2020 IEEE 36th International Conference on Data Engineering (ICDE)

ML-based Cross-Platform Query Optimization

Zoi Kaoudi[1,2*] Jorge-Arnulfo Quiané-Ruiz[1,2*] Bertty Contreras-Rojas[3]
Rodrigo Pardo-Meza[3] Anis Troudi[3] Sanjay Chawla[3]

[1]*Technische Universität Berlin* [2]*DFKI GmbH* [3]*Qatar Computing Research Institute, HBKU*
{zoi.kaoudi, jorge.quiane}@tu-berlin.de {brojas, rpardomeza, atroudi, schawla}@hbku.edu.qa

***Abstract*—**
Cost-based optimization is widely known to suffer from a major weakness: administrators spend a significant amount of time to tune the associated cost models. This problem only gets exacerbated in cross-platform settings as there are many more parameters that need to be tuned. In the era of machine learning (ML), the first step to remedy this problem is to replace the cost model of the optimizer with an ML model. However, such a solution brings in two major challenges. First, the optimizer has to transform a query plan to a vector million times during plan enumeration incurring a very high overhead. Second, a lot of training data is required to effectively train the ML model. We overcome these challenges in Robopt, a novel vector-based optimizer we have built for Rheem, a cross-platform system. Robopt not only uses an ML model to prune the search space but also bases the entire plan enumeration on a set of algebraic operations that operate on vectors, which are a natural fit to the ML model. This leads to both speed-up and scale-up of the enumeration process by exploiting modern CPUs via vectorization. We also accompany Robopt with a scalable training data generator for building its ML model. Our evaluation shows that (i) the vector-based approach is more efficient and scalable than simply using an ML model and (ii) Robopt matches and, in some cases, improves Rheem's cost-based optimizer in choosing good plans without requiring any tuning effort.

***Index Terms*—query optimization, machine learning, cross-platform data processing, polystores.**

I. Introduction

Cross-platform systems are fast emerging to address sophisticated and complex needs of data science and business intelligence applications [3], [10], [13], [15], [16], [35]. As they combine multiple data processing platforms to run such complex tasks, optimization is a crucial process. Cost-based optimization is the prominent choice and has already been adopted by cross-platform systems, such as [3], [13]. Yet, cost-based optimization in general suffers from a major weakness: it requires substantial work from administrators to fine-tune the cost model in order to produce efficient query execution plans. This problem gets exacerbated in cross-platform settings where the number of coefficients can easily arrive to hundreds and thus, tuning them is not only challenging but also time consuming. Few works focused on reducing such manual work by running sample queries and calibrating these coefficients [5], [12] or learning them using machine learning [3]. However, these solutions assume a fixed function form, e. g., linear, which may not reflect reality.

An intuitive remedy for this problem is to replace the cost model with an ML model that can predict query runtimes. Although such a direction looks promising it comes with two main challenges. First, this approach requires to transform each query (sub)plan during plan enumeration into a vector and feed it to the ML model. In contrast to works on performance prediction [4], [9], [34], [37], where the plan transformation is required once, this plan transformation can easily be in the order of millions due to the exponential size of the search space. Second, building an ML model for query optimization requires a large number of execution logs. This not only requires finding a large number of real-world queries, but also executing each query using diverse execution plans, which might simply be impractical. For instance, given that a TPC-H query for 200GB takes around 13 minutes to run on Spark in our cluster, running thousand alternative plans of it will take 9 days. Increasing the dataset size may result to an overall time of several months!

We overcome these challenges with Robopt, a novel vector-based cross-platform optimizer that replaces the cost model by an ML model in Rheem [3]. Its novelty lies in basing the entire plan enumeration on a set of algebraic operations that operate on vectors, which are a natural fit to the ML model. This results into two benefits: First, it avoids the high cost of transforming a query (sub)plan into a vector a million times, i. e., each time the ML model is invoked; Second, it allows us to speed up the enumeration process by exploiting modern CPUs for performing vectorization. Figure 1 shows the improvement of using vectors in the plan enumeration (vector-based plan enumeration) over the simple approach of only replacing the cost model with ML (traditional plan enumeration). These results consider two underlying platforms and three tasks: Wordcount (having 6 operators); TPC-H Q3 (having 17 operators); and a synthetic pipeline dataflow (having 40 operators). We observe that using a vector-based plan enumeration is several times faster than its traditional counterpart, even if both approaches explore the same number of plans. This performance difference gets larger when increasing the number of operators or platforms.

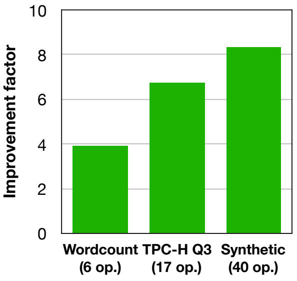


Fig. 1: Benefit of using vectors in the plan enumeration.

In summary, after further elaborating the problem with cost-

[*] Work partially done while at Qatar Computing Research Institute.

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00132

based cross-platform optimization in Section II, we give an overview of Robopt in Section III and present our major contributions in the following sections:

(1) We propose a fine-granular set of algebraic operations that allows Robopt to run the plan enumeration process over query vectors. These operations ease the design of plan enumeration algorithms and enable parallelism. Additionally, we employ an ML-based pruning mechanism that is lossless and reduces the exponential number of plans to quadratic. (Section IV)
(2) To maximize the pruning effect, we propose a *pruning-aware plan enumeration algorithm*, which is built using the pre-defined algebraic operations. For this, we employ a priority metric that determines the order in which partial plan vector enumerations are concatenated. (Section V)
(3) We propose a scalable training data generator that is able to generate large training datasets in a reasonable amount of time using polynomial interpolation. The generator creates synthetic queries, which are representative of real queries, as well as synthetic executions logs, which significantly speed up the training data generation. (Section VI)
(4) We implemented Robopt in Rheem[1] [3], an open source cross-platform system. We show that it not only matches, but also exceeds, the performance of its cost-based optimizer counterpart, with almost no tuning effort. Still, even if we implemented Robopt in Rheem, our approach can be adopted by any other cross-platform system. (Section VII)

We conclude this paper with related work (Section VIII) and some final remarks (Section IX).

II. What's Wrong with Cost-based Optimization?

Cost-based optimization has been at the core of traditional databases since the early days. Following this success story, cross-platform systems such as Rheem [3] and Musketeer [13] employ a cost-based optimization approach for determining on which platform(s) a query has to be executed. However, failing to correctly define or tune the cost model in a cross-platform system can negatively impact performance. This is because of the large diversity in execution operators implementations.

We show this cost modeling problem with a simple experiment: We ran Rheem's cost-based optimizer using a well-tuned (using trial-and-error) and a simply-tuned (using single operator profiling) cost model. We injected the real cardinalities in both cost models to avoid any negative impact in performance due to wrong estimates. We used Spark, Flink, and a standalone Java-based execution engine as underlying platforms. Figure 2 shows the results of this experiment. We observe that a simply-tuned cost model can negatively impact performance by more than one order of magnitude, even if the real cardinalities are used: For example, for Word2NVec, the simply-tuned model forces Rheem to use Java instead of Spark. These results clearly show the importance of having a well-tuned cost model in cross-platform query optimization.

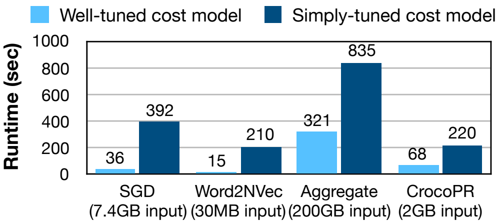


Fig. 2: Impact of a well-tuned cost model in cross-platform optimization: The impact in performance can be substantial - up to a factor of 10.

Unfortunately, arriving to a very well-tuned cost model requires substantial work even from expert administrators. It took us around two weeks of trial-and-error to arrive to the above results. This is because there are many more parameters that need to be tuned compared to monolithic systems. In addition, as platforms are continuously being added the job of the administrator becomes even harder. For example, in Rheem [3], system administrators must define a set of cost functions for *each* operator of a newly added platform and tune the cost model by providing the right coefficients. Although Rheem comes with default values for such coefficients, users should find out the right values in their deployments to gain in performance, as shown above. Few works focused on reducing such manual work by running sample queries and calibrating these coefficients [5], [12]. We have also used machine learning to learn such parameters [21]. However, all these solutions assume a fixed form of function, e. g., linear, which may not reflect reality and thus, hurt performance.

