A Cost Model for SPARK SQL

Lorenzo Baldacci, Matteo Golfarelli DISI - University of Bologna

Abstract—In this paper we propose a novel cost model for Spark SQL. The cost model covers the class of Generalized Projection, Selection, Join (GPSJ) queries. The cost model keeps into account the network and IO costs as well as the most relevant CPU costs. The execution cost is computed starting from a physical plan produced by Spark. The set of operations adopted by Spark when executing a GPSJ query are analytically modeled based on the cluster and application parameters, together with a set of database statistics. Experimental results carried out on three benchmarks and on two clusters of different sizes and with different computation features show that our model can estimate the actual execution time with about the 20% of errors on the average. Such an accuracy is good enough to let the system choose the most effective plan even when the execution time differences are limited. The error can be reduced to 14%, if the analytic model is coupled with our straggler handling strategy.

Index Terms—Spark, Spark SQL, Cost Model, Query Optimization.

1 Introduction

Big Data imposed a change of paradigm in the way data are handled and analyzed. The volume of data to be processed, as well as their variety, pushed the development of a broad set of solutions and platforms. In this context, Apache Hadoop is the framework that, more than any other, has gained great popularity over the past years. The first version of Hadoop was strictly limited to the MapReduce programming paradigm. MapReduce requires skilled programmers, so Hadoop designers tried to relieve their users by providing more high-level and easy-to-use programming environments such as Pig and Hive – the first SQL system based on Hadoop and MapReduce.

Spark is a fast and general purpose computing engine for large-scale data processing that can run on Hadoop. It builds on the MapReduce paradigm by enabling pipelining of transformations. In this way it reduces the number of times data are written back to the disk and can be orders of magnitude faster than MapReduce [22] while preserving fault tolerance features. Spark can process data stored in several storage engines (e.g. MongoDB, Cassandra). In this work we focus on Hadoop (and HDFS) that is, by far, the most used architecture for Spark. In particular, Spark includes an SQL-based sub-system: standard SQL queries are rewritten in terms of Spark commands and executed in parallel on a cluster. Spark SQL performance allow on-line computations on big data, and many OLAP vendors such as Tableau and Micro Strategy already provide connections to their systems.

Although Spark is already largely adopted, it is still under development and continuously evolving. The Spark SQL engine cannot be considered as mature as traditional relational DBMSs and many performance improvements are still possible. In particular, Catalyst [3], the module that is in charge of translating an SQL query in a sequence of Spark commands, still relies on a rule-based optimizer and very little work has been done to develop a cost model that can predict the cost of running an SQL query.

In this paper we propose a cost model for Spark SQL which covers the class of GPSJ (Generalized Projection / Selection / Join) queries that were first studied in [9]. A GPSJ query is composed of joins, selection predicates and aggregations. Since it is not mandatory that all the three operators are present, therefore our cost model covers also simple selection and join queries. Although other cost models have been developed for generic MapReduce applications, to the best of our knowledge, this is the first result that keeps into account the Spark computation paradigm and Spark SQL. More in details, the distinguishing features of our cost model are the following:

- It relies on a limited number of task types that, properly composed, model a wide family of SQL queries. Since the behavior of these task types is known, their cost model can be analytically shaped, thus providing accurate estimates.
- It does not require complex job profiles aimed at capturing a generic job behavior (e.g. [11]), but rather it bases its estimate on a small set of the cluster and application features, coupled with the DBs statistics.
- The cost function models an SQL execution plan in terms of Spark costs. This enables an SQL-aware evaluation of the execution costs. We believe that moving the cost evaluation to a more *conceptual level* may help SQL-users to better understand the system behavior.
- The returned cost is not a logical one, but rather the actual execution time also keeping the cluster features into account. This enables cluster performance analysis and cluster tuning to be supported.

Although the cost model is built specifically for Spark (in terms of task types and system behavior), it is generalizable to other big data SQL engines such as HIVE and Impala. The novelty of our approach goes beyond the boundaries of the Spark SQL world; it represents an original approach to the calculation of the cost of executing SQL queries on the Big Data platforms.

The execution time is obtained by summing up the time needed to execute the nodes of the tree coding the physical plan produced by Catalyst. The cost model is based on the disk access time and on the network time spent to transmit the data across the cluster nodes. We also consider CPU times for data serialization/deserialization and compression. These costs are implicitly counted by the disk throughput. As explained in [22], Spark workloads that are mainly *one-pass*[1], like the one produced by SQL queries, are either network-bound or disk-bound, whereas CPU can become a bottleneck with regard to serialization and compression costs. This is not true for *multi-pass*[2] workloads that are typically CPU-bound. Experimental results on clusters of different sizes and with different computation features show that our model can estimate the actual execution time with an error of 20% in average. Such accuracy is good enough to let the system choose the most effective plan even when the execution time differences are limited. The error can be reduced to 14%, if the analytic model is coupled with a straggler handling strategy.

The cost model can be adopted in several contexts that range from estimating the query execution time given the cluster and application setting, to obtaining a clear understanding of query performances under different settings. These are key information in workload management, capacity planning, and system tuning. Our cost model is also a first step towards turning Catalyst into a fully[3] cost-based optimizer and to compare the execution cost of different physical plans even when adaptive execution is considered.

The main contributions of the paper are: (1) the analytic cost model that covers GPSJ query expressiveness; (2) a large set of tests (overall, we run more than 1000 queries) that analyze its accuracy from several different points of view. Tests have been carried out on 3 benchmarks and 2 clusters.

The paper outline is as follows: Section 2 reports the related literature; Section 3 provides a background on Spark and describes how GPSJ queries are executed on Spark; the cost model is defined in Sections 4 and 5; Section 6 reports the experiments; in Section 7, the conclusions are drawn.

2 Related Work

The literature about cost models for predicting the performance of SQL engines dates back to the mid-70s, when the first query optimizers were developed. In [21] the authors propose a cost model for the relational DBMS System R: filtering, projection and join operations were considered. The costs were computed in terms of disk pages fetched and a set of basic assumptions on the predicate selectivities was made. Effectiveness of cost models for centralized SQL engines depends on the quantity of statistics available about data. For example, moving from basic attribute and table cardinality to histograms [15] allows to improve the accuracy of selectivity predicates in presence of skewed distributed data. When passing from centralized to distributed DBMSs [13] the communication time must also be taken into account. The topology of the network plays a central role in defining the weight of the different cost components: whereas transmission costs are typically predominant [20] in a WAN, a cost model devised for a LAN must consider local access costs, too. A further step forward came with parallel DBMSs that required the cost models to keep resource contention and data dependencies into account [7].

The number of SQL engines for big data platforms is continuously growing. Hive has been the first solution providing a SQL-like query language, called HiveQL [24]. Hive compiles HiveQL queries into a series of MapReduce jobs that imply high latency. To address this issue, different directions have been followed: Tez can run Directed Acyclyc Graphs (DAGs) as a single job, reducing the latency in launching jobs; Presto, uses a traditional MPP DBMS runtime instead of MapReduce. Spark SQL relies on the Catalyst optimizer to turn SQL queries into optimized DAGs that can be executed on the Spark general purpose engine, which performs much faster than MapReduce. The Catalyst versions we analyzed in this paper are mainly-rule based; Catalyst exploits statistics and a simple cost model since its 2.3.0 release. For example, as concern join ordering, the cost function estimates a logical cost in terms of number of returned rows.

We emphasize that we do not propose a *cost-based optimizer* but rather a *cost model*. A cost-based optimizer does not necessarily compute the whole cost of a query neither in absolute nor in relative terms (e.g. query plan 1 is better than query plan 2). Furthermore, a cost-based optimizer implements query plan transformations (typically based on rules) that exploit the cost of specific query portions (e.g. a join) to make choices and to create an optimized plan. Conversely, our cost model computes the absolute query cost (in secs.) given the physical plan provided by Catalyst (i.e. the Spark query optimizer). In Section 6.5 we prove that our cost model returns an absolute cost that is accurate enough to choose the best execution plan among the feasible ones. Closing the loop and changing the actual Spark plan is out of the scope of the paper. With reference to Hive [17] the main difference between our cost-model and the costs computed by the Hive optimizer are:

- Hive does not compute the whole query cost but rather the cost of the specific portion it is optimizing.
- Hive costs are computed on the physical operator tree disregarding the cluster configuration. Consequently. the optimizer choices will be the same for different clusters and for executions with different resources.
- Hive makes simpler assumptions concerning Read and Write disk throughputs that are fixed and do not vary depending on the processes concurring on the disk.
- Hive optimizer supports a larger set of SQL operators such as UNION ALL and OUTER JOIN that cannot be modeled as a GPSJ.

As to similarities, we adopt the same set of Hive statistics. In particular, Hive considers table cardinalities and attribute cardinalities, while it does not consider attribute value distributions. The Impala and Spark optimizers make similar assumptions, apart from Spark that in its 2018 version (ver. 2.3.0) started collecting histograms [18]. Differently from Spark, far more numerous are the research efforts and results related to cost models for MapReduce [12]. In [25] the authors propose an analytical model for MapReduce that keeps into account, besides the task execution time,

1. In a one-pass application data are read from the disk, processed and written back to the disk (e.g. word-count, sort).

2. In a multi-pass application algorithms iterate over data several times before writing them back to the disk (e.g. K-means and Page-rank algorithms).

