Multi-objective query optimization in Spark SQL

Michail Georgoulakis Misegiannis
School of ECE, NTU Athens
Athens, Greeece
michailgeorgoulakis@dblab.ece.ntua.gr

Verena Kantere*
School of ECE, NTU Athens
Athens, Greece
verena@dblab.ece.ntua.gr

Laurent d'Orazio*
Univ. Rennes, CNRS, IRISA
Lannion, France
laurent.dorazio@univ-rennes1.fr

ABSTRACT

Query optimization is a challenging process of DBMSs. When tackling query optimization in the cloud, there exists a simultaneous need of providing an optimal physical query execution plan, as well as an optimal resource configuration among available ones. Cloud computing features like resource elasticity and pricing make the process of finding this optimal query plan a multi-objective problem, with the monetary cost being an equally important factor to query execution time. Apache Spark is a popular choice for managing big data in the cloud. However, query optimization in its SQL module (Spark SQL) involves a number of limitations due to the rule-based nature of its optimizer, Catalyst. We propose a multi-objective cost model for the extension of the query optimizer of Apache Spark, aiming to minimize both objectives of query execution time and monetary cost, as well as a methodology for exploring the space of Pareto-optimal query plans and selecting one. The cost model is implemented and tuned, and an experimental study is conducted to validate its accuracy.

CCS CONCEPTS

• **Information Systems** → **Query optimization**.

KEYWORDS

query optimization, cost model, cloud computing, multi-objective optimization, Apache Spark, Catalyst optimizer

ACM Reference Format:
Michail Georgoulakis Misegiannis, Verena Kantere, and Laurent d'Orazio*. 2022. Multi-objective query optimization in Spark SQL. In *International Database Engineered Applications Symposium (IDEAS'22), August 22–24, 2022, Budapest, Hungary*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3548785.3548800

1 INTRODUCTION

Query optimization is the most challenging step of query processing. A query optimizer can either be rule-based, using heuristics to convert the logical query plan to a physical one, or cost-based, using cost functions to compare alternative query plans and return the optimal according to its estimations. The architecture of a cost-based query optimizer consists of three key stages that determine the quality of its predictions [9]: cardinality estimation, cost modeling and plan enumeration.

The majority of works on query optimization aims to minimize query execution time on fixed hardware, which is a valid assumption in the on-premise world. Query processing in the Cloud, however, presents an extra challenge as alternative hardware instances are available. Depending on the resource configuration that is used, the execution might be completely different. Decision making involves deciding on the type of the cluster, the number of instances that will be used, their type, and their characteristics (e.g. RAM size). In such a scenario, a query optimizer should be able to pick both an optimal physical query execution plan, as well as a resource configuration among available hardware instances, thus bridging the gap between query and resource optimization [18].

In order to achieve this, optimizer cost models should be hardware-agnostic, being able to model the behaviour of a query plan in different clusters and systems. A hardware-agnostic cost model could lead to lower costs as well as better resource efficiency [14].

Query optimization is usually associated with minimizing query execution time. The performance of a query, however, can be evaluated in terms of more objectives. Features of the cloud, like resource elasticity and pricing increase the objectives that can be simultaneously optimized in a cloud setting [5]. Adding instances to achieve maximum parallelization during query execution will in general lead to lower execution times, but may also lead to much higher monetary costs, as well as increased energy consumption. Monetary cost is one of the most prevalent query optimization objectives in the cloud [8, 10, 15]. Energy consumption has also been considered lately [13], as cloud providers yearn for reducing energy cost. Other objectives that can be considered in multi-objective query optimization are result precision [16], or data security.

Query optimization in big data systems, which are usually hosted in the cloud, is particularly challenging. As a result, it is necessary that the estimation components of query optimizers (cardinality estimator, cost model) are accurate. One of the most popular big data processing frameworks is Apache Spark, which is widely used in research and industry. However, the query optimizer of Spark's SQL-based component, Spark SQL, has a limited cost model.

In this work, we propose a multi-objective cost model for Spark SQL, for the objectives of query execution time and monetary cost. For the time objective, we adopt an existing single objective cost model [4] for Spark SQL, which shows promising accuracy. We also conduct a detailed experimental study to validate it. For the money objective, we introduce a formula in order to estimate the monetary cost of a query, based on real cloud pricings.

*Both supervisors contributed equally to this work.