III. Overview

The main goal of cross-platform query optimization is to find the most suitable data processing platforms combination to process an input query in the most efficient way. In detail, the idea is to split a query formed as a logical plan[2] into multiple atomic operators and to find the most suitable platform for each (set of) operator(s), i. e., execution plan, so that the runtime is minimized. Below, we first set the necessary Rheem background and then provide an overview of Robopt.

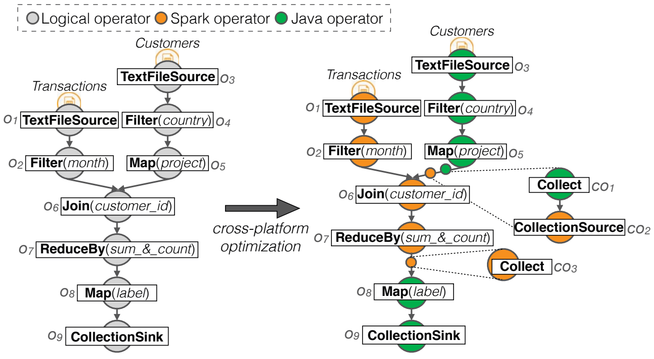
A. Rheem background

Robopt receives as input a logical Rheem query plan, which is a directed dataflow graph. The vertices are *logical* operators and the edges represent the dataflow among the operators. Logical operators are platform-agnostic and define a particular data transformation over their input. Figure 3(a) shows a running example: the logical plan for a query aiming at classifying customers of a certain country according to the total amount of their credit card transactions in the last month.

Robopt outputs an execution plan, which is also a directed dataflow graph but differs from an input logical plan in two aspects. First, its vertices are platform-specific *execution* operators (e. g., SparkMap). Second, it may comprise additional execution operators for data movement among platforms,

[1] https://github.com/rheem-ecosystem/rheem

[2] We follow the query optimization design choice as in [2], where logical optimization is done at the application level. Therefore, logical optimizations, such as join ordering, is beyond the scope of this paper.



(a) Logical plan (b) Execution plan

Fig. 3: Running example: (a) The logical plan of a Join query between two relations; (b) The most efficient execution plan of this logical plan.

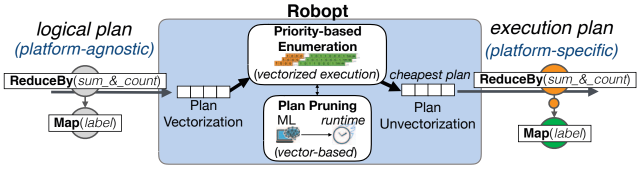


Fig. 4: Robopt obviates the need for fine-level tuning of cost model. Instead, the challenge is to design a representative plan vector, speedup the enumeration using such vectors, and have access to a large training set.

called *conversion* operators, e. g., a SparkCollect operator transforms an RDD into a Java Collection. Conceptually, given a logical plan, an execution plan indicates on which platform each logical operator must be executed. Figure 3(b) shows an execution plan for the logical plan of Figure 3(a) when Spark and Java are the available platforms. This plan exploits both Spark's parallelism for the large transactions dataset and the low latency of Java for the small set of customers. Note that there are additional execution operators for data movement (JavaCollect and SparkCollectionSource).

B. Robopt in a nutshell

At its core, Robopt has an ML model trained to predict the runtime of an execution plan and its vector-encoded plan enumeration. Figure 4 shows the execution flow of Robopt. Once it receives a logical plan, it transforms it into a numerical vector, which we call *plan vector*. A plan vector is an array of features that represent an execution plan. Then, it performs the entire plan enumeration using vectors. This allows Robopt to (i) directly feed a plan vector to the ML model without any transformation and (ii) speed up the enumeration process by using primitive operations and vectorized execution [7].

We have defined a set of algebraic operations that formalizes the vector-based query plan enumeration process. This allows not only for easily exploring different algorithmic alternatives but also for optimization opportunities. Although this set of operators eases the design of plan enumeration algorithms, enumerating the entire search space of possible execution plans remains an exponential problem. We thus propose a pruning technique that reduces the search space from exponential to quadratic. Our pruning technique is guaranteed to not prune any plan vector representing a subplan that is part of the optimal execution plan. It is based on the notion of *boundary* operators, which are the operators that are in the border of a subplan. Moreover, using this set of algebraic operators, Robopt adopts a priority-based plan enumeration algorithm. Using a metric of *priority* it determines the order in which subplans should be concatenated. We propose a priority that maximizes the pruning effect and yields a plan enumeration that exploits the entire search space simultaneously.

Another challenge that Robopt has to overcome is the number of training data required for training the ML model: a large amount of disparate execution plans (data points) together with their runtime (labels) is required. However, it is not only hard to find thousands of queries but also impractical to execute thousands of execution plans for different input dataset sizes. Robopt comes with a scalable training data generator that builds synthetic execution plans of different shapes and for different input data cardinalities. The generator executes only a subset of the plans and applies polynomial interpolation to guess the runtime of the remaining plans.

Last but not least, notice that even though we implemented Robopt in Rheem, it is general enough to be applied to any cross-platform system. The integration effort required for that would be to adapt the input and output as well as the feature vector of Robopt to the plans of the respective system.

IV. Query Planning Vectorization

We propose to vectorize the query planning process so that Robopt is able to efficiently find the execution plan with the lowest estimated runtime (w. r. t. its ML model). The benefit of doing so is two-fold: First, we can directly feed a query plan represented as a vector into an ML model to drastically prune the search space; Second, we can leverage primitive operations and SIMD (Single Instruction Multiple Data) to perform multiple vector operations with a single CPU instruction (vectorized execution [7]). Inspired by relational algebra, we take a principled approach and define a set of algebraic operations that formalizes the vector-based plan enumeration process. Besides the above two benefits of using vectors, using these operations also allow us to: (i) define the enumeration problem in a simple, elegant manner; (ii) concisely formalize our plan enumeration algorithm; (iii) explore different algorithmic alternatives; and (iv) consider parallelization and optimization opportunities inside the plan enumeration itself.

A. Query vector representation

Our entire plan enumeration process relies on a *plan vector* structure, which is an array of features that are representative of an execution plan. In other words, the plan vector v for an execution (sub)plan p is a set of features $v = \{f_1, ..., f_n\}$, where each f_i represents a characteristic of p. We encode four main characteristics of execution plans: (i) their shape, (ii) their operators, (iii) their data movement profile, and (iv) their input dataset. We experimented with different sets

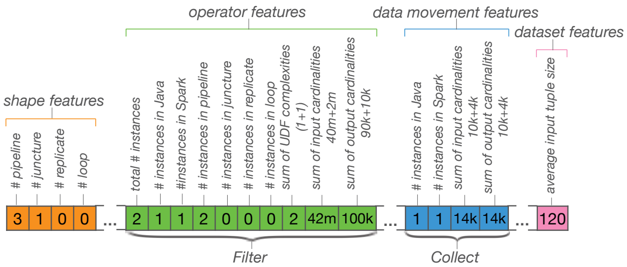


Fig. 5: Snapshot of a plan vector: vector representation of the execution plan in Figure 3(b).

of features and we found that these features are representative enough for building an ML model that accurately orders the plan vectors according to their predicted runtime. We convert these characteristics into a vector using one-hot encoding. Figure 5 illustrates the features for the execution plan of the query shown in Figure 3(b):

(i) *Topology Features.* Taking into consideration diverse query workloads ranging from relational to machine learning, we conclude in four representative plan topologies: *pipeline*, *juncture*, *replicate*, and *loop*. A pipeline is mainly a series of operators receiving one input and having one output, such as the Map and ReduceBy operators in the plan of Figure 3(a). A juncture is a plan with at least one operator receiving two inputs and having a single output, such as the Join operator in Figure 3(a). In contrast to a juncture, a replicate is a plan with at least one operator receiving one input and having two outputs. A loop is basically a plan repeating the execution of a pipeline plan. For example, OLAP queries contain pipeline and junctures topologies, and, the k-means, gradient descent, svm, logistic regression, and pagerank algorithms are mainly pipelines with loops. Note that one plan may contain more than one topology. For instance, the plan in Figure 3(a) has three pipelines and one juncture. We use one feature per topology to encode how many times this topology exists in a query plan as shown in the orange part of Figure 5.