3. Catalyst, the Spark optimizer, already makes some cost-based choices when choosing the join algorithms.

the delays due to parallel execution: *queuing* delays due to contention at shared resources and *synchronization* delays due to precedence constraints among tasks. Herodotou [11] proposed a performance model for describing the execution of a MapReduce job in Hadoop 1.x. The performance model describes data flow and cost information at the finer granularity of phases within the map and reduce tasks. In terms of Herodotou's model, the overall job execution time is simply the sum of the costs from all map and reduce phases, no considerations about synchronization and queuing delays are provided. Similarly to the previous one, our model considers the single execution phases at a high level of detail and also keeps queuing and synchronization delays into account by considering pipeling and resource contention. Many differences arise with the Spark SQL context: with reference to [25] the precedence graph between tasks comes from the query's physical execution tree provided in input and it has to be analyzed in the light of how Spark implements SQL operators. Considering that Spark uses a finite and known set of task types to execute GPSJ queries and that the DB statistics are available, we are able to provide a detailed modeling of them. Differently from [11], we do not need a general job profile to characterize the task types. This is because, instead of general MapReduce jobs, we are modeling SQL queries having available a detailed description of the query execution plans, plus the DB statistics that allow us to analytically derive such parameters.

3 Background and Spark Basics

Spark [2], [27] is a parallel computational framework compatible with Hadoop. The Spark architecture consists of a *driver* and in a set of *executors*. The driver negotiates resources with the cluster resource manager (e.g. YARN) and distributes the computation across the executors. The executors are in charge of carrying out the operations on data. Data are distributed across the cluster and are organized in *Resilient Distributed Datasets* (RDDs). A RDD is a collection of immutable and distributed elements, named *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). Spark provides a rich set of operators to manipulate RDDs. Operators can be classified in *Trasformations* and *Actions*. Transformations can be carried out in-memory on each RDD partition, whereas actions either return a result to the driver or imply a shuffling to combine data distributed in several RDD partitions. Besides RDDs, Spark introduces the concept of *lazy evaluation* of operations too, i.e., Spark does not compute the operation results right away, but it keeps track of the sequence of operations to be applied to a RDD until an *action* is triggered. The Spark computation paradigm overcomes the one based in MapReduce by enabling in-memory pipelining (according to the *Volcano model* [8]) of subsequent transformations that do not require shuffling.

At the highest level of abstraction a Spark computation is organized in jobs: a *job* is created when an action is requested over a RDD; it is composed by simpler units of execution called *stages*, linked together through a directed acyclic graph. A stage is still a logical unit of work as it is composed by a pipeline of transformations to be applied

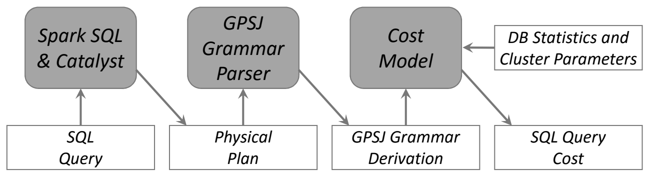


Fig. 1. A description of the process for our cost model. The grey blocks are system modules, whereas white ones are input/output data.

to an input RDD. 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.

Spark SQL [3], is the Spark module that enables the execution of SQL queries on the structured data stored in Hadoop. It provides a relational abstraction layer over data and a SQL-like language to query it. The core of Spark SQL is Catalyst [3], an extensible optimizer which is in charge of translating a declarative SQL query into a set of jobs. Given a SQL query, Catalyst 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 in order to select a join algorithm among the available ones [3]. Spark join algorithms deserve a special mention because of the importance they have in GPSJ queries:

- Broadcast join: can be used only when one of the two tables to be joined fits in memory. In this case the smaller one is broadcasted to every executor so that each task can perform the join between a partition of the bigger table and the broadcasted data which is available in memory.
- Shuffle join: both tables are sorted/hashed on the joining attributes and then split in the same number of chunks. Chunks are saved on the local disk of the executors. When shuffling occurs, all the chunks with the same range of attributes and from the two different tables are shipped to a reducer task which verifies the join predicate and saves the filtered data back to the local disk.

Our cost model computes the query execution time for a given Spark physical plan (see Figure 1). The cost model covers a wide class of queries by composing three basic SQL operators: selection, join and generalized projection. The combination of these three operators determines GPSJ (Generalized Projection / Selection / Join) queries that were first studied in [9]. A GPSJ is a generalized projection π over a selection σ over a set of joins χ: $\pi\sigma\chi$. The generalized projection operator, $\pi_{P,M}(R)$, is an extension of duplicate eliminating projection, where P denotes an aggregation pattern on a relation R, i.e., the set of group-by attributes, and M denotes a set of aggregate operators applied to the attributes in R. Thus, GPSJ expressions extend select-join expressions with aggregation, grouping and group selection. GPSJ queries are the most common class of queries in OLAP applications. It is not mandatory that all the three operators are present, thus our cost model also covers simple selection queries and join queries.

```
<GPSJ>::=<Expr> | <GB(<Expr>)>
<Expr>  ::=<SJ(<Expr>,<Expr1>,F)> | <Expr1> |
            <BJ(<Expr2>,<Expr3>,F)>
<Expr1> ::=<SC(<Table>,F)>
<Expr2> ::=<SB(<Table>)>
<Expr3> ::=<SC(<Table>,T)> | <SJ(<Expr>,<Expr1>,T)>
            | <BJ(<Expr2>,<Expr3>,T)>
<Table> ::={pipe-separeted set of database tables}
```

Fig. 2. Backus-Naur representation for the GPSJ grammar.

TABLE 1
Task types characterization.

Task Type	Addittional 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

Each Spark physical plan modeling a GPSJ query can be represented as a tree whose nodes apply operations to one or more input tables, either physical or resulting from the operations carried out in its sub-tree. The feasible trees are coded by the context free grammar represented in Figure 2, that we call *GPSJ grammar*. A feasible tree (i.e., a GPSJ grammar derivation) properly composes the following 5 *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 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 last task to be carried out. For the sake of clarity, in Figure 2 we omitted the parameters that characterize the task types, but do not impact on the tree structures. Such parameters are listed in Table 1: $pred$ is an optional filtering/join predicate, $cols$ is the subset of columns to be actually retrieved, $groups$ identifies a group-by set of a generalized projection operation. SC(), SJ(), and BJ() can carry out grouping as Catalyst may push GB() down for optimization purpose. The grammar defines the task execution pipeline. In particular, as shown in Figure 2, the last parameter for SC(), SJ(), and BJ() is a boolean named $pipe$ (see Section 5). When $pipe$ is set to true the task type does not write back data to the disk, since the parent task type can be carried out in pipeline. Task type pipelining[4] can only be exploited before a broadcast join, since all the other task types either correspond to a leaf node of the tree (i.e., SC() and SB()) or require a shuffle (i.e., GB() and SJ()). In the GPSJ grammar, such constraint is ensured by the usage of <Expr3> that appears only as a SB() parameter.

