Optimizing Applications for Cross-Platform Execution

Sebastian Kruse*[2], Zoi Kaoudi[1], Jorge-Arnulfo Quiané-Ruiz[1], Sanjay Chawla[1], Felix Naumann[2], and Bertty Contreras[1]

[1]Qatar Computing Research Institute (QCRI), HBKU
[2]Hasso Plattner Institute (HPI)

Abstract

In pursuit of efficient and scalable data analytics, the insight that "one size does not fit all" has given rise to a plethora of specialized data processing platforms and today's complex data analytics are moving beyond the limits of a single platform. To cope with these new requirements, we present a cross-platform optimizer that allocates the subtasks of data analytic tasks to the most suitable platforms. Our main contributions are: (i) a mechanism based on graph transformations to explore alternative execution strategies; (ii) a novel graph-based approach to efficiently plan data movement among subtasks and platforms; and (iii) an efficient plan enumeration algorithm, based on a novel enumeration algebra. We extensively evaluate our optimizer under diverse real tasks. The results show that our optimizer is capable of selecting the most efficient platform combination for a given task, freeing data analysts from the need to choose and orchestrate platforms. In particular, our optimizer allows certain tasks to run more than one order of magnitude faster than on state-of-the-art platforms, such as Spark.

1 The Power of Many

Companies and organizations are continuously collecting many diverse datasets for an ever-increasing number of (often complex) data analytics. As a result, the modern data analytic ecosystem is characterized by (i) increasing query/task[1] complexity, (ii) heterogeneity of data sources, and (iii) a proliferation of data processing platforms (*platforms*, for short). For example, IBM reported that North York hospital needs to process 50 diverse datasets, which are on a dozen different internal systems [6]. While, on one hand, specialized platforms excel at different kinds of analytics, on the other hand the use of multiple platforms to enhance functionality and improve performance is becoming a common practice [53, 38, 20]. For example, as illustrated by Figure 1(a) for a simple `SELECT` task[2] and the popular WordCount task, different platforms have divergent performance. However, combining platforms, as shown in Figure 1(b), to train a logistic regression model using the stochastic gradient descent algorithm can substantially improve performance.

These are just a few examples of the need for cross-platform task processing; more and more emerging applications (e. g., from data warehouses, business intelligence, healthcare, and machine learning) also need to perform data analytics over several platforms. Overall, we identify four situations in which an application needs support for cross-platform task processing: (i) Depending on the input datasets and parametrization, some applications perform best on different platforms (*platform independence*); (ii) Other applications benefit from using

*Work partially done while interning at QCRI.

[1]Henceforth, we use the term task without loss of generality.

[2]This task selects tuples from TPC-H's `lineitem` table by the `receipt_date` attribute.

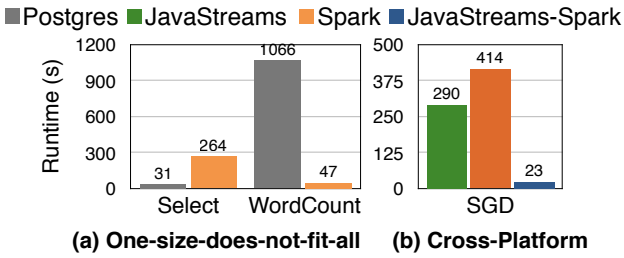


Figure 1: Need for a cross-platform system.

multiple platforms throughout a single task (*opportunistic cross-platform*); (iii) In other cases, applications require to use multiple platforms because the input data is stored on multiple data stores (*polystores*); and (iv) Some applications require to use multiple platforms, because the platform where the data resides, e. g., PostgreSQL, cannot perform the incoming task, e. g., a machine learning task (*mandatory cross-platform*).

The current practice to cope with cross-platform requirements is to write ad-hoc programs to glue different specialized platforms together. This approach is not only expensive and error-prone, but it also requires to know the intricacies of the different platforms to achieve high efficiency. While some systems have appeared with the goal of facilitating platform integration [29, 1, 2, 9, 11], they require a high level of expertise and the goal of carrying out efficent and transparent *cross-platform task processing* has remained elusive.

The research community has recently recognized the need for a systematic solution that enables efficient cross-platform task processing [28, 15, 54, 14, 46]. The holy grail would be to replicate the success of DBMSs to cross-platform applications: users formulate platform-agnostic data analytic tasks and an intermediate system decides on which platforms to execute each task with the goal of minimizing cost (e. g., runtime or monetary cost). Recent research works have taken first steps towards that direction [53, 42, 32, 14, 28]. Nonetheless, they all lack important aspects. For instance, none of these works considers the planning of complex data movement among platforms. Additionally, most focus on specific applications [53, 42, 14].