(ii) *Operator Features.* For each logical operator available in Rheem, we first encode the total number of times it appears in the query plan. In addition, we encode the number of times each corresponding execution operator appears as well as in which topology it is located in the plan. For example, the green part of Figure 5 shows that the Filter operator exists twice in our running execution plan (first green feature): once as a Java operator (second green feature); and the other time as a Spark operator (third green feature); both appear in a pipeline topology (fourth green feature). We also encode the CPU complexity of the UDF that an operator contains. We assume four different complexities: logarithmic, linear, quadratic, and super-quadratic. In addition, we consider the input and output cardinality for each operator. Having both input and output cardinalities allows for encoding the operators' selectivity and for knowing how the operators are connected (the structure of the query): it, thus, improves the ML model accuracy.

(iii) *Data Movement Features.* For each conversion operator we encode the number of instances per platform. We additionally encode the input and output cardinality of the conversion operators. For example, the blue features in Figure 5 show the features of the Collect operator in Figure 3(b).

(iv) *Dataset Features.* We include the average tuple size in bytes of the input dataset as the only characteristic of the input dataset (pink feature in Figure 5). The cardinality of the dataset is already taken into account by the input cardinality of the source operators.

Although these features work well with our ML model, one could use other representations, such as structured deep neural networks [27]. However, further investigation on feature extraction is out of the scope of this paper.

B. *Plan enumeration problem*

Having defined the main data structure of our plan enumeration, we now proceed to define the plan enumeration problem. For this, let us first introduce the *plan vector enumeration* structure, which represents the different execution plans for a given logical query (sub)plan. Formally:

Definition 1 (Plan Vector Enumeration): The plan vector enumeration $\mathcal{V} = (s, V)$ of a logical (sub)plan p consists of a scope s, denoting the set of operators ids O_p in p, and a set of plan vectors V, representing a set of execution plans for p, where $\forall v \in V, O_v = O_p = s$.

If p is the initial logical query plan and the scope s of a plan vector enumeration $\mathcal{V}$ contains all the operators of p (i. e., $s = O_p$), then each plan vector $v \in V$ represents a complete execution plan. Formally:

PLAN VECTOR ENUMERATION PROBLEM. *Given a logical query plan p with Ω_p the search space of all possible execution plans, the goal is to find a plan vector enumeration $\mathcal{V}_{min} = (s, V)$ with $s = O_p$ such that $\exists v_{min} \in V$ with $cost(v_{min}) < cost(v_j), \forall v_j \in \Omega_p \wedge v_{min} \neq v_j$.*

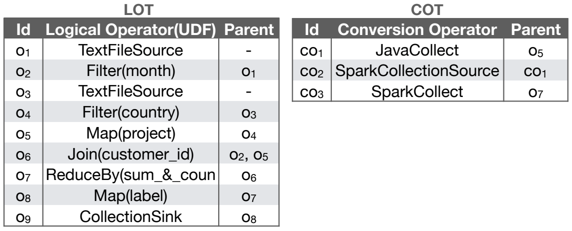
C. *Core operations*

We now introduce three core operations that allow us to explore the entire search space to find $\mathcal{V}_{min}$: vectorize, enumerate, and unvectorize.

(1) vectorize$(p) \rightarrow \overline{v}$: It transforms a logical plan p into an *abstract* plan vector $\overline{v}$. Unlike a plan vector v, $\overline{v}$ does not instantiate the operators per platform. Instead, it simply indicates the possible alternatives per operator by assigning the value -1. For example, for the logical plan in Figure 3(a) the first three cells for the Filter operator would be $2, -1, -1$, respectively, denoting that there are two Filter operators in p which can be executed either on Java or Spark.

(2) enumerate$(\overline{v}) \rightarrow \mathcal{V}$: It receives an abstract plan vector $\overline{v}$ and outputs a plan vector enumeration $\mathcal{V} = (s, V)$, where each $v_i \in V$ represents an alternative execution plan for $\overline{v}$. It must hold that: $s = O_{\overline{v}}$. Essentially, this operation figures out how the operators must be instantiated with their alternative execution operators, i. e., it creates all possible plan vectors (execution plans) for the given abstract plan vector $\overline{v}$.

(3) unvectorize$(v) \rightarrow p$: This is the reverse operation of the vectorize operation. It translates back a query plan p from its plan vector representation v into a format that the system can



LOT

Id	Logical Operator(UDF)	Parent
o_1	TextFileSource	-
o_2	Filter(month)	o_1
o_3	TextFileSource	-
o_4	Filter(country)	o_3
o_5	Map(project)	o_4
o_6	Join(customer_id)	o_2, o_5
o_7	ReduceBy(sum_&_coun	o_6
o_8	Map(label)	o_7
o_9	CollectionSink	o_8

COT

Id	Conversion Operator	Parent
co_1	JavaCollect	o_5
co_2	SparkCollectionSource	co_1
co_3	SparkCollect	o_7

Fig. 6: Logical plan shape: Logical and conversion operators tables, which are required to unvectorize a plan vector into an executable execution plan.

execute. To achieve this, we use two auxiliary structures that capture the exact shape of a query plan: the *Logical Operators Table* (LOT, for short) and the *Conversion Operators Table* (COT, for short). LOT keeps the structure of the logical query plan and is immutable through the entire enumeration process. COT keeps the processing platform switches in a specific execution plan, being, thus, specific to each plan vector. The optimizer reconstructs a query plan p by reading the LOT and COT tables and replacing each operator by its execution operator(s) as denoted by its plan vector v. Figure 6 illustrates these tables for the execution plan in Figure 3(b).

Assuming that $opt(\mathcal{V})$ returns the fastest plan vector of $\mathcal{V}$, one can use these three core operations to define an exhaustive plan enumeration algorithm for a logical plan p as $\mathsf{unvectorize}(opt(\mathsf{enumerate}(\mathsf{vectorize}(p))))$.

D. Auxiliary operations

Although one can use only the core operations to solve the plan vector enumeration problem, enumerating the exponential-size search space Ω_p is impractical. A logical plan with n operators, each having k execution operators, results to k^n possible execution plans. We thus introduce three auxiliary operations that ease the design of more efficient enumeration algorithms by plugging a pruning mechanism, and at the same time enable parallelism: split, iterate, and merge.

(4) $\mathsf{split}(\overline{v}) \rightarrow \overline{V}$: It divides a plan vector $\overline{v}$ into a set of plan vectors $\overline{V}$, which are pair-wise disjoint and their union is equal to the set of operators of $\overline{v}$. Formally: $\forall \overline{v}_i, \overline{v}_j \in \overline{V}$: $O_{\overline{v}_i} \cap O_{\overline{v}_j} = \emptyset$ and $\bigcup_{\overline{v}_i \in \overline{V}} O_{\overline{v}_i} = O_{\overline{v}}$. For example, we could split a plan vector into singleton plan vectors (i. e., query plans containing a single operator) to render the plan enumeration process parallelizable.

(5) $\mathsf{iterate}(\mathcal{V}_1, \mathcal{V}_2) \rightarrow list(\langle v_1, v_2 \rangle)$, $v_1 \in V_1, v_2 \in V_2$: It gets as input two plan enumerations $\mathcal{V}_1 = (s_1, V_1)$ and $\mathcal{V}_2 = (s_2, V_2)$ and returns a list of all possible pairs of plan vectors, where the first element of each pair is from V_1 and the second from V_2. That is, it returns the cartesian product of V_1, V_2.

(6) $\mathsf{merge}(v_1, v_2) \rightarrow v$: It receives two plan vectors v_1, v_2, having equal number of features, and outputs a single vector v. The resulting vector v represents the concatenation of the two subplans corresponding to v_1 and v_2. In detail, we merge two plan vectors as follows. The topology cells of v_1 and v_2 are added up to produce the new cells of v, with the exception of the first cell that is the pipeline, for which we keep the maximum value of the two. The reason behind this is that when concatenating two pipeline subplans the resulted plan is still a single pipeline. For the operators and data movement cells we simply add up their values, while for the input tuple size, we take the max value of the two plan vectors' feature cells. merge is commutative and associative, i. e., the order of its application on plan vectors does not affect the final result.