Example 1 *The following GPSJ query is taken from the TPC-H benchmark [16], one of the benchmarks we used for testing. The query 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
```

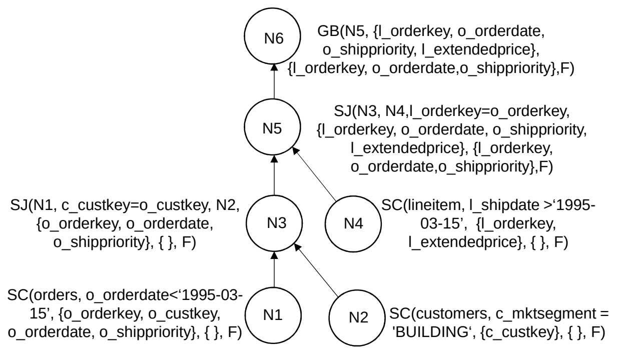

N6
GB(N5, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
N5
SJ(N3, N4,l_orderkey=o_orderkey, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
SJ(N1, c_custkey=o_custkey, N2, {o_orderkey, o_orderdate, 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)

Fig. 3. GPSJ grammar derivation for query in Example 1. Node names substitute sub-expressions in the inner nodes.

A graphical representation of the Spark physical plan chosen by Catalyst is reported in Figure 3. The tree is a GPSJ grammar derivation. Each tree node corresponds to a task type applied to the relational table(s) obtained by the node sub-trees.

In the selected physical plan, orders *and* customer *(i.e.,* N1,N2*) are initially fetched from HDFS through a* SC() *task type that also filters and projects the tuples in main memory before shuffle-writing them to the executor's disks.* N3 *performs a shuffle join of the two tables. The same join type is used in* N5 *where the result from* N3 *is further joined with* lineitem *that has been retrieved from HDFS, filtered and projected in* N4*. Each RDD partition resulting from* N5 *shuffle join is also (locally) grouped by* l_orderkey, o_orderdate, o_shipriority *to reduce the cost of the last operation (i.e., group-by push down reduces the quantity of data to be shuffled).The generalized projection (i.e.,* GROUP BY*) is finalized in node* N6*.*

We finally emphasize that the trees coding the physical plans of a GPSJ query involving n tables have at most $2 \cdot n$ nodes. n nodes (either SC() or SB()) are needed to access the tables; $n-1$ nodes (either SJ() or BJ()) are needed to join the tables; finally a GB() node may be used in case of GROUP BY clause. Since only left-deep trees are produced by Spark, the depth of trees is at most $n+1$. Left-deep trees generate the deepest physical plans; indeed, for balanced trees the depth decreases to $\lceil log_2(n) \rceil + 1$.

4 Cost Model Basic Bricks

Precisely modeling Actions and Transformations would lead to an useless complexity for our model that is based on network and disk access costs. For this reason, we focus on a set of *basic bricks* that determine such type of costs and are used within GPSJ queries computations. Noticeably, the level of abstraction of such bricks is even lower than that of Trasformations and Actions. Each brick models the execution of an operation on a single RDD partition, still considering the resource contentions given by parallel execution. Basic bricks are SQL-agnostic and require just the

4. Transformations pipelining and task types pipelining are not synonyms: the first term refers to pipelining single Spark transformations, the second one refers to pipelining of coarser unit of work. A single task type may include several Spark transformations.

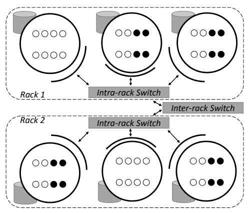


Fig. 4. A two-racks cluster abstraction. Circles denote executors, black dots represent cores involved in the computation, white ones are idle.

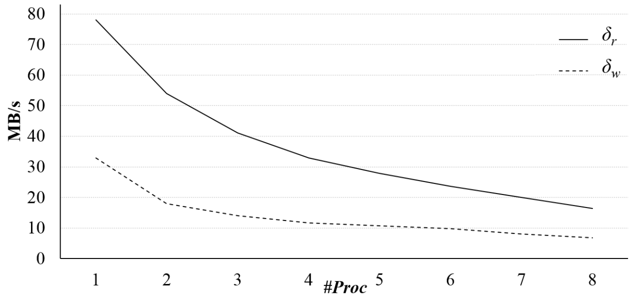


Fig. 5. Read $\delta_r()$ and write $\delta_w()$ throughput varying the number of processes concurring on the disk.

Spark and cluster parameters (see Table 2) to be known. Sub-sections 4.2 to 4.5 characterize the execution cost for each brick and define a cost function for them.

4.1 Cluster Abstraction and Cost Model Parameters

With reference to Figure 4 a cluster is composed by $\#N$ nodes evenly distributed on $\#R$ racks. Each node mounts $\#C$ cores. All the racks/nodes are assumed to be equal in terms of HW features. Data are stored on HDFS file system in blocks with redundancy factor rf ($rf = 3$ by default). The disk throughput plays a central role in precisely computing the execution time. It is modelled it through two functions: $\delta_r(\#Proc)$ and $\delta_w(\#Proc)$, that return the *per-process* throughput in MB/s. The functions values are obtained through a tuning test that must be carried out on each cluster. In particular, we consider the total time needed for loading data from the disk and making them available in a RDD for further processing, so that the disk throughput implicitly incorporates CPU time for serialization and decompression. Figure 5 shows the values of the two functions for one of the two clusters we used for testing. Read/Write throughput decreases as the number of processes reading/writing data increase due to contention of the (shared) disk resource.

Cluster nodes are connected through a network; we adopt a point-to-point network model with a bandwidth limit for each connection. Similarly to disk behavior, the network throughput depends on the number of processes that are concurrently transmitting between a couple of nodes. It is a fair assumption that the intra-rack network speed ($IntraRSpeed$) is higher or equal to the inter-rack one ($ExtraRSpeed$). The formulae for the network *per-process* throughput are:

$$\rho_i(\#Proc) = \frac{IntraRSpeed}{\#Proc}$$

TABLE 2
Cost model parameters and basic functions. Horizontal lines split those related to cluster, application and data respectively.

Parameter	Description
$\#R$	Number of racks composing the cluster.
$\#RN$	Number of nodes in each rack.
$\#N$	Overall number of nodes (i.e., $\#R \cdot \#RN$).
$\#C$	Number of cores available on each node.
rf	Redundancy factor for HDFS.
$\delta_r(\#Proc)$	Disk read throughput (in MB/sec) as a function of the number of concurrent processes.
$\delta_w(\#Proc)$	Disk WRITE throughput (in MB/sec) as a function of the number of concurrent processes.
$\rho_i(\#Proc)$	Network throughput (in MB/sec) between nodes in the same rack as a function of the number of concurrent processes.
$\rho_e(\#Proc)$	Network throughput (in MB/sec) between nodes in different racks as a function of the number of concurrent processes.
$\#SB$	Number of buckets used for shuffling.
$sCmp$	Percentage of data reduction due to compression when transmitting on the network.
$fCmp$	Average size reduction achieved by a compressed file format.
$hSel$	Constant selectivity for HAVING clauses.
$\#RE$	Number of executors allocated to the Spark application in each rack.
$\#E$	Overall number of executors allocated to the Spark application (i.e., $\#RE \cdot \#R$).
$\#EC$	Number of cores for each executor allocated to Spark application.
$t.Attr$	Set of attributes in table t.
$t.Size$	Size (in MB) of table t stored in a uncompressed file format.
$t.PSize$	Average size (in MB) of RDD partitions for table t.
$t.Card$	Number of tuples in table t.
$t.Part$	Number of partitions table t is composed of.
$a.Card$	Number of distinct values for attribute a.
$a.Len$	Average length (in byte) of attribute a .

$$\rho_e(\#Proc) = \frac{ExtraRSpeed}{\#Proc}$$

Please note that network and disk performances must be computed at the node level rather than at the core one, since they exploit resources that are contended among all the cores in the same node.

Each Spark application running on a cluster has its own set of resources and parameters. In our model we assume that, once assigned, the resources cannot be modified during the execution. The two main resources to be defined are the number of executors ($\#E$) and the number of cores ($\#EC$) on each executor. Each application has an application driver that runs on a cluster node different from the executor ones. Due to its relevance in estimating shuffle execution time, we also consider the number of shuffle buckets $\#SB$ ($\#SB = 200$ by default). We finally assume that each shuffle bucket fits the executor's memory when read, so that data are never spilled to local disks.

Since the number of RDD partitions is typically higher than the number of cores available for the computation, the resource manager schedules the tasks on the cores in multiple *waves*.

Definition 1 (Wave) *A* wave *identifies the parallel execution of a set of tasks of the same type, one by each core of the the application executors. Each task processes a distinct RDD partition. All the executions in the same wave are considered to behave similarly.*

It is apparent that since all the executor cores work in parallel during a wave and behave similarly, the time needed to run a wave can be estimated as the time needed to run a task on a single core.

4.2 Read

Since Spark applies the data locality principles, it always loads RDD partitions from the "closest" position. Reading time varies if the executor reads the RDD partition from the local disk, from the disk of a node belonging to the same rack or from a different one. $Read(Size, X)$ computes the reading time of a RDD partition of a given $Size$ and depending on the place X where data are stored. $X \in \{L, R, C\}$ stands for Local, Rack and Cluster. On the one hand, if data are read locally, no transmission through the network is needed; on the other hand, for Rack and Cluster fetching, the time for transmitting must be considered. Since disk reading and data transmission happen in pipeline, the overall time is the maximum of the two components:

$$Read(Size, X) = \mathrm{MAX}(ReadT_X; TransT_X)$$

In a local wave the RDD partitions are read from the local disks and no data are transmitted over the network (i.e., $TransT_L = 0$).

$$ReadT_L = \frac{Size}{\delta_r(\#EC)}$$