IDEAS'22, August 22–24, 2022, Budapest, Hungary

ACM ISBN 978-1-4503-9709-4/22/08…$15.00
https://doi.org/10.1145/3548785.3548800

The cost model receives a query and a set of Spark application configurations as an input, and returns the optimal query plans for each configuration, resulting in a Pareto front. The returned plans present different tradeoffs for the two objectives, and the user can either select one or assign preference weights to the two objectives, and be provided with a single query plan that best meets them.

We also conduct an experimental study on a private cloud environment to validate the cost model accuracy and optimality for a broadly adopted architecture, consisting of Spark, Yarn and HDFS.

Overall, the contributions of this work are the following:

- proposal of a multi-objective cost model for Spark SQL
- introduction of a formula for query monetary cost estimation
- reimplementation and validation of existing cost model for query execution time estimation
- a detailed experimental study on a real private cloud environment
- a user-interactive method for exploring the space of alternative query plans and choosing an optimal one

The rest of this paper is organized as follows. Related work is discussed in Section 2. Section 3 describes the cost model that we implemented. The experimental evaluation and its results follow in Section 4, while Section 5 concludes the paper.

2 RELATED WORK

Optimization in Spark The Spark SQL query optimizer, Catalyst [3], is an extensible optimizer where new rules can be added. However, it is not ideal for cost-based query optimization. It only uses a limited cost model, being unable to provide analytical estimations for the execution time of a query plan. A number of research works have focused on improving specific limitations of the Catalyst optimizer [17].

Although Spark is highly configurable, its manual tuning is time consuming and complex, due to the high-dimensional configuration space. A lot of works provide frameworks for tuning Spark applications [12, 15], in most cases with learned methods. The proposed cost model can be useful in this perspective too, as apart from producing optimal query plans, it can also be used for tuning and comparing different application configurations.

Multi-objective query optimization Karampaglis et al. [6] proposed a bi-objective query cost model suitable for query optimization over a multi-cloud environment. It successfully provides estimates of both the expected execution time and monetary cost.

A number of works have considered multi-objective query optimization in the cloud. Kllapi et al. [7] proposed a technique to optimize dataflow scheduling on a set of containers and form one schedule best meeting user constraints. Their work can also be used for query optimization, when the execution of a query can happen over multiple containers. Multi-objective parametric query optimization [16] takes a different approach to query optimization, which happens before runtime with the use of an exhaustive DP algorithm, and models queries as functions of parameters.

3 COST MODEL AND IMPLEMENTATION

In this work, we propose a cost model for cost-based multi-objective query optimization in Apache Spark. For the objective of query execution time, we adopt a proposed cost model for Spark SQL [4], which we also experimentally evaluated. For the objective of monetary cost, we introduced a cost estimation formula for a given query plan. By combining them, we tackle query optimization as a multi-objective optimization (MOO) problem and use different methods to explore the space of Pareto-optimal query plans.

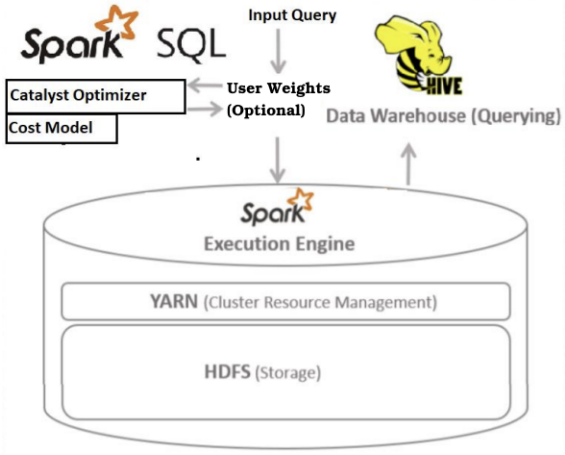

Figure 1: System architecture

	Catalyst C.M.	Proposed C.M.
Query types	ALL	GPSJ
Cost based join selection	YES	YES
Tables and Columns statistics	YES	YES
Considers cluster topology	NO	YES
Based on system disk access time	NO	YES
Takes into account network speed	NO	YES
Analytic estimation of QET	NO	YES

Table 1: Comparison of Catalyst and proposed cost model

3.1 System Architecture

Figure 1 shows the system upon which the cost model operates. The storage layer includes a number of datanodes inside an HDFS filesystem. Data is processed in Spark, and Yarn is the resource negotiator between HDFS and Spark. For data management, data is stored in Apache Hive tables and accessed through Spark SQL queries. As for the optimizer, an extended version of the Catalyst is envisioned, operating cost-based by using the proposed cost model. We implemented the cost model outside Spark and used it manually.