Example 1 (Plan concatenation with vectors): Figure 7 illustrates the concatenation of two plan enumerations $\mathcal{V}_1, \mathcal{V}_2$ as a result of the iterate and merge operations. $\mathcal{V}_1$ corresponds to the subplan containing the Reduceby followed by the Map in Figure 3(a), while $\mathcal{V}_2$ corresponds to the execution subplan containing just the CollectionSink operator. The concatenated plan enumeration $\mathcal{V}_{12}$ contains the plan vector v_{1a2a} resulted by merging plan vector v_{1a} (consisting of a Spark Reduceby followed by a Spark Map) with plan vector v_{2a} (consisting of a Java CollectionSink). v_{1a2a} consists of all three operators: Reduceby followed by Map and CollectionSink. Similarly, $\mathcal{V}_{12}$ contains also the merged plan vectors v_{1a2b}, v_{1b2a}, v_{1b2b}. We do not illustrate all four vectors due to space limitations.

E. Pruning operation

We now introduce the prune operation, which allows us to reduce the search space significantly.

(7) $\mathsf{prune}(\mathcal{V}, m) \rightarrow \mathcal{V}'$: It receives a plan vector enumeration $\mathcal{V} = (s, V)$ and a model m and outputs a new pruned plan vector enumeration $\mathcal{V}' = (s', V')$, where $V' \subseteq V$ and $s' = s$. The model m is an oracle that given a plan it returns its cost: It can be a cost model, an ML model, or even a pricing catalogue. We instantiate this prune operation with a novel pruning technique that ensures that no subplan that is part of the optimal plan will be pruned. This pruning technique builds upon the notion of *boundary* operators in plan vector enumerations. A boundary operator in a plan vector enumeration with scope s is *adjacent* to some other operator *outside* of s, i. e., belonging to another plan vector enumeration. The rationale is that if two plan vectors in the same plan vector enumeration share the same boundary operators, pruning the plan with the highest cost will not affect the process of finding $\mathcal{V}_{min}$. Formally:

Definition 2 (Boundary Pruning): Let $\mathcal{V} = (s, V)$ be a plan vector enumeration and $s_b \subset s$ be the set of its *boundary* operators. The boundary pruning removes all $v_i \in V$ for which there is another $v_j \in V$ that (i) employs the same platforms for all s_b operators as v_i, and (ii) has a lower cost than v_i w. r. t. model m.

In our case, m is an ML model that predicts the runtime of a plan vector. The prune operator invokes the ML model for each $v_i \in \mathcal{V}$ and compares the predicted runtimes among the plan vectors that employ the same platforms in their boundary operators. It keeps the one with the lowest predicted runtime.

To apply this pruning, we need to retrieve the boundary operators from a plan vector efficiently. For this reason, we introduce the *pruning footprint vector* (see Figure 7). This vector keeps the number of boundary operators per platform

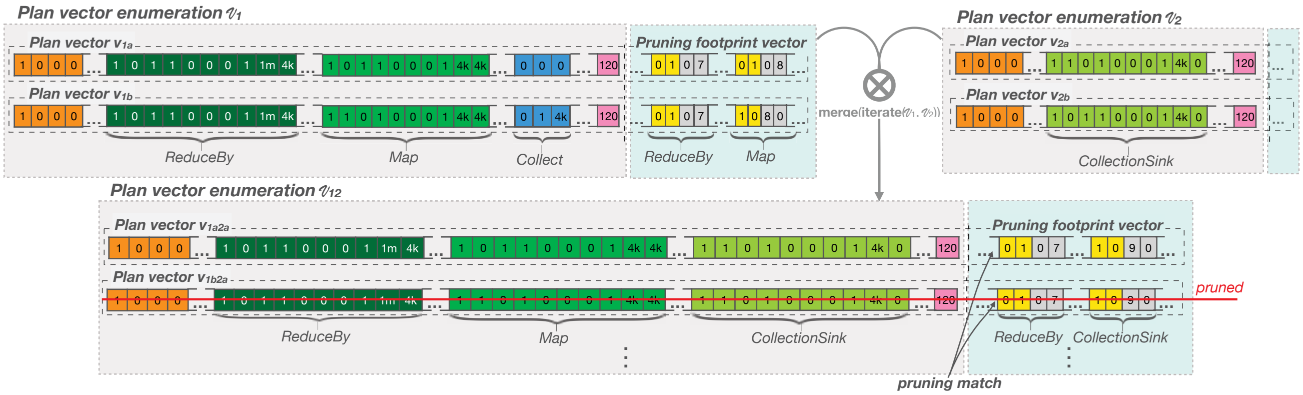


Fig. 7: Plan concatenation: Concatenating two plan vector enumerations $\mathcal{V}_1$ and $\mathcal{V}_2$. In the resulted concatenated plan vector enumeration $\mathcal{V}_{12}$, the plan vector with the highest estimated runtime among the plans that have a pruning match is pruned. Our pruning technique reduces the search space from an exponential to a quadratic size.

(yellow cells) and the operator identifier per platform (grey cells). Whenever there is an exact match on the pruning footprint vector of a set of plan vectors, the optimizer keeps the plan vector with the lowest cost.

Example 2 (Pruning footprint): The plan vector v_{1a2a} in Figure 7 has boundary operators the Spark ReduceBy and Java CollectionSink, whose identifiers from the ROT table are 7 and 9 respectively. The pruning footprints of the plan vectors v_{1a2a} and v_{1b2a} (bottom part of Figure 7) match and hence one can safely prune the one with the highest cost, e. g., v_{1b2a}.

An important property of the boundary pruning is that it is *lossless*, i. e., it does not prune any plan vector of a subplan p that is contained in the optimal plan p_{min} resulting from unvectorize(v_{min}). This is because the boundary pruning technique operates only over those plan vectors having the same pruning footprint vector: non-boundary operators in a plan vector do not affect the cost of consequent plan vectors.

V. Pruning-Aware Plan Enumeration

We now describe how Robopt uses the above algebraic vector operations to find the execution plan with the lowest estimated runtime for a given logical query plan. The literature for traversing the plan search space can mainly be divided into two strategies: bottom-up or top-down. In Robopt, this means starting either from the source or the sink operators, respectively. However, both strategies are pruning-oblivious and hence might not be very efficient in reducing the search space. We thus introduce a plan enumeration algorithm that is driven by the *priority* of each partial plan vector enumeration $\mathcal{V}$: it chooses to concatenate (and prune) first those plan vector enumerations with high priority. We argue that it is highly beneficial if such priority is in accordance with the used pruning technique (the boundary pruning in our case).

A. Plan vector enumeration priority

We define the priority of a plan vector enumeration $\mathcal{V}$ as the cardinality of the plan vector enumeration resulted from the concatenation of $\mathcal{V}$ with its children, i. e., the total number of plan vectors in the resulting plan vector enumeration. The higher the cardinality is, the higher the priority is. Formally:

Definition 3 (Plan Vector Enumeration Priority): Given a plan vector enumeration $\mathcal{V} = (s, V)$ and its children plan vector enumerations $\mathcal{VC} = \{\mathcal{V}_1, ..., \mathcal{V}_m\}$, where m is the number of $\mathcal{V}$'s children, we define the *priority* of $\mathcal{V}$ as: $|V| \times \Pi_{V_i \in \mathcal{VC}} |V_i|$.

Example 3 (Plan Vector Enumeration Priority): Assume that the Join operator (OP$_6$) in Q_c of Figure 3(a) has three alternative execution operators (in Java, Spark, and Postgres), i. e., $|V_{OP_6}| = 3$, while the ReduceBy operator (OP$_7$) has only two alternative execution operators (in Java and Spark), i. e., $|V_{OP_7}| = 2$. Then, the priority of the plan vector enumeration for Join is equal to $|V_{\mathsf{OP}_6}| \times |V_{\mathsf{OP}_7}| = 6$.

By using this priority, our algorithm is neither top-down nor bottom-up: it rather exploits the entire search space simultaneously leading to greater pruning opportunities. The rationale behind this priority definition is twofold. First, the more inner (i. e., non-boundary) operators exist in a plan vector, the more effective our pruning technique. Second, the more plan vectors with the same boundary operators in a plan vector enumeration, the larger the pruned search space. Still, one could change the priority to result into the standard top-down or bottom-up strategies (e. g., a priority defined as the distance between the last operator of a subplan and the sink operator results into a bottom-up traversal).