$ReadT_L$ is the time needed by a core to read a RDD partition of size $Size$ from the local disk whose per-process throughput is $\delta_r(\#EC)$. $ReadT_L$ keeps into account disk resource contention due to the $\#EC$ cores hosted on the same executor.

During a rack wave each executor core receives a RDD partition from a node of its rack that does not host an executor (i.e., $\#RN - \#RE$ nodes). The disks of the executors are not involved, since, if a copy of the requested RDD partition was stored on them, it would be read during a local wave in accordance with the data locality principle. If all nodes host an executor, all waves are local. We model the probability of a wave to be local/rack/cluster in Section 5.2.

Each of the non-executing nodes serves $\lceil \frac{\#RE \cdot \#EC}{(\#RN - \#RE)} \rceil$ requests on average[5], where $\#RE \cdot \#EC$ is the number of processes/cores working in parallel in a rack. Consequently:

$$ReadT_R = \frac{Size}{\delta_r(\lceil \frac{\#RE \cdot \#EC}{\#RN - \#RE} \rceil)}$$

In rack and cluster waves, network time must be taken into account. In particular, given an executor running a rack wave, each of its $\#EC$ cores receives a RDD partition from one of the $\#RN - \#RE$ non-executing nodes in the rack. Assuming a uniform distribution of RDD partitions across the rack nodes, the number of cores sharing the same node-to-node network connection can be bounded by $\lceil \frac{\#EC}{\#RN - \#RE} \rceil$. Thus, the time needed to transmit data through each single network connection is:

$$TransT_R = \frac{Size}{\rho_i(\lceil \frac{\#EC}{\#RN - \#RE} \rceil)}$$

5. Ceiling is necessary since, apart from assuming workload uniform distribution, a RDD partition is entirely read from one node.

Example 2 *In the cluster shown in Figure 4, $\#R = 2$, $\#RN = 3$, $\#RE = 2$ and $\#EC = 4$. During a rack wave, each core reads a RDD partition within its rack, precisely from the only node with no executors allocated. Such node is therefore in charge of fetching 8 partitions from its disk. The per-process disk throughput is:*

$$\delta_r(\lceil \frac{2 \cdot 4}{3 - 2} \rceil) = \delta_r(8)$$

We recall that, as per the network model discussed in Section 4.1, nodes are linked together through point-to-point connections, each ensuring a given bandwith limit. The node with no executors allocated, once having read the 8 RDD partitions, transmits the data to the 2 executors allocated in its rack. The per-process network throughput is then given by:

$$\rho_i(\lceil \frac{4}{3 - 2} \rceil) = \rho_i(4)$$

A similar modeling can be applied to cluster waves where each node not hosting an executor can potentially receive a request from all the cores of the executors belonging to a different rack (i.e., $(\#R-1) \cdot \#RE \cdot \#EC$). Such requests are actually distributed on the $(\#R - 1) \cdot (\#RN - \#RE)$ nodes ouside the rack that are not hosting an executor. The per-process throughput of the disk of each non-executing node is $\delta_r(\lceil \frac{(\#R-1) \cdot \#RE \cdot \#EC}{(\#R-1) \cdot (\#RN - \#RE)} \rceil)$, and thus the time needed to access the disk is:

$$ReadT_C = \frac{Size}{\delta_r(\lceil \frac{(\#R-1) \cdot \#RE \cdot \#EC}{(\#R-1) \cdot (\#RN - \#RE)} \rceil)} = \frac{Size}{\delta_r(\lceil \frac{\#RE \cdot \#EC}{(\#RN - \#RE)} \rceil)}$$

During a cluster wave, $\#EC$ partitions are transferred in parallel to the cores of each executor from one of the $(\#R - 1) \cdot (\#RN - \#RE)$ non-executing nodes belonging to a different rack of the cluster, thus the number of cores sharing the same node-to-node network connection can be bounded by $\lceil \frac{\#EC}{(\#R-1) \cdot (\#RN - \#RE)} \rceil$. The time needed to transmit data through each single network connection is:

$$TransT_C = \frac{Size}{\rho_e(\lceil \frac{\#EC}{(\#R-1) \cdot (\#RN - \#RE)} \rceil)}$$

4.3 Write

Once read and processed in main memory, each RDD partition is written back to the local disk. $Write(Size)$ computes the time taken by an executor to write to the disk $Size$ MB. The disk write throughput depends on the number of executor cores.

$$Write(Size) = \frac{Size \cdot sCmp}{\delta_w(\#EC)}$$

The formulae embed the size reduction (e.g. $sCmp = 0.6$) due to data compression carried out before data are saved to the disks. This is a configuration option in Spark.

4.4 Shuffle Read

When performing a shuffle read, Spark generates $\#SB$ tasks which are in charge of processing the $\#SB$ buckets previously created during the shuffle write phase. Each bucket is evenly distributed between the executors, that is, each executor stores a portion of each bucket. $SRead(Size)$

models the time needed for reading a single bucket of $Size$ MBs.

The reading of a data bucket and its transmission to the executor, happen in pipeline so that, according to the Volcano model, the loading time is computed as the maximum of the times needed to carry out the two operations:

$$SRead(Size) = \text{MAX}(ReadT, TransT)$$

We emphasize that all and only the cluster nodes hosting an executor are involved in a shuffle read. Each executor behaves in the same way and data locality cannot be applied since data are not replicated. Moreover, each single bucket is distributed across all the executors. Each executor stores $ExecBucketSize = Size/\#E$ MBs for each bucket. Each core requests in parallel to all the executors the bucket portion, and each executor must fulfill $\#E \cdot \#EC$ requests (i.e., $\#EC$ from the local cores, $(\#E - 1) \cdot \#EC$ from the remote ones) of size $ExecBucketSize$. Due to parallelism of the requests, the reading time is:

$$ReadT = \frac{ExecBucketSize}{\delta_r(\#E \cdot \#EC)}$$

As to network time, we emphasize once again that the role of each executor is symmetric and that each node-to-node connection is shared by $\#EC$ processes (i.e., each of the executor's core requests a bucket portion from all the other executors). For the same reason, the number of processes transmitting and receiving on each connection is the same, therefore send and receive time is the same. Assuming that inter-rack network speed is lower than intra-rack one, we can use the inter-rack network throughput to constraint the time needed to complete the transmission, except when all the v executors are allocated in the same rack. This happens with probability:

$$P_{SR}(v) = \begin{cases} \frac{\binom{\#RN}{v}}{\binom{\#N}{v}} \cdot \#R, & \text{if } v \leq \#RN \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

the probability is 0 if the number of executors is higher than the number of nodes in a rack. If this is not the case, the fraction gives the probability of allocating v executor on a single rack with $\#RN$ nodes out of a cluster.

Therefore, the transmission time can be computed as follows:

$$TransT = \frac{ExecBucketSize}{P_{SR}(\#E) \cdot \rho_i(\#EC) + (1 - P_{SR}(\#E)) \cdot \rho_e(\#EC)}$$

Example 3 *In a cluster with $\#R = 2$, $\#RN = 3$, and $\#E = 2$, the probability of having the 2 executors allocated in the same rack is:*

$$P_{SR}(2) = \frac{\binom{3}{2}}{\binom{6}{2}} \cdot 2 = \frac{3}{15} \cdot 2 = 0.4$$

where 3 and 15 are the number of different executor allocations on the a single rack and on the whole cluster, respectively.

4.5 Broadcast

A broadcast on a RDD (a) collects at the application driver all the RDD partitions, each of size $Size$; (b) distributes the whole RDD to the executors for further processing. The time needed to complete a broadcast is the sum of time needed to complete steps (a) and (b), since only when the whole RDD is loaded on the application driver, it sends the RDD back to the executors.

$$Braodcast(Size) = CollectT + DistributeT$$

$\#E \cdot \#EC$ partitions are collected in parallel (i.e., one by each executor core) and each node-to-driver network connection is shared by $\#EC$ processes. Similarly to the shuffle read case, the network throughput can be limited through the inter-rack network one except when both the executors and the application driver (i.e., $\#E+1$) have been instantiated in the same cluster (See Formula 1). $CollecT$ can be computed as:

$$\frac{Size}{P_{SR}(\#E+1) \cdot \rho_i(\#EC) + (1 - P_{SR}(\#E+1)) \cdot \rho_e(\#EC)}$$

As to step (b), for the purpose of the cost model (see Subsection 5.3), we consider here the cost for distributing the whole RDD in tranche of $\#E \cdot \#EC$ RDD partitions (i.e., those that are collected in parallel). Since the application driver sends all the data collected to each node, only one process is active on each network connection. $DistributeT$ can be computed as:

$$\frac{Size \cdot \#E \cdot \#EC}{P_{SR}(\#E+1) \cdot \rho_i(1) + (1 - P_{SR}(\#E+1)) \cdot \rho_e(1)}$$

$DistributeT$ is not the time needed for distributing the whole RDD, but rather the one needed for distributing the data collected in a wave. As we will shown in Section 5.3, we consistently use such value to compute the total time.

5 Modeling GPSJ Queries

In this section we describe how the basic bricks described in the previous section can be composed to model a GPSJ query execution plan. Execution time sums up the time needed to execute the nodes of the tree coding the physical plan (see Section 3). Each node corresponds to a task type, and each task type is associated to a specific cost function (see Sub-Section 5.2 - 5.6) that returns the cost along with the features of the relational table produced in output. A depth-first visit of the tree ensures all the input table features to be available when computing the cost of current node.

5.1 Statistics and Selectivity Estimates