3.2 Cost Model Preliminaries

The proposed single-objective Spark SQL cost model [4] can provide more than Catalyst when it comes to estimating query execution, as highlighted by Table 1. It is based on disk access time and network speed, as disk and network performance is critical in one-pass workloads, like Spark SQL queries. It is a reconfigurable model, that can be tuned for any (homogeneous) cluster and system. It also performs traditional SQL optimizations by collecting table and columns statistics. It is aware of the cluster topology, and takes into account Spark application parameters that influence query execution time. The Spark application parameters considered are the number of Spark executors, and the number of executor cores.

One of its limitations is that it covers the class of Generalized Projection, Selection, and Join (GPSJ) queries, which are a subset of

SQL queries. This means that the use of specific SQL operators like UNION ALL or OUTER JOIN are not supported by the cost model. In addition to that, its use is also limited for homogeneous clusters.

The cost model is capable of analytically estimating the execution time of the five essential RDD transformations that occur in Spark SQL GPSJ queries: (1) Full table scan (2) Full table scan and broadcast (3) Shuffle hash join (4) Broadcast hash join (5) Group by. Each of the transformations is modeled as a function in the cost model code. Precisely modeling these operations is challenging, so the cost model focuses on a set of basic bricks that determine transformations and actions cost, for which it provides cost estimates (Read, Write, Shuffle Read, Broadcast). Each one of these functions receives a data table set as an input (or two, in case of a join operator), as well as the table's cardinality, size, partitions and any filtering predicates. It returns the estimated time needed to execute the transformation, the columns returned, the cardinality and the adjusted size of the table set. The estimated execution time for a query is obtained by summing up the time needed to execute each RDD transformation forming the query physical plan.

3.3 Bi-objective cost model

In the Spark-Yarn-HDFS architecture, query execution involves two parts. A user submits a query, and then specifies some parameters to configure the Spark application. Spark application tuning, although often done empirically, is a complex decision to make, as Spark has a considerable number of parameters that can be configured. One of the most critical parameters is the number of executors that will be allocated for an application. Each Spark executor runs within a Yarn container. Yarn containers are provided by Yarn on demand at the start of each Spark Application. Each one hosts a Spark executor as well as a number of cores that are assigned to it. They are deallocated when the Spark application completes.

The second part of our cost model involves the prediction of the monetary cost of executing a query. In a public IaaS cloud platform, execution of a Spark SQL query requires renting a number of computing instances to host the Spark executors, using them during the runtime of a query and leasing them when execution is completed. As a result, we make monetary cost estimations with the following formula:

$$cost = ci(\#executors) * runtime * hcost(\$/hour)/3600 \quad (1)$$

ci represents the number of computing instances rented as a function of the number of executors and hcost is the hourly cost of using a single computing instance, which we divide by 3600 to scalarize it to seconds. The formula assumes per-second billing.

To define the ci function, we need to assume a Spark application deployment method. In our work, Spark application deployment was well spread, as we assigned each executor on a different computing instance, in order to achieve maximum parallelization. As a result, $ci(\#executors) = \#executors$

In our experimental part, we used prices from Amazon EC2 instances. For an example of a Spark application with 4 executors and 4 executor cores, we rent 4 homogeneous computing instances with 4vCPUs each, for the time needed for the query to execute. If we use a1.xlarge instances ($0.102 hourly cost) and the query takes 20 minutes to complete, its cost is estimated to be about $0.136.

3.4 Query plan enumeration

For a given query, the cost model is able to compare all valid query plans. In our case, two alternative join operators are available and we considere all their combinations, as well as all possible join orderings. We base our monetary cost estimations on the price of renting an a1.medium instance from Amazon EC2, considering cases of renting from one to eight a1.medium instances for query execution. The cost model is easily extensible therefore prices for more computing instance types can be included, to compare different cluster and application scenarios. As a result, our search space for a query involving X join operations involves 2^X join operator combinations, $X!$ join orderings, and 8*N available application configurations, for N computing instance types. For example, for TPC-H Query 3 that involves 3 join operations and considering only one computing instance, results in 384 alternative query plans. The search space can become much larger for more complex queries, however our cost model is able to compare all plans for queries with up to 6 joins with no significant optimization overhead.

For the case of even more complex queries, the search space can be reduced significantly if we do not perform join reordering but keep the join order that the Catalyst produces, and if we introduce distributed query optimization heuristics for join selection [11].