B. Priority-based plan enumeration

Algorithm 1 depicts our priority-based plan enumeration process, which uses the pre-defined algebraic operations. It takes as input a logical query plan and an ML model that predicts the runtime of execution plans and passes through four main phases: (i) it vectorizes and splits the input query plan into singleton abstract plan vectors (Line 2); (ii) it enumerates each singleton plan vector and sets their priority (Lines 3–5), (iii) it concatenates the resulting singleton plan vector enumerations based on their priority (Lines 6–13), and (iv) it outputs the plan with the minimum estimated runtime according to the ML model (Line 18). The reason of first

Algorithm 1: Priority-based plan enumeration

```
1  Function MAIN
       Input: $p$, ML //logical query plan, ML model
       Output: $xp_{min}$ //optimal execution plan
2      $\overline{V} = \mathsf{split}(\mathsf{vectorize}(p))$;
3      foreach $\overline{v}_i \in \overline{V}$ do
4          $\mathcal{VQ}$.enqueue(enumerate($\overline{v_i}$));
5      $\mathcal{VQ}$.setPriorities();
6      while $\mathcal{VQ}.size() > 1$ do
7          $\mathcal{V} = \mathcal{VQ}$.dequeue();
8          foreach $\mathcal{V}_c \in getChildren(\mathcal{V})$ do
9              $P_V$ = iterate $(\mathcal{V}, \mathcal{V}_c)$;
10             $\mathcal{V}' = \emptyset$ ;
11             foreach $\langle v_1, v_2 \rangle \in P_V$ do
12                 $v$ = merge $(v_1, v_2)$;
13                 $\mathcal{V}'$.append($v$);
14             $\mathcal{V}$ = prune ($\mathcal{V}'$, ML);
15         $\mathcal{V}$.setPriority();
16         $\mathcal{VQ}$.enqueue($\mathcal{V}$);
17         $\mathcal{VQ}$.updatePriority($getParents(\mathcal{V})$);
18     $xp_{min}$ = unvectorize $(getOptimal(\mathcal{V}, \mathtt{ML}))$;
```

creating singleton plan vector enumerations (Lines 2–5) is to guide the entire plan enumeration process based on the plan vector enumerations' priority. We detail below the plan enumeration process and we discuss how we build a robust ML model for query runtime prediction in the next section.

Let us now explain how Robopt incrementally concatenates the plan vector enumerations until a final plan vector enumeration for the entire input query plan is created. It first dequeues a plan vector enumeration from the priority queue $\mathcal{VQ}$ (Line 7) to concatenate it with its children's plan vector enumerations to get a new plan vector enumeration $\mathcal{V}$ (Lines 8-13). Note that in case of equal priority between two plan vector enumerations, we take the one introducing the less number of new boundary operators. If the equality continues, we consider the order with which they were entered in the queue. In more detail, it uses the iterate operation (Line 9) to go through all possible pairs of plan vectors (Line 11) between two plan vector enumerations and merge them using the merge operation (Line 12). After each child's concatenation, it uses the boundary ML-based pruning operation (see Section IV-E) to significantly reduce the number of plan vectors in $\mathcal{V}$ (Line 14). Once the concatenation with all $\mathcal{V}$'s children's is done, Robopt computes the priority of the resulting plan vector enumeration $\mathcal{V}$ and updates the priority of its parents' plan vector enumeration (Lines 15-17). It terminates when queue $\mathcal{VQ}$ has only one plan vector enumeration, which is the final plan vector enumeration, i. e., $\mathcal{V} = \mathcal{V}_{min}$. As a final step, it feeds all plan vectors in $\mathcal{V}$ to the ML model to estimate their runtime, outputs the fastest one according to the ML model and uses the unvectorize operation to return the corresponding execution plan (Line 18). As our pruning is lossless, our algorithm can determine the optimal execution plan with respect to the ML model.

Note that our fine-granular operations allow us to easily change Algorithm 1. For example, our boundary-operator pruning is similar to the concept of *interesting sites* in distributed relational query optimization [18], which is an instance of *interesting properties* [30]. Hence, on can easily extend the enumeration algorithm to account for other interesting properties by simply modifying the prune operation. In addition, by changing the priority metric to be equal to the distance of an operator from the source or sink, one can easily design a bottom-up or top-down enumeration algorithm.

Lemma 1: The boundary pruning reduces the search space from $O(k^n)$ to $O(nk^2)$, where n is the number of operators and k the number of platforms.

Proof: Let p be a pipeline logical plan having n operators each being able to be executed in k platforms. We will show that using the pruning of Definition 2 we have $|\Omega_p| = (n-1) \times k^2$. The plan enumeration of p is composed of $n-1$ steps in total, where in each step we concatenate a single operator to the previous composed subplan. In the first step of the plan enumeration, we compose plans of two operators and therefore pruning is not possible. This results in k^2 possible plan vectors. For each next step, we have 2 boundary operators and thus, k^2 combinations of platforms for these boundary operators. For each such combination, we keep only the plan with the lowest estimated time (see Definition 2). We, thus, end up with $(n-1) \times k^2$ total execution plans. □

VI. Training Data At Scale

Although ML-based query optimization is a promising direction, its effectiveness strongly depends on the training data. In our case, a large amount of disparate execution plans (data points) together with their runtime (labels) is required. However, it is not only hard to find thousands of queries but also impractical to execute thousands of execution plans for different input dataset sizes in a reasonable amount of time.

In contrast to works that utilize previously executed workloads as training data [36], we assume that there is no or very limited access to query workloads. Therefore, we accompany our ML-based query optimizer with a scalable training data generator (TDGen). The generator allows for producing a representative training dataset in three different ways: (i) users can provide their real query workload and let the generator to create a specified number of training data that resembles their query workload, (ii) users can specify only the shapes (e. g., pipeline- or loop-style as specified in Section IV) and the maximum size (i. e., number of operators) of the queries they expect to have, and (iii) users can let the generator to exhaustively create training data of all possible query shapes given the maximum number of operators only. Overall, TDGen proceeds in two main phases: the *job generation*, which is basically generating the training data points, and *log generator*, which assigns labels to each training data point. We explain each of these two phases below.

A. Job generation

TDGen starts by creating synthetic logical plans of different shapes and sizes. In case a user provides a real query workload, TDGen extracts the shapes and maximum size of the given queries and creates similar synthetic query plan templates. It then populates these query plan templates with logical operators so that the input (output) of each operator complies with its parent's output (children's input, respectively). It can populate a query plan template with all possible combinations of logical operators or with a randomly chosen set of logical operators. The result is a set of logical plans.

Next, TDGen uses our priority-based plan enumerator to create a number of execution plans for each logical plan. The difference is that it uses a different pruning operation to prune the search space. It uses as a heuristic the number of platform switches. The idea is to prune an execution plan that has more than β (by default $\beta = 3$) number of platform switches as this is very unlikely to be an optimal execution plan in practice. Our algebraic operations defined in Section IV allowed us to easily reflect these changes. Then, TDGen instantiates each execution plan with different configuration profiles, which provide input dataset sizes, UDF complexities and selectivities of the involved execution operators. An execution plan with a specific configuration profile specifies a runnable job.

B. Log generation

Running all jobs is practically infeasible, because they are not only too many but also often have a large input cardinality and thus, take very long time to terminate. To overcome this challenge TDGen proceeds in two steps. First, given a set of jobs $\mathcal{J}$ with the same structure (execution plan), it actually runs only a subset $\mathcal{J}_r \subset \mathcal{J}$. This set $\mathcal{J}_r$ includes: (i) all the jobs with small input cardinalities, (ii) few jobs with medium and large input cardinalities, and (iii) only jobs with low and high UDF complexity. Second, it imputes the runtime of each $j \in \mathcal{J}_i$, with $\mathcal{J}_i = \mathcal{J} \setminus \mathcal{J}_r$, via polynomial interpolation using the already executed jobs $\mathcal{J}_r$. We use piecewise polynomial interpolation with degree 5[3] in order to learn the function that fits the points of $\mathcal{J}_r$. Figure 8 exemplifies this log generation process: the blue points are executed jobs of plans consisting of 6 operators with different input cardinality ($\mathcal{J}_r$) while the green line predicts the runtime of all unknown the jobs ($\mathcal{J}_i$). TDGen uses interpolation for all different types of execution plans.

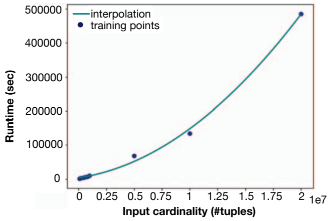


Fig. 8: Interpolation to predict jobs runtime.