Creating a cost based model requires to collect statistics from the DB in order to estimate the predicate selectivities and the size of the tables. The literature on such topics is very broad. The accuracy of the estimate strictly depends on the collected information and on the assumptions made on data distribution. Following several query cost models, in this paper we assume uniformity of attribute values, attribute values independence and join containment [6].

A table t includes a set of attributes $t.Attr$. For each table we collect its cardinality $t.Card$ and its size $t.Size$ when stored in an uncompressed file format. We also collect the average size reduction achieved by a compressed file format (e.g. Parquet [1]) $fcmp$, and the average size of the HDFS partitions $t.PSize$ storing the table. Although, the partition size is a HDFS parameter (typically set to 128 MB), its actual size can vary heavily and can be much smaller than the theoretic size when the tables are small or when they are created through a Spark command and they are not compacted.

For each attribute $a \in t.Attr$ we collect the number of distinct values $a.Card$, and its average length $a.Len$ in bytes. Based on the previous statistics, and considering the well-known works by [23] and [21] we are able to estimate the selectivity of conjunctive selection predicate $Sel(t, pred)$, the cardinality $JCard(t_1, t_2, pred)$ and the size $JSize(t_1, t_2, pred)$ of equi-join $t_1 \bowtie_{pred} t_2$. We do not report here the related formulae for space reasons.

We also estimate the percentage length reduction determined by projection $Proj(t, cols)$ on attributes $cols \subseteq t.Attr$

$$Proj(t, cols) = \frac{\sum_{a \in cols} a.len}{\sum_{a \in t} a.len}$$

An estimatimate of the cardinality reduction induced by a generalized projection (i.e., grouping factor) can be obtained exploiting the Cardenas formula[6] [5]

$$Group(\#tuples, \#groups) = \frac{\Theta(\#tuples, \#groups)}{\#tuples}$$

For example grouping a table t on a group-by set $group$ after applying a filtering predicate $pred$ will be defined as:

$$Group(t.Card \cdot Sel(t, pred), \prod_{a \in Group} a.Card)$$

$Sel()$, $Proj()$, $Group()$ are set to 1 if the corresponding parameters are not defined.

5.2 The SC() Task Type

A Scan task accesses a table t stored in HDFS. The function $SC(t, pred, cols, groups, pipe)$ returns the time needed to carry out the task. The basic operations this task type carries out are:

(a) Fetching the RDD partitions storing t into the memory. Fetching involves accessing HDFS to retrieve the RDD partitions and sending them to the executors that are in charge to process them. Transmission of data through the network is required only for those partitions that are not stored locally to the executor. Since Spark adopts a Volcano style pull model, the scan time will be the maximum between the access and transmit times.

(b) Filtering (optional) the table tuples according to the predicate $pred$. Since Catalyst applies selection push down, filtering is carried out as soon as a tuple is no more useful for further computations. When supported by the used file format (e.g. Parquet), Spark can push filtering into the sources files, in this case the unused tuples will not be read at all.

(c) Projecting (optional) the table by dropping the unused columns (i.e., the columns of t not included in $cols$). Since Catalyst applies projection push down, dropping is carried out whenever a column is no more useful for further computations. When supported by the underlying file format (e.g. Parquet), Spark can push projection into the sources files, in this case the unused columns will not be read at all.

(d) Aggregating (optional) the tuples when Catalyst pushes down generalized projection in order to reduce the quantity of data to be handled in further processing.

(e) Writing (optional) to the disks the remaining tuples for further elaborations or for saving the final result. Writing is avoided when a broadcast join is pipelined (i.e., $pipe = T$).

According to our cost model only operations (a) and (d) must be directly modeled, whereas operations (b) and (c) influence performances since they reduce the quantity of data to be written back to the disk.

The number of RDD partitions composing t is

$$\#TableP = \frac{t.Size \cdot fcmp}{t.PSize} \tag{2}$$

if the table is stored in compressed format $0 < fcmp < 1$, $fcmp = 1$ otherwise. Each RDD partition implies $RSize = t.PSize$ bytes to be fetched from the disk. The quantity of data to be read from each RDD partition is reduced to $RSize = t.PSize \cdot Sel(t, pred) \cdot Proj(t, cols)$ when filtering and projection are pushed down to the data sources.

Fetching is executed in $\lceil \frac{\#TableP}{\#E \cdot \#C} \rceil$ waves. We distinguish 3 types of waves, namely L *-local*, R - *rack* and C - *cluster*, depending on where the RDD partition to be processed is stored with reference to the executor in charge of fetching it. Given a RDD partition p, the probability an executor has to fetch p locally (P_L), within its rack (P_R) or anywhere else (P_C) depends on the cluster topology and can be calculated through the following formulae:

$$P_L = 1 - \frac{\binom{\#N - rf}{\#E}}{\binom{\#N}{\#E}} \tag{3}$$

$$P_C = \sum_{x=1}^{min(\#R, rf)} \sum_{y=1}^{min(\#R, \#E)} P_{Part}(x) \cdot P_{Exe}(y) \cdot \frac{\binom{\#R - x}{y}}{\binom{\#R}{y}} \tag{4}$$

$$P_R = 1 - P_L - P_C \tag{5}$$

In Formula 3, the fraction gives the rate of executor allocations on cluster nodes that do not store one of the rf replicas of p, over the number of executor allocations on the whole cluster. In Formula 4, $x \in \{1, \cdots, min(\#R, rf)\}$ is the number of racks having at least one of the rf replicas of p in any of their nodes, $y \in \{1, \cdots, min(\#R, \#E)\}$ is the number of racks having at least an executor allocated. The fraction gives the rate of allocations of y racks on $\#R - x$ racks (i.e., racks with no p replicas in their nodes), over the total number of allocations of y racks. Such fraction must be weighted by the probability that exactly x racks host one of the rf replicas of p (i.e., $P_{Part}(x)$) and the probability that exactly y racks have at least one executor allocated (i.e., $P_{Exe}(y)$):

$$P_{Part}(x) = \binom{\#R}{x} \cdot \sum_{j=0}^{x} (-1)^j \binom{x}{j} \frac{\binom{\#RN \cdot (x-j)}{rf}}{\binom{\#N}{rf}} \tag{6}$$

6. Cardenas formula $\Theta(R, N)$ has been originally defined to estimate the number of data blocks actually accessed to retrtieve, through an indexed attribute, R records from a relational table stored in N blocks. Cardenas formula assumes uniform distribution for the indexed attribute values and for the tuples accross blocks.

$$P_{Exe}(y) = \binom{\#R}{y} \cdot \sum_{j=0}^{y} (-1)^j \binom{y}{j} \frac{\binom{\#RN(y-j)}{\#E}}{\binom{\#N}{\#E}} \quad (7)$$

In Formula 6, the inclusion-exclusion principle has been employed in order to calculate the probability that x given racks, and only them, host at least one of the rf replicas of p. Such probability is then multiplied by $\binom{\#R}{x}$ as any x racks chosen show the same probability. Formula 7 has been built up symmetrically.

Once a RDD partition is fetched operations (b) to (d) are performed on its tuples and $WSize$ bytes are written back to the disk, where:

$$\begin{aligned} WSize =& t.PSize \cdot Sel(t, pred) \cdot Proj(t, cols) \cdot \\ & Group(t.Card \cdot Sel(t, pred), \prod_{a \in groups} a.Card) \end{aligned}$$

If no grouping is carried out, reading and writing of data can be performed in pipeline, otherwise writing can start only after all the tuples are loaded and grouped. In the first case the execution time of each task is estimated as the maximum between reading and writing times.

$$\begin{aligned} SC(t, pred, cols, groups, pipe) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \\ \sum_{X \in \{L,R,C\}} P_X \cdot \text{MAX}(Read(RSize, X), Write(WSize)) \end{aligned} \quad (8)$$

In the second case, the two times must be summed up.

$$\begin{aligned} SC(t, pred, cols, groups, pipe) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \\ \sum_{X \in \{L,R,C\}} P_X \cdot (Read(RSize, X) + Write(WSize))) \end{aligned} \quad (9)$$

In formulae 8 and 9 $WSize = 0$ if data must not be written back to the disk (i.e., the $pipe = T$).

Table 3 reports the features of the returned table t'.

5.3 The SB() Task Type

A Scan & Broadcast accesses a table t stored in HDFS and sends it to the application driver that collects the RDD partitions and broadcast the whole table to all the executors. The function $SB(t, pred, cols)$ returns the time needed to carry out the task.

Data collections steps are the same for the SC() task type, we refer to formulae from 2 to 5 for its modeling. The size of each RDD partition to be broadcasted may be reduced due to filtering and projection predicates:

$$BrSize = t.PSize \cdot Sel(t, pred) \cdot Proj(t, cols)$$

Note that, since data are filtered and projected either in main memory or directly whereas they are read from the disks, they do not directly impact on the task cost. Since data collection and broadcast work in pipeline, the execution time is:

$$\begin{aligned} SB(t, pred, cols) = \lceil \frac{\#TableP}{\#E \cdot \#EC} \rceil \cdot \\ \sum_{X \in \{L,R,C\}} P_X \cdot \text{MAX}(Read(RSize, X), Broadcast(BrSize)) \end{aligned}$$

Table 3 reports the features of the returned table t'.

5.4 The SJ() Task Type