We believe that a *cross-platform optimizer* is the key component for such a systematic solution and it shall thus be the focus of this paper. Concretely, we consider the problem of *finding the set of platforms that minimizes the total execution cost of a given task*. Now the reader might suppose that a rule-based optimizer is an appropriate solution to this problem: one might simply decide to execute a task on a centralized/distributed platform when the input dataset is small/large, respectively. However, this approach is neither practical nor effective. First, setting rules at the task level implicitly assumes that all the operations in a task have the same computational complexity and input cardinality. Such assumptions do not hold in practice, though. Second, the cost of a task on any given platform depends on many input parameters, which hampers a rule-based optimizer's effectiveness as it oversimplifies the problem. Third, as new platforms and applications emerge, maintaining a rule-based optimizer becomes difficult and cumbersome. For these reasons, we pursue a cost-based approach instead.

Challenges. Devising a cost-based optimizer for cross-platform settings is quite challenging for many reasons: (i) the optimization search space grows exponentially with the number of atomic operations of the given data analytic task; (ii) the optimizer must not only consider the cost of executing the task but also of moving data across platforms; (iii) the data movement might be complex itself and requires proper planning and optimization; (iv) platforms vastly differ w. r. t. their supported operations and processing abstractions; and (v) the optimizer must be extensible to accommodate new platforms and emerging application requirements.

Contributions. We propose a cross-platform optimizer that is the first to tackle all of the above challenges. The main idea is to split a single task into multiple atomic operators and to find the most suitable platform for each operator (or set of operators) so that its cost is minimized. Our optimizer is an integral component in RHEEM [7], an open source cross-platform system[3]. After giving a brief background on RHEEM (Section 2) and an overview of our optimizer

[3]Link omitted because of the double-blind policy.

(Section 3), we make the following major contributions:

(1) We introduce flexible graph-based operator mappings, which allow to employ platforms even if they have only limited operator coverage, and plan inflation, which is a very compact representation of the execution plan search space (Section 4).

(2) We model data movement for cross-platform optimization as a new graph problem, which we prove to be NP-hard, and propose a novel, efficient algorithm to solve it (Section 5).

(3) We propose a new algebra to enumerate executable cross-platform plans for a given data analytics task. In addition, we formulate a new lossless pruning technique to make the enumeration highly efficient (Section 6).

(4) We extensively evaluate our optimizer under diverse analytic tasks using real-world datasets and show its substantial performance benefits in all four cross-platform cases (Section 7).

Eventually, we discuss related work (Section 8) and close with a summary and directions for future work (Section 9).

2 SystemX Background

Before delving into the details of our cross-platform optimizer, let us briefly outline RHEEM [7] so as to establish the optimizer's context. RHEEM acts as a unifying abstraction layer for co-existing data processing platforms with the goal of enabling cross-platform task processing. Applications do not submit data analytics tasks directly to a processing platform. Instead, they issue their tasks to RHEEM, which dynamically executes them on the platforms it deems to be the most suitable. Overall, RHEEM is composed of two main components: the *cross-platform optimizer* and the *executor*. After explaining how applications express their tasks, we briefly outline both components. For more details about RHEEM' architecture, we would like to refer the interested reader to [7].

Rheem plan. Applications send tasks to the system as RHEEM *plans*, which are essentially directed acyclic data flow graphs (DAG). The vertices are RHEEM *operators* and the edges represent the data flow among the operators. Specific Loop operators further accept feedback edges, thus enabling iterative data flows. Conceptually, the data is flowing from source operators through the DAG and is manipulated in the operators until it reaches a sink operator. RHEEM operators are platform-agnostic and define a particular kind of data transformation over their input, e. g., a Reduce operator aggregates all input data into a single output.

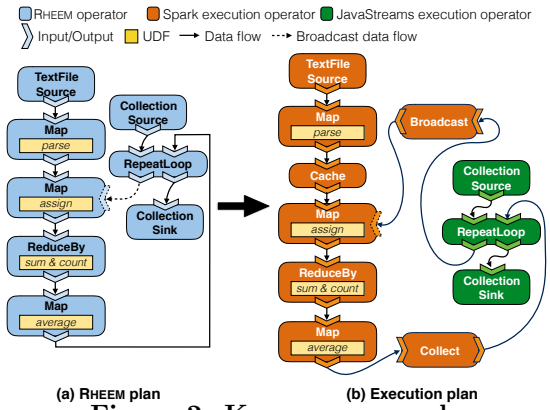


Figure 2: K-means example.