VII. Evaluation

We integrated Robopt, which comprises more than 6,000 lines of Java code, in Rheem[4]. We compare Robopt with the previous cost-based optimizer of Rheem (RHEEMix) [3], [21], [22] and with the approach of simply replacing the cost model with an ML model without using vectors in the plan enumeration (Rheem-ML). We used the same pruning strategy in both baselines to have a fair comparison. We aim at answering the following: (i) How efficient is Robopt compared to these two baselines (Section VII-B)? Can Robopt match the ability of the cost-based optimizer to (ii) choose the best platform for performing a given query (Section VII-C1)? and (iii) boost performance by using multiple platforms (Section VII-C2)?

TABLE I: Number of enumerated subplans.

(#ops, #plats)	**(5,2)**	**(5,3)**	**(5,4)**	**(5,5)**	**(20,2)**	**(20,3)**	**(20,4)**	**(20,5)**
w pruning	36	117	272	525	156	522	1232	2,400
w/o pruning	60	724	4,090	15,618	10^6	10^9	10^{12}	10^{14}

A. Setup

We ran all our experiments on a cluster of 10 machines, each with: 2 GHz quad-core CPU, 32 GB memory, 1 TB storage, 1 Gigabit network, and 64-bit Ubuntu OS. We used Java 9 (with vectorization enabled), Spark 2.4.0, Flink 1.7.1, GraphX 1.6.0, Postgres 9.6.2, and HDFS 2.6.5. We, in purpose, considered data processing platforms that are quite similar in terms of capability and efficiency as underlying platforms: Having two platforms with quite similar performance makes it harder for an optimizer to choose the fastest, which better evaluates the effectiveness of Robopt. We used each of these platforms with their default settings and 20 GB of max memory. We set up Rheem with its default settings and hand-tuned its cost-based optimizer as doing so produced a more precise cost model than using the cost learner explained in [3].

For building the ML model in Robopt, we tried linear regression, random forests, and neural networks and found random forests to be more robust. Still, one can plug any regression algorithm. Further investigating models for runtime prediction is out of the scope of our paper. We generated training data generated with TDGen by giving as input three different topology shapes (pipeline, juncture, and loop) and a maximum number of operators equal to 50. Generating the training data and building the ML model took only a couple of days. No further tuning was then required.

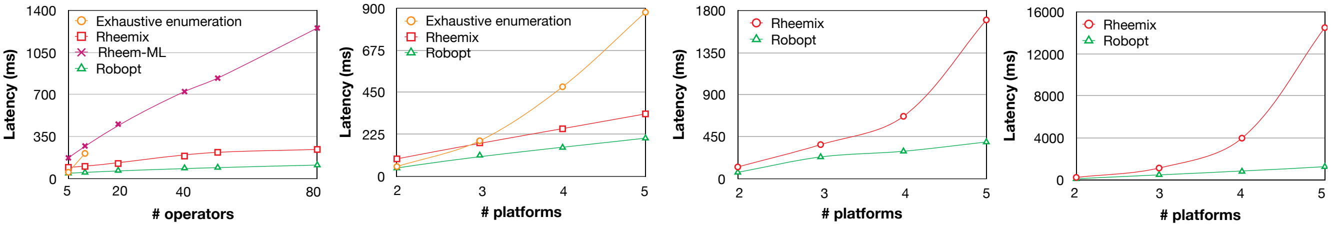
B. Robopt efficiency and scalability

We first evaluate the efficiency and scalability of Robopt. We measured the latency of the optimization process for increasing number of operators and platforms: from the time that we receive the logical plan until the time we have the execution plan. In this evaluation, we used synthetic logical plans consisting of an increasing number of operators and assume all operators are available in $2-5$ platforms. Note that having plans with many operators is not unusual, especially for complex workflows, e. g., a simple query of finding similar words contains 26 operators.

Figure 9(a) shows the latency of the plan enumeration with increasing number of operators for two platforms. We observe that Robopt scales better than all baseline systems. This is thanks to its vector-based plan enumeration that speeds up processing by performing primitive operations. Rheem-ML

[3] Degree 5 was giving us better accuracy without sacrificing runtime.

[4] https://github.com/rheem-ecosystem/rheem



(a) 2 platforms (b) 5 operators (c) 20 operators (d) 80 operators

Fig. 9: Robopt efficiency and scalability: Its vector-based plan enumeration is much more efficient than simply replacing the cost model with an ML model (Rheem-ML in the figure). In addition, it is more scalable than the cost-based in both the number of operators in a plan and the number of underlying platforms.

confirms this aspect: keeping the traditional plan enumeration and calling the ML model as an external black box leads to up to 11x worse performance. We observed that Rheem-ML took 47% of the time just vectorizing the subplans when calling the ML model. Invoking the ML model for both approaches took only 10% of the optimization time on average. In fact, an optimizer based on our vectorized enumeration but without pruning (exhaustive enumeration in Figure 9(a)) is also faster than RHEEMix and Rheem-ML for 5 operators.

Figures 9(b), 9(c), and 9(d) demonstrate the scalability of Robopt. Note that we do not report the Rheem-ML baseline as we already showed that it is outperformed by all other methods with only two platforms. We do not report the exhaustive enumeration's numbers for 20 and 80 operators either, because it cannot run at such a scale due to the large number of enumerated plans (see Table I). This shows the efficiency of our pruning technique. Overall, we observe Robopt scales gracefully and its superiority becomes more apparent as the number of operators and platforms increases. For example, for 80 operators with 3 platforms Robopt takes 0.5s, which is half the time of RHEEMix (1.1s). This performance difference increases exponentially with the number of platforms: it can be more than one order of magnitude different. This is because there is a huge overhead of concatenating and pruning subplans when using objects rather than when merging and matching vectors using primitive methods.

We additionally show the benefit of our enumeration algorithm using the traditional top-down and bottom-up enumeration strategies as baselines. We could easily implement top-down and bottom-up by simply adjusting the priority function to measure the distance from a given operator to the sources and sink, respectively. Figure 10 illustrates the results for plans with increasing number of joins. In the worst case, Robopt performs similarly to the top-down enumeration (i. e., for small queries with 2 joins). As the number of joins and the number of platforms increases, the benefit of our enumeration is evident: up to 2.5 improvement factor over the top-down and up to 8.5 improvement factor over the bottom-up. This is thanks to our priority-based enumeration strategy, which in most of the cases results in enumerating less number of subplans.

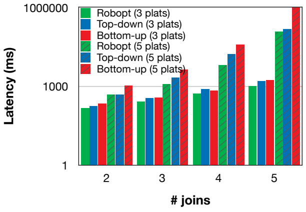


Fig. 10: Effectiveness of priority-based enumeration.

TABLE II: Real queries and datasets.

Query	Description	#operators	Dataset (size)
WordCount	count distinct words	6	Wikipedia (30MB – 1TB)
Word2NVec	word neighborhood vectors	14	Wikipedia (3MB – 3GB)
SimWords	clustering of similar words	26	Wikipedia (3MB – 3GB)
TPC-H Q1	aggregate query	7	TPC – H (1GB – 1TB)
TPC-H Q3	join query	18	TPC – H (1GB – 1TB)
Kmeans	clustering	7	USCensus1990 (36MB – 1TB)
SGD	stochastic gradient descent	6	HIGGS (740MB – 1TB)
CrocoPR	cross-community pagerank	22	DBpedia (200MB – 1TB)

Note that, even if we enabled vectorization in Java, the improvement factor was only 60% on average. This is a well-known problem in Java and there are already efforts that aim at improving this. We expect that in the future our approach can benefit even more with more efficient SIMD support.