A Shuffle Join carries out a join of two tables t_1 and t_2 whose partitions have been previously hashed in $\#SB$ buckets. Input RDD partitions are stored in the local disk of the executors. The function $SJ(t_1, t_2, pred, cols, groups, pipe)$ returns the time needed to carry out the task. The operations carried out by SJ() in one wave are:

(a) Shuffle Read: the corresponding buckets from t_1 and t_2 are fetched. A portion of each bucket is stored on each executor.
(b) Join: once two corresponding buckets from t_1 and t_2 are fully available, the $pred$ predicate is used to to merge the tuples.
(c) Project (optional): columns that are no more useful for the remaining part of query are dropped. Only the columns in $cols$ are returned.
(d) Aggregating (optional) the tuples when Catalyst pushes down generalized projection in order to reduce the quantity of data to be handled in further processing.
(e) Writing (optional) to the disks the remaining tuples. Writing is avoided when a broadcast join is pipelined.

Join cannot start before all the tuples of the corresponding buckets are loaded. Consequently, the time needed for carrying out one wave is the sum of time needed for shuffle reading and writing back (if needed) the processed tuples. The quantity of data to be read at each wave is

$$RSize = \frac{t_1.Size + t_2.Size}{\#SB}$$

The quantity of data to be written back to the disk by each core at each wave is computed as:

$$\begin{aligned} WSize = \frac{JSize(t_1, t_2, pred) \cdot Proj(t_1 \bowtie t_2, cols)}{\#SB} \cdot \\ \cdot Group(JCard(t_1, t_2, pred), \prod_{a \in groups} a.Card) \end{aligned}$$

$WSize = 0$ if data must not be written back to the disk (i.e., the $pipe = T$). The overall time is the sum of the time taken by all the wave:

$$\begin{aligned} SJ(t_1, t_2, pred, cols, groups) = \lceil \frac{\#SB}{\#E \cdot \#EC} \rceil \cdot \\ \cdot (SRead(RSize) + Write(WSize)) \end{aligned}$$

Table 3 reports the features of the returned table t'.

5.5 The BJ() Task Type

A broadcast join carries out a join of tables t_1 and t_2 when one of the two tables, let say t_1, is small enough to be broadcasted and completely kept in the main memory of each executor. The function $BJ(t_1, t_2, pred, cols, groups, pipe)$ returns the time needed to carry out the task. To the aim of our model, the only relevant operation is the writing of the data back to the disk. This is because the costs for loading the two tables are charged on the children nodes of the execution tree. According to the GPSJ grammar (see Figure 2): t_1 is loaded through a SB() task type, whereas t_2 is either resulting from a previous operation (i.e., SJ(), BJ()) or is loaded through a SC() operation. Similarly to shuffle join

TABLE 3
Estimated features for the output table t' of the different task types.

Task type	$t'.Attr$	$t'.Card$	$t'.Size$	$t'.Part$
SC()	$cols$	$t.Card \cdot Sel(t,pred) \cdot Group(t.Card \cdot Sel(t,pred), \prod_{a \in groups} a.Card$	$WSize \cdot \#TableP$	$\#TableP$
SB()	$cols$	$t.Card \cdot Sel(t,pred)$	$BrSize \cdot \#TableP$	$\#TableP$
SJ()	$cols$	$JCard(t_1,t_2,pred) \cdot Group(JCard(t_1,t_2,pred), \prod_{a \in groups} a.Card)$	$WSize \cdot \#SB$	$\#SB$
BJ()	$cols$	$JCard(t_1,t_2,pred) \cdot Group(JCard(t_1,t_2,pred), \prod_{a \in groups} a.Card)$	$WSize \cdot t_2.Part$	$t_2.Part$
GB()	$cols$	$t.Card \cdot Group(t.Card, \prod_{a \in groups} a.Card)$	$WSize \cdot \#SB$	$\#SB$

filtering, projection and grouping may be optionally carried out in main memory.

Broadcast join is always pipelined to another operation, either scan, shuffle join, or broadcast join, thus the number of waves it requires depends on the partition number of t_2. The quantity of data to be written by each core at each wave is:

$$WSize = \frac{JSize(t_1,t_2,pred) \cdot Proj(t_1 \bowtie t_2, cols)}{t_2.Part} \cdot \\ \cdot Group(JCard(t_1,t_2,pred), \prod_{a \in groups} a.Card)$$

The time needed to complete the operation is the sum of the time taken by the waves:

$$BJ(t_1,t_2,pred,cols,groups) = \lceil \frac{t_2.Part}{\#E \cdot \#EC} \rceil \cdot Write(WSize)$$

Table 3 reports the features of the returned table t'.

5.6 The GB() Task Type

Group by carries out the final grouping. The tuples of the input table t have been previously hashed in $\#SB$ buckets. Input RDD partitions are stored in the local disk of the executors. The function $GB(t,pred,cols,groups)$ returns the time needed to carry out the task. At each wave, the operations carried out by GB() are:

(a) Shuffle Read the corresponding buckets from t. A portion of each bucket is stored on each executor.
(b) Data are grouped according to the $groups$ attributes and the aggregated values for the remaining $cols - groups$ attributes are computed.
(c) Aggregated tuples are written back to the disk.

Grouping cannot start before all the tuples of the corresponding buckets are loaded. Consequently, the time needed for carrying out one wave is the sum of time needed for shuffle reading and writing back the processed tuples. The quantity of data to be read by each core at each wave is:

$$RSize = \frac{t.Size}{\#SB}$$

The quantity of data to be written back to the disk by each core at each wave is:

$$WSize = RSize \cdot hSel \cdot Proj(t,cols) \cdot \\ \cdot Group(t.Card, \prod_{a \in groups} a.Card)$$

where $hSel$ is a constant selectivity factor for all having clauses ($hSel = 0.33$ by default); $hsel$ is set to 1 if $pred$ is undefined. Although such a simple estimate is often different from the actual selectivity, it is standard solution in commercial systems. More refined estimates imply assumptions that are not coherent with our framework [10]. The overall time is the sum of the time taken by all the waves:

$$GB(t,pred,cols,groups) = \lceil \frac{\#SB}{\#E \cdot \#EC} \rceil \cdot \\ \cdot (SRead(RSize) + Write(WSize))$$

Table 3 reports the features of the returned table t'.

5.7 Handling Stragglers

The cost model presented so far computes the analytic cost of a GPSJ query assuming cluster resources to be sufficient to run the workload without any straggler taking place. A *straggler* is a task that performs more poorly than similar ones due to insufficient assigned resources [26]. Stragglers probability is strictly related to cluster/nodes loading and to their consequent level of resource contentions. We emphasize that our model inherently considers resource contentions when it defines network and disk throughput (see Section 5), but when resources become insufficient, performances can be reduced in unpredictable ways. Several techniques [4] have been studied to reduce and prevent stragglers effects. Spark itself implements a speculative execution technique that is running, on a different node, a copy of a task suspected to be a straggler: the result will be taken from the task finishing first.

Building an analytic model for stragglers is challenging due to complex task-to-node and task-to-task interactions [26], thus in our model we keep stragglers into account by adding, to the basic task cost estimation, an extra time computed through a straggling profile computed for each task type $TT \in$ {SC(),SB(),SJ(),BJ(),GB()}. Building a profile requires the real execution times $t()$ of a set of tasks L_{TT}^{Load} of a given type TT to be collected. $Load$ defines the quantity of resources devoted to the task executions in terms of percentage of cluster executors $Load.Ex = \#E/\#N$ and executor cores $Load.C = \#EC/\#C$.

Definition 2 (Straggler [26]) *A task $i \in L_{TT}^{Load}$ is said to be a* straggler *if*

$$t(i) > \beta \cdot median(t(L_{TT}^{Load}))$$

β measures the sensitivity of the approach to stragglers [7].

7. We set $\beta = 1.3$ through an optimization process. Noticeably, it is the same value adopted in [26] that adopts a similar definition.

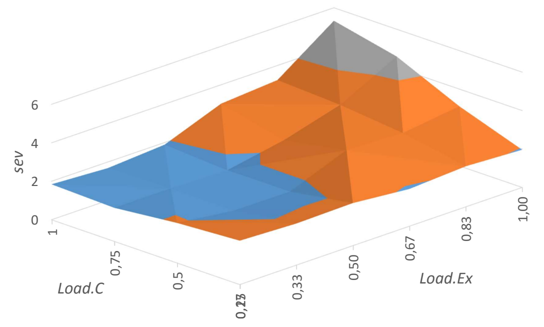


Fig. 6. The severity profile for $SC()$ on C_1.

Definition 2 allows L_{TT}^{Load} to be partitioned in stragglers $\widehat{L}_{TT}^{Load}$ and non-straggler $\bar{L}_{TT}^{Load}$ tasks. We can now define the straggler tendency and severity as:

$$tend_{TT}^{Load} = |\widehat{L}_{TT}^{Load}|/|L_{TT}^{Load}|$$
$$sev_{TT}^{Load} = \frac{AVG(t(\widehat{L}_{TT}^{Load}))}{AVG(t(\bar{L}_{TT}^{Load}))}$$

$tend$ measures how often stragglers take place; sev measures how longer is the execution time for stragglers with reference to non-straggler tasks. Since both measures are expressed as a percentage, they can be applied to tasks of different absolute duration. Since computing sev and $tend$ for all possible resource configurations can be heavy for large clusters, we compute the profile for a limited subset of configurations and then we estimate the remaining ones through a regression. Figure 6 shows the cluster severity profile for the scan task. As expected, severity increases when for higher cluster loads.

Given a task t of type TT and the resources setting $Load$, it is now possible to include stragglers effect in our analytic model as follows:

$$\widehat{et}(t) = et(t) \cdot (1 - tend_{TT}^{Load}) + et(t) \cdot tend_{TT}^{Load} \cdot sev_{TT}^{Load}$$

where $et()$ is the expected execution time returned by the analytic model. The second part of the formula applies a execution time adjustment proportional to straggler severity and tendency.

6 Experimental Results