In the experimental part, we also had to overcome the limitation of Catalyst returning a single query plan and not providing alternatives. By reconfiguring certain configuration parameters (disabling broadcast joins, changing broadcast joins thresholds), we were able to produce and compare more query plans.

3.5 Multi-objective optimization

We follow a three-step process for multi-objective optimization.

Step 1 First, we apply single-objective optimization to find the optimal query plan in terms of execution time, for every available system configuration. A single query plan is returned for each configuration. As query monetary cost is dependent on execution time, the fastest plan is also the cheapest, for a fixed system configuration, which explains the reason behind this first step. The different tradeoffs are created by the alternative configurations, and not by alternative query plans inside a certain application setting.

Step 2 The second step is the multi-objective optimization one, as all the query plans from the first step are compared in terms of both objectives. Dominated plans are discarded, and the remaining, Pareto-optimal plans are the output, forming a Pareto front. The selected query plan will determine both the physical query plan that will be executed, as well as the hardware configuration.

Step 3 After the Pareto front is formed, the final step is the query plan selection. As the number of alternative query plans can be large, the process of presenting the alternatives to the user, and assisting him/her to make a decision is challenging. In order to reduce the number of alternatives presented to the user and take into account budget and needs, price and latency filters can be applied. The cost model can also be used in a user-interactive mode, where the user submits preference weights to the objectives and receives a single plan best meeting them. In that case the problem is scalarized to a single-objective one, using the equation:

$$F(x) = \frac{1}{1 + w_1 * f_1(x)} * \frac{1}{1 + c * w_2 * f_2(x)} \quad (2)$$

In order to normalize the values of time and money to the same order of magnitude we use constant c. We set c to an empirical value of 25000, so that 25 seconds of query execution are equivalent with a monetary cost of 0.001 US $. The value for c was selected empirically based on our experiments, in which execution time varied between 50-400 seconds depending on the query and configuration and monetary cost between 0.001 - 0.01 US $ per query. In the case of equal weights, a query plan near the middle of the Pareto curve is selected, meaning that it does not prioritize any of the time-money objectives over the other. The query plan that has the maximum value for F is selected for execution. Before the user submits the preferred weights, he/she can also be provided with a value showing the time-money relationship for the selected weights.

4 EXPERIMENTAL EVALUATION

Methodology The single-objective cost model makes the following assumptions, which our work inherits too:

- It covers the class of GPSJ queries
- It assumes uniform distribution of data in table sets
- It performs single query optimization, assuming a cold start
- It assumes operating on a homogeneous cluster

We evaluated the cost model for a main-memory scenario, assuming that all data fits in memory, as well as intermediate results. We also assumed exclusion of exogenous factors potentially affecting cluster performance, and we calculated a Spark-Yarn initialization overhead in each query, which we did not take into account.

For the evaluation of the cost model, we follow a two step methodology. First, we examine the estimation accuracy of the cost model. Second, we evaluate the optimality of the cost model, aiming to see if it can point to an optimal query plan among alternatives.

The cost model can be used to produce a Pareto front, including plans that offer different time-money tradeoffs. The decision maker can choose the query plan that best suits his/her needs or can use the cost model in a user-interactive mode, and assign weights to the objectives, in order to be provided with a single query plan.

Setup We conducted our experiments in Grid '5000 [1], a large-scale and flexible platform for experiment-driven research in computer science, with a focus on parallel and distributed computing.

In our experiments Grid '5000 was used as a private cloud, where we deployed up to 8 homogeneous computing nodes, each one containing 2 CPUs Intel Xeon E5-2630 v3, 8 cores/CPU, 128GB RAM, 5x558GB HDD, 186GB SSD, and 2 x 10Gb Ethernet. Inside our cluster, we set up an HDFS filesystem where each node worked as a datanode, and our dataset was stored in the SSDs.