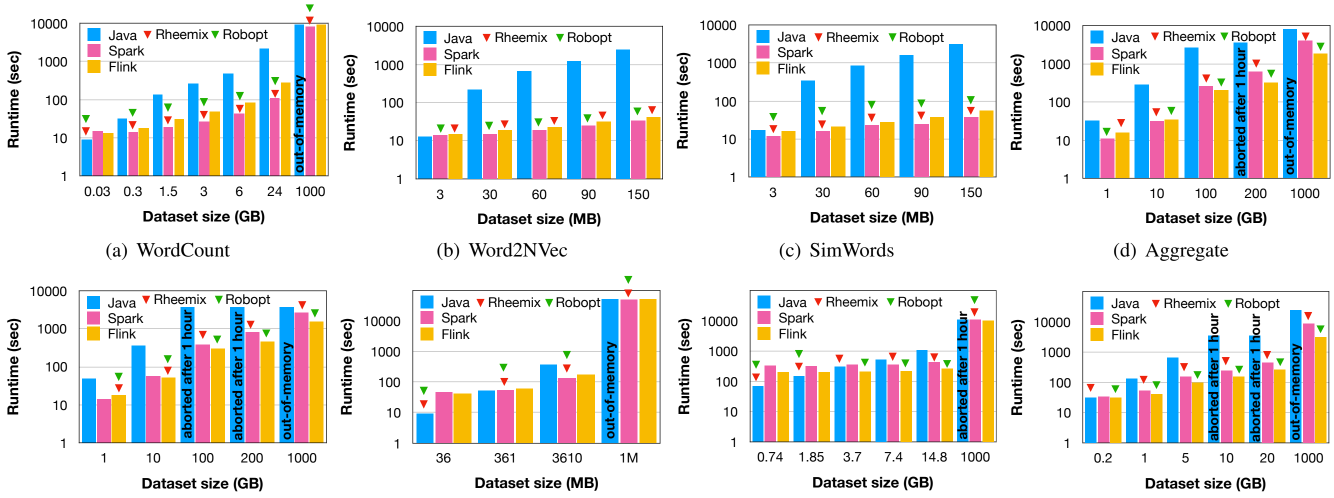
C. Robopt effectiveness

Although Robopt's optimizer is quite efficient, one may wonder how well it performs in choosing execution plans as the execution time is the lion's share in the total query runtime. For this reason, we evaluate the effectiveness of Robopt in comparison with Rheem's cost-based optimizer (RHEEMix) in two different execution modes: single platform execution and multi-platform execution. In this evaluation, we considered a broad range of real data analytics queries and datasets (Table II) in the fields of text mining, relational analytics, machine learning, and graph mining, also used in [3], [13], [21]. Most of these are in Rheem's repository[5]. As in this experiment we are solely interested in performance, and not in the correctness of the results, we varied the datasets size up to 1TB by replicating the input data.

1) Choosing a single platform: We first evaluate the success of Robopt in choosing a single platform to run a query and compare it with RHEEMix. For this, we set Rheem to choose only one platform for executing a query.

Figure 11 presents the results for the eight queries of Table II and for increasing dataset sizes. Each bar in the graphs shows the runtime of each underlying platform and the red and green triangles show the choice of Rheem and Robopt, respectively. We observe that in most cases Robopt

[5] https://github.com/rheem-ecosystem/rheem-benchmark



(a) WordCount (b) Word2NVec (c) SimWords (d) Aggregate

(e) Join (f) K-means (g) SGD (h) CrocoPR

Fig. 11: Single-platform execution mode: Robopt matches or improves the performance of the execution plans derived from Rheem's well-tuned cost model. The red and green triangles denote the choices of Rheem and Robopt, respectively.

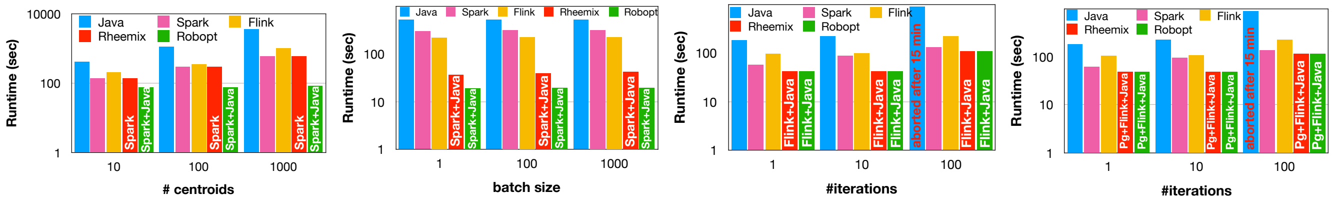
chooses either the same platform with RHEEMix or a faster one. For example, for the WordCount (Figure 11(a)), both Robopt and RHEEMix choose the fastest platform, i. e., Java for the smallest dataset size and Spark for the rest. For Word2NVec (Figure 11(b)), in contrast to RHEEMix, Robopt always chooses the fastest platform, i. e., it chooses Spark for all dataset sizes while Rheem chooses Flink instead. The only case where Robopt chose to run a query on a slower platform than the one chosen by RHEEMix is for Aggregate with 10GB input. However, the performance difference in this case is only 3sec: Spark executes the query in 31s and Flink in 34s. More interestingly, we observe that Robopt chooses the fastest platform in 84% of the cases while RHEEMix does so in only 43% of the cases. Table III summarizes the results of Figure 11 by showing the maximum and average runtime difference of Robopt and RHEEMix from the optimal runtime (diff*) for each query. Among the cases where Robopt does not choose the fastest platform, its maximum performance difference from the optimal is only 343sec (for SGD with 1TB input). In contrast, RHEEMix's maximum performance difference from the optimal goes up to 90min (for CrocoPR with 1TB input). Interestingly, when removing these two cases (SGD for Robopt and CrocoPR for RHEEMix), we observe that Robopt's average overhead for each query is in the order of milliseconds while for RHEEMix ranges from seconds to minutes. These results confirm the caveats of a cost model and the benefit of an ML-based model in a cross-platform setting: (i) the parameters to tune are so many that makes manual tuning very hard, (ii) the cost formulas RHEEMix uses are linear, which do not reflect the real interactions of the different parameters, (iii) ML models are more efficient in capturing complex relationships resulting in better performance.

TABLE III: Summary results of Figure 11 in seconds.

Query	$\text{diff}^*_{Rheemix}$		diff^*_{Robopt}	
	max	avg	max	avg
WordCount	0	0	0	0
Word2NVec	8	5	1	0.2
SimWords	0	0	0	0
Aggregate	305	73.8	3	0.6
Join	1152	317.2	4	0.8
K-means	5	1.25	0	0
SGD	343	120	343	63
CrocoPR	5412	828	0	0

2) Combining multiple platforms: We proceed to evaluate if Robopt is able to combine multiple platforms to speed up the execution of a single query. For this experiment, we compared again Robopt's performance with RHEEMix's performance as well as with the runtimes of running the query on a single platform. In contrast to previous experiment, we show the results only for those queries where Robopt or RHEEMix selected multiple platforms: namely K-means, SGD, and CrocoPR. The rest of the queries perform better on a single platform and hence Robopt and RHEEMix selected a single platform for such queries. We thus omit showing the results for these queries as we already showed them in the previous section. We also varied different parameters for each of these three queries to better evaluate Robopt: the number of centroids for K-means, the batch size for SGD, and the number of iterations for CrocoPR. To further stress the necessity of cross-platform, for CrocoPR we show two cases: the data is initially stored in HDFS (CrocoPR-HDFS) and the data is stored in Postgres but need to be cleaned from null values (CrocoPR-PG). As Postgres is not suitable of running Pagerank for CrocoPR-PG, using different platforms is mandatory.

Figure 12 show the results. We observe that Robopt outperforms any single-platform execution for all three queries. Surprisingly, we observe that Robopt also outperforms RHEEMix for K-means and SGD. For K-means (Figure 12(a)), in contrast to RHEEMix that chooses to execute the entire plan on Spark, Robopt is able to find a plan that combines Spark with Java. This plan uses Java to keep the centroids and broadcast



(a) K-means (b) SGD (c) CrocoPR-HDFS (d) CrocoPR-PG

Fig. 12: Multiple-platform execution mode: Robopt matches the performance of the execution plans derived from Rheem's well-tuned cost model; Robopt even outperforms Rheem's execution plans, such as for SGD.

them to the Spark operators. Broadcasting the centroids as a collection is more beneficial than broadcasting them as an RDD. This minor change in the plan leads Robopt to 7x better performance than RHEEMix: the benefit increases with the number of centroids. For SGD (Figure 12(b)), although Robopt selects the same platforms (Spark and Java) as RHEEMix, it is able to outperform RHEEMix by factor 2 on average. This is because Robopt produces a slighter different plan: it does not place a cache operator before the Sample operator, which is inside a loop. Although caching the data before a loop seems a rational decision, it is not always beneficial as it strongly depends on the operator(s) used after the cache. In this case, SGD uses the ShufflePartitionSample operator, which shuffles a partition the first time it is called and then sequentially retrieves data points from that partition. For this, this operator keeps a flag of whether it is the first time that is executed. Therefore, having a cache operator before this sampling operator causes the loss of the sample operator's state. This results into a high overhead for shuffling the data in each iteration. Such cases are very hard to spot and even harder to model in a cost formula. However, Robopt is able to find such cases by observing patterns in the execution logs.