The main goal of our tests is to verify the accuracy of the cost model estimates both in absolute and in relative terms. On the one hand, an accurate estimate of a query duration is useful to obtain a clear understanding of the system performance under different settings. On the other hand, being able to compare execution plans in term of their execution time it is mandatory to select the best one. In the latter case, it is not relevant the absolute accuracy of the estimates, but rather it is important that the cost model preserves the relationship between the two execution times. For these reasons, given a workload $Q = \{q_1, ...q_n\}$ we define two accuracy measures based on a couple of functions: $t(q)$ returning the execution time (in seconds) measured by Spark and, $et(q)$ returning the expected execution time (in seconds) estimated by our cost model. The first accuracy measure is the *relative error*

$$err(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{|t(q) - et(q)|}{t(q)} \quad (10)$$

that captures the ability of the model to properly estimate the execution times, the second measure is the *statistical correlation*[8] between $t()$ and $et()$ on Q:

$$cor(Q) = \frac{\sum_{q \in Q}(t(q) - \bar{t}(Q))(et(q) - \overline{et}(Q))}{\sqrt{\sum_{q \in Q}(t(q) - \bar{t}(Q))^2 \sum_{q \in Q}(et(q) - \overline{et}(Q))^2}}$$

where $\bar{t}(Q)$ and $\overline{et}(Q)$ are the sample means and standard deviations for $t()$ and $et()$ on Q. $cor(Q)$ measures how well the cost model estimates are related to the Spark execution times. We do not carry out efficiency tests since the time needed to compute the cost is always negligible (49 millisecs in the average). The computational effort is given by the in memory visit of the tree coding the physical plan that, as mentioned in Section 3, has at most $2 \cdot n$ nodes, where n is the number of tables involved in the query. On the contrary, building the straggler profiles comes with a cost that is proportional to the number tested configurations (at most $\#E \cdot \#C$). The more the tested configurations, the greater the computation time and the lower the regression error. For example, building the straggler profiles for C_1 testing $33/17$ configurations (uniformly distributed on the 56 feasible ones) reduces the computation time from 7.1 to $3.3/2.2$ hours respectively. Conversely, $\widehat{err}$ for Q_2 only marginally grown from 0.155 to 0.159 $/0.162$.

Table 4 reports the features of the two clusters used for testing. The clusters have different sizes: C_1 is on-premisis, C_2 is built on the Google cloud platform; disk and network throughputs have been derived experimentally through a separate tuning procedure. Having similar performances on different versions of the platform testifies to the robustness of the approach. The evaluation has been carried out employing 3 well-known benchmarks: Q_1 is the Big Data benchmark [14] with size 120GB (highest cardinality among tables: $7.5 * 10^8$); Q_2 is the TPC-H [16] benchmark sized to 100GB (highest cardinality among tables: $6 * 10^8$), finally Q_3 is the TPC-H benchmark sized to 1 TB (highest cardinality among tables: $6 * 10^9$). Data in the three benchmarks are uniformly distributed as assumed by our model. Queries in Q_1 are rather simple and typically include one or few task types. Conversely, queries in Q_2 and Q_3 show the full GPSJ expressiveness and beyond, thus we have chosen 4 *base* queries (namely TPC-H queries q_1, q_3, q_6 and q_{10}) and we built on them to create benchmarks able to stress our cost model. In all the following tests we varied the cluster configurations in terms of number of executors ($\#E \in [1, .., 6]$ for C_1; $\#E \in \{10, 30, 50\}$ for C_2) and number of cores ($\#C \in \{2, 4, 6, 8\}$ for C_1; $\#C \in \{2, 6\}$ for C_2) for a total of 24 and 6 configurations for C_1 and C_2, respectively. All the queries have been run three times[9] and the average execution time has been considered. As shown in Table 4,

8. For stability reasons we do not compute $cor(Q)$ for benchmarks such that $|Q| < 10$

9. In absence of stragglers, 3 repetitions are sufficient to ensure result stability: for example, $AVGt(q)$ on Q_2 computed on 3 and 16 repetitions differ 10.3% only.

the benchmarks have been tested only on the cluster with an appropriate computational power.

6.1 Accuracy for Single Task Types

The first test is aimed at evaluating the accuracy of the cost functions on specific task types. To this aim we extracted the costs of different task types for queries in Q_1, Q_2 and Q_3. For the aim of this test, we varied the selectivity of predicates (i.e., $Sel() \in \{0, 0.25, 0.5, 0.75, 1\}$) and the reduction rate of the aggregation ($Group() \in \{0.66, 0.95, 0.99\}$), along with the cluster configurations. As shown in Table 5 all the task types are modeled with a similar accuracy independently from the cluster they are executed on.

6.2 Accuracy for Full GPSJ Queries

A similar accuracy have been obtained with queries with full GPSJ expressiveness and, as shown in Table 6, it is 20% in average. The accuracy of the cost model is stable for different cluster configurations and for queries that differ in the duration, number of tasks and waves. This behavior can be better appreciated in Figure 7 where, for each query q summarized in Table 6, compares $t(q)$ and $et(q)$. It is apparent, that most of the queries are close to the scatter plot diagonal (i.e., $t(q) = et(q)$) and that, although the error is higher for longer queries, it is constant in relative terms. The scatterplot also shows a slight upward translation. This means that the estimated costs are, in general, slightly lower than the real ones. This is due to presence of stragglers, as we will shown in Section 6.3.

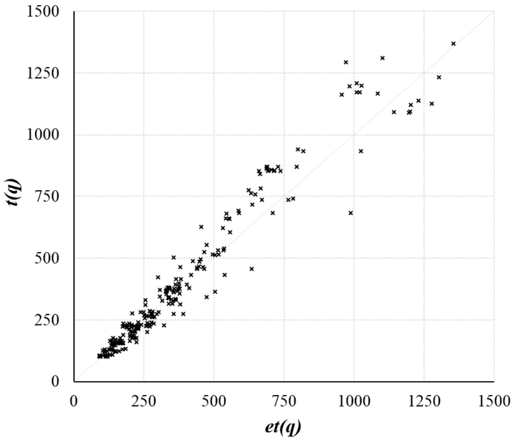
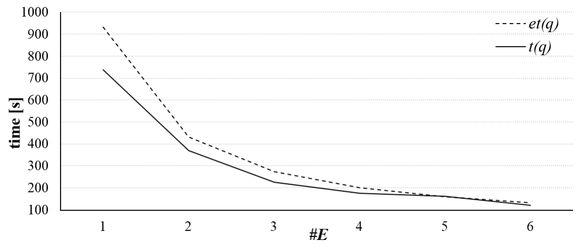


Fig. 7. Scatterplot of execution time VS model expected time for Q_2.

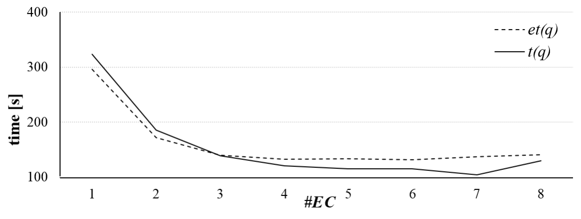
The detailed performances for different cluster configurations are reported in Figure 8.a and 8.b that refer to the execution of query $q_1 \in Q_2$ varying the number of cores ($\#EC$) and the number of executors ($\#E$) allocated to the Spark application running the query. The figures also demonstrate a possible application of the cost model: both charts show a sub-linear improvement of performances, thus it is useless to allocate more than 4 cores to each of the six executors due to the concurrence on the disk resource.

6.3 Error Analysis and the Straggler Handling Strategy

We emphasize that only few works propose a cost model detailed enough to enable a comparison between the estimated



(a)



(b)

Fig. 8. Cost Model vs Spark execution time for $q_1 \in Q_2$ varying (a) the number of executors allocated to the query ($\#EC = 4$), and (b) the number of cores allocated to each executor ($\#E = 6$).

and real execution times. As a point of comparison we report the error rates measured in [25] for generic MapReduce jobs ($err() \in [15\%; 25\%]$) and in [19] for federated DBMSs ($err() \leq 20\%$). Both of them are consistent with our one.

A deviation from the correct value is intrinsically expected due to the assumptions and simplifications made by the cost model. In particular, the model assumes that no skewed attributes are present. If this is not true, the model would misjudge the output cardinalities of selections and joins. To evaluate the impact of such errors we modified data distribution in Q_2 so that in each query, the first selection predicate and the first join predicate operate on a skewed attribute. Skewness is obtained through a gaussian distribution of attribute values. We run two additional tests where 200% and 50% of the tuples expected by the model are actually selected. The model error, $err(Q_2)$, increases by 8.7% and 9.8% for selections and joins, respectively.

We also noted that, running the same query several times, execution times is likely to fluctuate. Although most of the executions behave similarly, query execution time deviates from the median value more than 2.5 times for the 7% of the observations. Correlation remains higher and more stable than err, this points out that, although the model may miss the correct estimate, it keeps consistently the correct relationship between queries that is necessary for comparing different execution plans of a given query (see Section 6.5).

Deviations are strictly related to stragglers that become more frequent and severe when the cluster load increases (see Figure 6). All the tests presented so far do not keep stragglers into account because we focused on the capabilities of our analytic model in returning a valuable execution time estimate without relying on execution profiles. The straggler strategy further improves the cost estimate accuracy. Column $\widehat{err}(Q)$ in Table 6 shows that the straggler handling strategy reduces the error by 6% on the average.

TABLE 4
Cluster features & benchmark usage.

Name	Installation	#R	#N	#C	Main Mem.	Disk	Sw. Releases	Q_1	Q_2	Q_3
C_1	On Premises	2	7	8	32 GB	6 TB	Hadoop 2.6.0 + Spark 1.5.0	✓	✓	
C_2	Cloud (Google)	1	51	8	30 GB	512 GB	Hadoop 2.7.2 + Spark 2.0.1			✓

TABLE 5
Task types accuracy, in bold the task type the test focuses on.

Task types	*Bench.*	$\|Q\|$	*AVG t(q)*	err	cor
SC()	Q_1	120	428	0.11	0.99
	Q_2	120	279	0.23	0.98
	Q_3	30	255	0.27	0.99
GB()	Q_1	72	138	0.33	0.87
	Q_2	72	23	0.26	0.95
	Q_3	18	53	0.18	0.99
SJ()	Q_1	120	606	0.17	0.99
	Q_2	120	662	0.31	0.99
	Q_3	30	1021	0.19	0.92
BJ()	Q_1	120	235	0.18	0.97
	Q_2	120	240	0.21	0.94
	Q_3	30	402	0.22	0.96
			Overall Average	0.22	0.96

TABLE 6
Accuracy for queries with full GPSJ expressiveness.

Base Query	*Bench.*	$\|Q\|$	*Total tasks*	*AVG waves*	*AVG $t(q)$*	err	cor	$\widehat{err}$
q_1	Q_2	24	1010	58	374	0.17	0.96	0.18
	Q_3	6	6263	40	666	0.23		0.21
q_3	Q_2	24	1562	90	516	0.24	0.97	0.13
	Q_3	6	8197	52	374	0.15		0.10
