we will develop with the audience an experience to showcase the various functionalities of the system and to understand their value from the perspective of different professional figures. In particular, two distinct scenarios will be simulated for data scientists (which will be interested in analyzing workloads from a more technical angle) and data architects (which will be interested in an analyzing the same workloads from a different, higher-level perspective). With data scientists we will mainly focus on SparkTune's Workload and Cluster analyses to answers several questions, such as: *"What are the reasons for the poor performance of this workload?"*, or *"How can I reallocate the resources to speed up the execution of the workload without exceeding my budget?"*. Differently, with data architects we will mainly focus on SparkTune's Cost and Performance analyses to address issues related to the design and deployment of the infrastructure; in particular, we will answer questions such as *"Which cloud provider will let me run this workload at the lowest price and at a reasonable performance?"*, or *"What kind of scale-up or scale-out improvements can I bring to the cluster to improve its performance?"*. The demo will put the audience in the shoes of one of these figures and we will develop a simulation aimed at answering the aforementioned questions.

Our real cluster that we use as a reference consists of 11 nodes with 8 cores per node. SparkTune will be configured with different benchmark databases, including the well-known TPC-H [7] and the Big Data Benchmark [6]. We will define different workloads on each database, in order to present scenarios that require the exploitation of the different features of SparkTune.

REFERENCES

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. 1383–1394.

[2] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems* 28, 5 (2012), 755–768.

[3] M. Golfarelli and L. Baldacci. 2018. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge & Data Engineering* (2018), 14. https://doi.org/10.1109/TKDE.2018.2850339

[4] Goetz Graefe and William J McKenna. 1993. The Volcano optimizer generator: Extensibility and efficient search. In *Data Engineering*. 209–218.

[5] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-Query Processing in Data Warehousing Environments. In *VLDB*. 358–369.

[6] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *SIGMOD*. 165–178.

[7] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[8] John Pullokkaran. [n. d.]. Introducing cost based optimizer to apache hive. https://cwiki.apache.org/confluence/download/attachments/ 27362075/CBO-2.pdf.. Online; accessed 18 dEC. 2017.

[9] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB* 8, 13 (2015), 2110–2121.