For CrocoPR (Figures 12(c) and 12(d)), we observe that Robopt produces exactly the same execution plan as RHEEMix. When the data is in HDFS, this plan uses Flink to preprocess the data and encode them as integers. Then, as the data is compressed, Java performs faster the pagerank algorithm as it has lower overhead than Spark or Flink. This leads both Robopt and RHEEMix to run up to twice faster than Spark (the fastest single-platform execution in this case). As Postgres is not suitable for running pagerank, for CrocoPR-PG we used as baseline the common practice of moving the data out of Postgres to a single platform to perform the task. Robopt and RHEEMix filter out the null values in Postgres and then move the rest into Flink for the preprocessing and Java for pagerank, similary to the previous case.

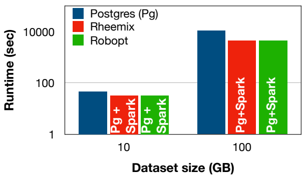


Fig. 13: Join query.

Finally, we ran the Join query in a different scenario in order to further evaluate the effectiveness of Robopt. We now assume that the input data of this query is stored on Postgres. In this case we additionally compare Robopt with the execution of Join on Postgres, which is the obvious platform to run this query as the data already resides on it. The results are quite interesting (Figure 13). First, we observe that Robopt significantly outperforms Postgres, even though the input data is stored there: it is up to 2.5x faster than Postgres[6]. This is because it simply pushes down the projection into Postgres and then moves the data into Spark to perform the join and aggregation, thereby leveraging the parallelism of Spark. Note that Robopt has the same performance as RHEEMix as both produce the same execution plan.

Conclusion. All above results confirm the high efficiency and scalability of Robopt as well as its superiority over all baselines. Robopt can match, and sometimes exceed, RHEEMix's success in choosing good execution plans in both single- and multiple-platform settings. This is in addition to being more efficient and scalable than RHEEMix. Finally, recall that Robopt requires much less tuning effort. It took us only a couple of days of automatic training data generation. In contrast, RHEEMix took us a couple of weeks to correctly tune its cost-model and achieve its most efficient results.

VIII. Related Work

There are few cross-platform optimizers in the literature, either rule-based [35] or cost-based [1], [3], [8], [13], [24]. However, they all require substantial effort from the system administrators to produce efficient execution plans. The rest of the cross-platform systems, such as [6], [10], do not provide any optimizer but rely on the users to specify the platforms to use or the desired execution plan.

Most works that explore the use of ML in query optimization focus on using ML as a black box in one module of the query optimizer: e. g., selectivity estimation [14], [17], [23], [28], [32] and join ordering [20], [25]. Other works use ML to aid the cost model of the optimizer [31], [38]. Robopt differs from these works as it uses ML as an integral piece of the *cross-platform optimization* and embeds ML into the core of optimization itself. More recently Neo [26] puts all these together into an end-to-end learned query optimizer. For its plan enumeration it uses a DNN-guided learned best-first search strategy, which cannot always find the optimal plan due to its heuristic nature. In contrast, our plan enumeration is guaranteed to find the optimal plan w. r. t. the ML model while being both efficient and scalable.

[6] We use cold caches in all cases.

Researchers have also used ML in other aspects of systems. For instance, few works use ML to ease the task of DBAs, such as tuning DBMSs configurations [29], [33] or building the correct indices [19]. There are also works that use ML for performance prediction [4], [9], [11], [27], [34], [37]. However, the goal of all these works is different from Robopt as they need to predict the runtime of each query only once.

IX. Conclusion

We presented Robopt, an ML-based optimizer whose goal is to find an efficient plan by combining multiple data processing platforms. Robopt uses ML as a core component to guide the plan enumeration and prune the search space. More importantly, it uses a set of algebraic operations on plan vectors to speed up the plan enumeration process via vectorized execution. Its plan enumeration algorithm is pruning-aware and hence more efficient. As training data is crucial for any ML-based solution, we accompanied Robopt with a scalable training data generator which uses polynomial interpolation to generate not only synthetic queries but also impute their execution time. We evaluated Robopt by comparing it with Rheem's cost-based optimizer, and showed how it can improve performance with almost no tuning effort.

Acknowledgments

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

References

[1] D. Agrawal, M. L. Ba, L. Berti-Équille, S. Chawla, A. K. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki. Rheem: Enabling Multi-Platform Task Execution. In *SIGMOD*, pages 2069–2072, 2016.

[2] D. Agrawal, S. Chawla, A. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and M. J. Zaki. Road to freedom in big data analytics. In *EDBT*, pages 479–484, 2016.

[3] D. Agrawal, B. Corteras-Rojas, S. Chawla, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. T. S. Thirumuruganathan, and A. Troudi. Rheem: Enabling Cross-Platform Data Processing – May The Big Data Be With You! *PVLDB*, 11(11):1414–1427, 2018.

[4] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based Query Performance Modeling and Prediction. In *ICDE*, pages 390–401, 2012.

[5] F. Andrès, F. Kwakkel, and M. L. Kersten. Calibration of a DBMS Cost Model with the Software Testpilot. In *CISMOD*, pages 58–74, 1995.

[6] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*, pages 221–230, 2018.

[7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[8] K. Doka, N. Papailiou, V. Giannakouris, D. Tsoumakos, and N. Koziris. Mix 'n' match multi-engine analytics. In *BigData*, pages 194–203, 2016.

[9] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *SIGMOD*, pages 337–348, 2011.

[10] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. *PVLDB*, 8(12):1908–1911, 2015.

[11] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*, pages 592–603, 2009.

[12] G. Gardarin, F. Sha, and Z. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System. In *VLDB*, pages 378–389, 1996.

[13] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: All for one, one for all in data processing systems. In *EuroSys*, pages 1–16. ACM, 2015.

[14] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, pages 1477–1492, 2015.

[15] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. In *CIDR*, 2013.

[16] Z. Kaoudi and J.-A. Quiané-Ruiz. Cross-Platform Data Processing: Use Cases and Challenges. In *ICDE*, pages 1723–1726, 2018.

[17] M. Kiefer, M. Heimel, S. Breß, and V. Markl. Estimating Join Selectivities Using Bandwidth-optimized Kernel Density Models. *PVLDB*, 10(13):2085–2096, 2017.

[18] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *TODS*, 25(1):43–82, 2000.

[19] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.

[20] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR*, abs/1808.03196, 2018.

[21] S. Kruse, Z. Kaoudi, J.-A. Quiané-Ruiz, S. Chawla, F. Naumann, and B. Contreras-Rojas. RHEEMix in the Data Jungle – A Cross-Platform Query Optimizer. arXiv: 1805.03533 https://arxiv.org/abs/1805.03533, 2018.

[22] S. Kruse, Z. Kaoudi, J.-A. Quiané-Ruiz, S. Chawla, F. Naumann, and B. Contreras-Rojas. Optimizing Cross-Platform Data Movement. In *ICDE*, pages 1642–1645, 2019.

[23] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *CASCON*, pages 53–59, 2015.

[24] J. Lucas, Y. Idris, B. Contreras-Rojas, J. Quiané-Ruiz, and S. Chawla. RheemStudio: Cross-Platform Data Analytics Made Easy. In *ICDE*, pages 1573–1576, 2018.

[25] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM Workshop*, pages 3:1–3:4, 2018.

[26] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[27] R. C. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11):1733–1746, 2019.

[28] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *DEEM Workshop*, 2018.

[29] A. Pavlo et al. Self-Driving Database Management Systems. In *CIDR*, 2017.

[30] 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.

[31] P. Shivam, S. Babu, and J. Chase. Active and Accelerated Learning of Cost Models for Optimizing Scientific Applications. In *VLDB*, pages 535–546, 2006.

[32] K. Tzoumas, A. Deshpande, and C. S. Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB*, 2011.

[33] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*, pages 1009–1024, 2017.

[34] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *NSDI*, pages 363–378, 2016.

[35] J. Wang et al. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*, 2017.

[36] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a Learning Optimizer for Shared Clouds. *PVLDB*, 12(3):210–222, 2018.

[37] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *PVLDB*, 6(10):925–936, 2013.

[38] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *VLDB*, pages 289–300, 2005.