q_6	Q_2	24	1010	58	313	0.34	0.99	0.21
	Q_3	6	6263	40	529	0.05		0.04
q_{10}	Q_2	24	1563	90	452	0.20	0.98	0.11
	Q_3	6	8198	52	656	0.19		0.12
					Overall Average	0.20	0.98	0.14

$\widehat{err}(Q)$ is defined as $err(Q)$ (see Equation 10) considering $\widehat{et}(q)$ instead of $et(q)$. Stragglers are properly detected for all task types and for all cluster loads.

6.4 Query Accuracy in Presence of Compressed Data

Our cost model handles projection predicates and data compressed file format. Projection predicates are modeled both when they are executed in-memory and when they are pushed-down to the disk. To test such capabilities we run the test reported in Table 7. The test is based on a select query implemented through a single SC() task type. In the base query we progressively reduced the number of attributes returned so that $Proj()$ ranges from 1.0 to 0.4. Tests have been executed on C_1 varying 24 cluster configurations. We run the tests both on data stored in HDFS as text files and on data stored in the compressed format. Parquet allows projection to be carried out directly on the disk by preventing the reading of the data. Execution time due to projection for queries run on text files takes place when data are written back to the disk since, in this case, Spark needs to read the whole table. Conversely, when data are stored in Parquet, saving takes place during both the reading and the writing phases since unnecessary data are not read at all. Beside reading/writing only the projected data, Parquet data are compressed and this further reduces query execution time. In all the cases the cost model correctly captures execution times reduction and keeps the relative errors under 30%. Correlation remains very high.

TABLE 7
Model accuracy varying the projection factor for scan queries on C_1.

Proj()	$\|Q\|$	Format	*AVG t(q)*	*r_err*	*cor*
1.0	24	txt	785	0.29	0.94
		parquet	*404*	*0.28*	*0.97*
0.70	24	txt	488	0.30	0.95
		parquet	*289*	*0.29*	*0.98*
0.40	24	txt	387	0.19	0.97
		parquet	*156*	*0.28*	*0.99*

6.5 Execution Plan Selection

One of the main goals of a cost model is to allow the *best* execution plan to be selected. In previous subsections we have shown the high level of correlation between $t()$ and $et()$, here we show that it is sufficient to select the *best* plan.

We remark that our starting point is the only physical plan returned by Catalyst. Catalyst does not make available other alternative plans to be compared, but since it does not carry out join reordering, it is still possible to generate alternative plans by changing the order of tables in the FROM clause. Further plans can be generated changing catalyst parameters (i.e. turning Brodacast join option on/off.) We initially set a cluster configuration ($\#E = 2$ and $\#EC = 4$) on C_1 and we defined three full GPSJ queries (q_a, q_b and q_c) on Q_2. The queries involve from 3 to 4 tables, and one or more selection predicates that reduce the number of involved tuples. For each query we obtained different plans by changing the join order, and by enabling/disabling the broadcast join parameter. For example in q_a a possible join order is $Lineitem \rightarrow Orders \rightarrow Customer \rightarrow Nation$ where $Nation$, due to its reduced size, can be joined to the other ones either using a shuffle or a broadcast join. Results are reported in Table 8. It is apparent that, given a query:

- the different execution plans determine different execution times, that can be either significantly distant or very close (see q_b and the third and fourth plans for q_a);
- the average relative error is comparable with the one measured in previous tests;
- $t()$ and $et()$ are strictly correlated and for all the three queries $et()$ allows to choose the cheapest plan between those available. More in details, the plans orderings induced by $t()$ and $et()$ are the same, thus the plan with the lowest expected execution time $et()$ is also the one with the lowest execution time $t()$.

TABLE 8
Comparison of alternative physical plans run on cluster C_1 and benchmark Q_2. Sel specifies the selectivity of the predicates on different tables Xb means that table X has been broadcasted. Bold times are the lowest ones.

Query	*Sel.*	*Plan*	$t()$	$et()$	$err()$
q_a	$N = 0.04$	$L \to O \to C \to N$	4 320	5 100	0.18
		$L \to O \to C \to Nb$	2 760	3 420	0.24
		$N \to C \to O \to L$	1 745	1 200	0.31
		$Nb \to C \to O \to L$	**1 740**	**1 140**	0.34
q_b	$L = 0.67$, $O = 0.50$, $C = 0.20$	$L \to O \to C$	1 680	1 560	0.07
		$C \to O \to L$	**1 500**	**1 020**	0.32
q_c	$O = 0.50$	$L \to O \to C$	2 280	1 920	0.16
	$C = 0.20$	$C \to O \to L$	**1 200**	**1 140**	0.05
				Overall Average	0.21

7 Conclusions

In this paper we proposed the first analytic cost model for Spark SQL. The cost model covers the expressiveness of GPSJ queries: a wide family of queries largely used in data analysis. It is the first cost model that implement the Spark computation paradigm keeping into account task pipelining, resource contention and stragglers. Differently from general purpose cost models for MapReduce, our model is aware of the single operations carried out to execute the queries and also relies on a set of statistics on the stored data. The cost model has shown a high accuracy on a large set of different tests and different configurations we tested. The accuracy is good enough to allow the system choose the most effective plan. Our efforts are not devoted to extend the cost model applicability by relaxing some of the assumptions made. In particular, we would remove the assumption about uniform data distribution. A straightforward solution in this direction is to consider histograms to model data distribution. Such an improvement would impact on the reliability of both selections and joins on skewed data.

Since other big data engines (e.g. Impala) adopts similar computation paradigms with different operations and with different optimizations, we would define ad hoc cost models applying our approach to them.

References

[1] The parquet project. parquet.apache.org, 2016.
[2] M. Armbrust et al. Scaling spark in the real world: performance and usability. *PVLDB*, 8(12):1840–1843, 2015.
[3] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.
[4] A. Bhandare et al. Review and analysis of straggler handling techniques. *IJCSIT*, 7(5):2270–2276, 2016.
[5] A. F. Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, 1975.
[6] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *ACM Trans. Database Syst.*, 9(2):163–186, 1984.
[7] S. Ganguly, W. Hasan, and R. Krishnamurthy. *Query optimization for parallel execution*, volume 21. 1992.
[8] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Data Engineering*, pages 209–218, 1993.
[9] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.
[10] P. J. Haas, J. E. Lumby, and C. P. Zuzarte. Selectivity estimation for processing sql queries containing having clauses, 2004. US Patent 6,778,976.
[11] H. Herodotou. Hadoop performance models. *arXiv preprint arXiv:1106.0940*, 2011.
[12] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.
[13] M. T. Özsu and P. Valduriez. *Principles of distributed database systems*. 2011.
[14] 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 *SIGMOD*, pages 165–178, 2009.
[15] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM SIGMOD Record*, 14(2):256–276, 1984.
[16] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.
[17] J. Pullokkaran. Introducing cost based optimizer to apache hive. https://cwiki.apache.org/confluence/download/attachments/27362075/CBO-2.pdf. Online; accessed 18 dEC. 2017.
[18] H. Ron and W. Zhenhua. Design specification of spark cost-based optimization. https://issues.apache.org/jira/browse/SPARK-16026. Online; accessed 18 May 2018.
[19] M. T. Roth, L. M. Haas, and F. Ozcan. *Cost models do matter: Providing cost information for diverse data sources in a federated system*. IBM Thomas J. Watson Research Division, 1999.
[20] P. G. Selinger and M. E. Adiba. Access path selection in distributed database management systems. In *ICOD*, pages 204–215, 1980.
[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.
[22] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *PVLDB*, 8(13):2110–2121, 2015.
[23] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *EDBT*, pages 287–300, 1994.
[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.
[25] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal. Analytical performance models for mapreduce workloads. *International Journal of Parallel Programming*, 41(4):495–525, 2013.
[26] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.
[27] M. Zaharia. *An architecture for fast and general data processing on large clusters*. Morgan & Claypool, 2016.

Lorenzo Baldacci received the PhD degree in 2007, and from 2013 to 2016, he was a researcher at DISI, University of Bologna. He has been publishing in refereed journals and international conferences in the fields of pattern recognition, business intelligence, and bioinformatics. His main research interests are in Business Intelligence and Data Warehousing, and Big Data Analytics.

Matteo Golfarelli is Associate Professor at the University of Bologna teaching Information Systems and Data Mining. His researches in the Business Intelligence area covered most of the design issues related to Data Warehouse systems. He is co-author of the book Data Warehouse Design: Modern Principles and Methodologies. His current research interests include Business Intelligence on non-conventional data and Big Data Analytics.