Experimental Evaluation The single-objective cost model was reimplemented to model each one of the five RDD transformations. For our experiments, we used TPC-H benchmark queries and its dataset, scaled to 100GBs. The performance of the cost model was validated for many different factors. This allowed us to fine-tune the cost model for our system, and also re-evaluate it and observe its strengths and inaccuracies. Figure 2 shows its performance for a number of TPC-H queries (7.7% error) in a scenario with 4 Spark executors and 4 executor cores. The estimations are quite accurate with the exception of Query 10, where the execution time of a number of broadcast joins is overestimated. Figure 3 shows that the cost model is also able to capture the impact adding Spark executors

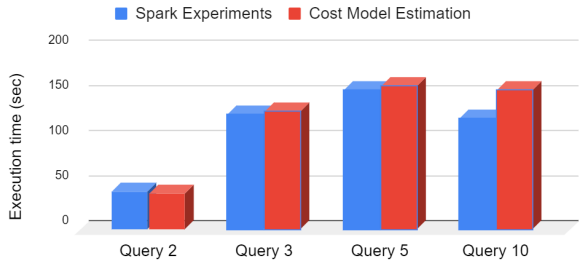

Figure 2: Execution time for different TPC-H queries

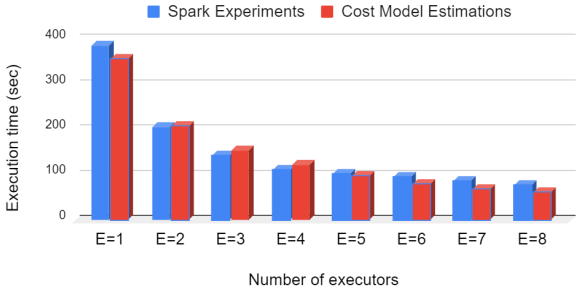

Figure 3: Query Execution times for different number of Spark executors

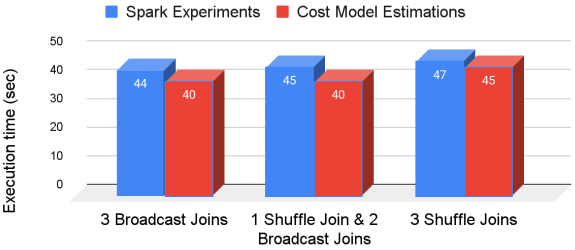

Figure 4: Execution times for three alternative TPC-H Query 2 query plans

has for query execution, with an error rate of 12.1%. The query execution time values are an average for a set of TPC-H queries. A more detailed evaluation of the cost model estimation accuracy can be found in the thesis of Georgoulakis Misegiannis (2021), upon which this paper is based [2].

Inaccuracies mainly have to do with the fact that the cost model does not precisely model Spark data actions and transformations, but only provides estimates on key operations. Furthermore, in some cases it assumes linear relations between different factors not considering heuristics or overkills that occur in Spark. Finally, stochastic processes happening in the cluster might be influencing system characteristics like the read/write throughput or the network speed, causing minor inaccuracies. As a result, tuning the cost model for a given cluster was a challenging task.

In terms of optimality (prediction accuracy), the cost model makes correct predictions for trivial cases. For complex queries, when it did not point to the optimal plan, it was able to at least spot the trends and propose a near-optimal query plan, having small impact in query execution time. Figure 4 shows the performance of the cost model on 3 alternative query plans for TPC-H Query 2 and a configuration involving 4 Spark executors and 4 executor

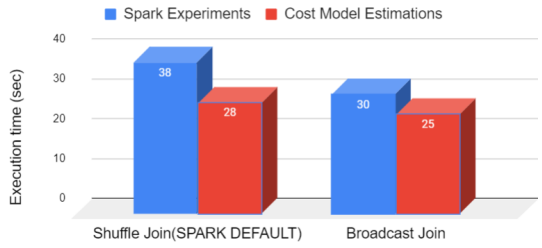

Figure 5: The cost model corrects the Catalyst

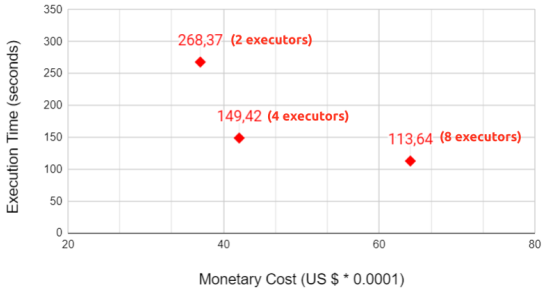

Figure 6: Pareto Front of the optimal query plans

cores. As the table sets considered in Query 2 are quite small, the query plan involving only broadcast joins performs better than the alternatives. Shuffle joining in every case results in a slightly worse execution time, whereas operating the first join with a shuffle join operator and the other two with a broadcast one has similar performance with the optimal case. The cost model gives equally good estimates for the two best execution plans.

The cost model is able to make better choices than the Spark SQL optimizer in many cases. For example, for a simple query involving just a join operation (Fig. 5), we can see that a broadcast hash join is a better choice, resulting in 8 seconds faster query execution than using a shuffle hash join. However, the Catalyst by default chooses to shuffle join these tables. The cost model is able to point to the broadcast hash join as the better option. In the experimental study, the cost model proved that it can be a "relevant first step for turning Catalyst into a fully cost based optimizer", showing significant estimation accuracy while staying on point when it comes to optimality and prediction quality.

When it comes to performing multi-objective optimization, Figure 6 shows the outcome of the extended cost model for a scenario with 3 alternative Spark application configurations with a varying number of executors (2,4 and 8), and 4 executor cores.

The cost model returns the fastest query plan for each application configuration, and then the plans are compared in terms of both objectives. In that case, all three query plans are Pareto optimal and form a Pareto front, as they present different time-money tradeoffs. Thus, the problem is formulated to a multi-objective optimization one. The user can either decide himself/herself which query plan best fits his/her application, or can assign weights of preferences to the objectives in the user-interactive mode of the cost model. For the case of $w_1 = w_2$, the plan with the 4 executors is picked. In case of $w_1 = 2 * w_2$, the plan with the 8 executors is picked, and in case of $w_2 = 2 * w_1$ the selected plan is the one with 2 executors, as the time-money relationship changes each time.

5 CONCLUSION - FUTURE WORK

In this paper we proposed a multi-objective cost model for query optimization in Spark SQL. We built on a promising proposed cost model, which we extended with a formula for estimating the monetary cost of query plans in Spark. The cost model is able to compare query plans providing different time-money tradeoffs, and we also introduce a method for assisting the user into picking a single one. The cost model was implemented and tested, as a detailed experimental study was conducted in a private cloud environment.

In the future, we aim to extend the cost model to consider heterogeneous resources. This will require modeling the execution costs on different hardware, like GPUs. We also aim to explore more optimization goals, like energy consumption, which is a critical objective in green and sustainable data centers. Finally, a long term goal is to try to integrate the cost model into the Catalyst.

ACKNOWLEDGMENTS

Experiments presented in this work were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

[1] 2005. *Grid'5000.* https://www.grid5000.fr/w/Grid5000:Home
[2] 2021. *Multi-objective query optimization for massively parallel processing in Cloud Computing.* https://dspace.lib.ntua.gr/xmlui/handle/123456789/55115
[3] Michael Armbrust, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei A. Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. *ACM SIGMOD* (2015).
[4] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge and Data Engineering* 31 (2019), 819–832.
[5] Michail Georgoulakis Misegiannis, Laurent d'Orazio, and Verena Kantere. 2022. From Cloud to Serverless : MOO in the new Cloud epoch. *EDBT* 2022, 1–4.
[6] Zisis Karampaglis, Anastasios Gounaris, and Yannis Manolopoulos. 2014. A Bi-objective cost model for database queries in a multi-cloud environment. *MEDES* (2014), 109–116.
[7] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. 2011. Schedule optimization for data processing flows on the cloud. In *ACM SIGMOD.*
[8] Trung Dung Le, Verena Kantere, and Laurent D'Orazio. 2020. Dynamic Estimation and Grid Partitioning Approach for Multi-objective Optimization Problems in Medical Cloud Federations. *LNCS* 12410 (2020), 32–66.
[9] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9 (2015), 204–215.
[10] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612.
[11] Sunita Mahajan. 2012. General Framework for Optimization of Distributed Queries. *IJDMS* 4 (06 2012), 35–47.
[12] Dimitra Nikitopoulou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2021. Performance Analysis and Auto-tuning for SPARK in-memory analytics. *DATE* 2021-Febru, 825061 (2021), 76–81.
[13] A. Sathya Sofia and P. GaneshKumar. 2018. Multi-objective Task Scheduling to Minimize Energy Consumption and Makespan of Cloud Computing Using NSGA-II. *Journal of Network and Systems Management* 26, 2 (2018), 463–485.
[14] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. *ACM SIGMOD* 1 (2020), 99–113.
[15] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. (2021), 396–407.
[16] Immanuel Trummer and Christoph Koch. 2017. Multi-objective parametric query optimization. *Commun. ACM* 60, 10 (2017), 81–89.
[17] Alexandru Uta, Bogdan Ghit, Ankur Dave, and Peter Boncz. 2019. [Demo] Low-latency spark queries on updatable data. *ACM SIGMOD* (2019), 2009–2012.
[18] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and Resource Optimization: Bridging the Gap. *ICDE* (2018), 1384–1387.