Example 1 *Figure 2(a) shows a* RHEEM *plan for k-means. Data points are read via a* TextFileSource *and parsed using a* Map*, while the initial centroids are read via a* CollectionSource*. The main operations of k-means (i. e., assigning the closest centroid to each data point and computing the new centroids) are repeated until convergence (i. e., the termination condition of* RepeatLoop*). The resulting centroids are output in a collection. To convey a tangible picture of the context in which our optimizer works, we present the actual code for this* RHEEM *plan in Appendix A.*

Cross-platform optimizer (*focus of this paper*). Given a RHEEM plan, the optimizer determines how to best execute this plan. That is, it produces an *execution plan* by selecting one or more data processing platforms as exemplified in Figure 2(b). The execution plan is again a data flow graph with two differences from a RHEEM plan: the vertices are (platform-specific) *execution operators*;

and the execution plan may comprise additional execution operators for data movement among platforms (e. g., Broadcast). In the next section, we give an overview of this component as well as how it transforms the k-means plan into an execution plan; and details of its functionality are presented in the subsequent sections.

Cross-platform executor. RHEEM' executor orchestrates the execution of the generated execution plan on the selected platforms. For that purpose, it has a decoupled platform-specific driver for each available processing platform. In addition, this component monitors the health of the ongoing execution and collects statistics to improve the cost-model of the optimizer.

3 Cross-Platform Optimizer Overview

Unlike traditional relational database optimizers, our cross-platform optimizer does not aim at finding good operator orderings. This is because, emphasizing flexibility, RHEEM does not define a fixed set of operators, which allows our optimizer to be agnostic of operator types. Therefore, we assume that all operator orderings as well as all algorithmic decisions are taken at the application level. Instead, the goal of our optimizer is to select one or more data processing platforms to execute a given RHEEM plan in the most efficient manner. The main idea is to split a single task into multiple atomic operators and to find the most suitable platform for each operator (or set of operators) so that its total cost is minimized.

Example 2 *Figure 2(b) shows the execution plan produced by our cross-platform optimizer for the k-means* RHEEM *plan when Spark [5] and Java Streams [8] are the only available platforms. This plan exploits Spark's high parallelism for the large input dataset and at the same time benefits from the low latency of Java Streams for the small collection of centroids. Also note the three additional execution operators for data movement (*Broadcast, Collect*) and to make data reusable (*Cache*). As we show in Section 7, such hybrid execution plans often achieve higher performance than plans with only a single platform.*

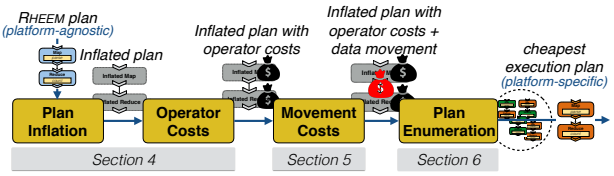


Figure 3: Cross-platform optimization pipeline.

Figure 3 depicts the workflow of our optimizer. At first, given a RHEEM plan, the optimizer passes the plan through a plan enrichment phase (Section 4). In this phase, the optimizer first *inflates* the input plan by applying a set of mappings. These mappings determine how each of the platform-agnostic RHEEM operators can be implemented on the different platforms with execution operators. The result is an *inflated* RHEEM *plan* that can be traversed through alternative routes. That is, the nodes of the resulting inflated plan are RHEEM operators with all its execution alternatives. Additionally, the optimizer *annotates* the inflated plan with estimates for both the intermediate result sizes and the costs of executing each execution operator. Then, the optimizer takes a graph-based approach to determine how data can be moved most efficiently among execution operators of different platforms and again annotates the results and their costs to the plan (Section 5). Finally, it uses all these annotations to determine the optimal execution plan via an enumeration algorithm (Section 6). This algorithm is centered around an enumeration algebra and a highly effective, yet lossless pruning technique. Eventually, the resulting execution plan can be enacted by RHEEM' cross-platform executor.

4 Plan Enrichment

When our optimizer receives a RHEEM plan, it has to do some preparatory work before it can explore alternative execution plans. We refer to this phase as *plan enrichment*. Concretely, our optimizer (i) determines all eligible platform-specific execution operators for each RHEEM operator (Section 4.1); and

(ii) estimates the execution costs for these execution operators (Section 4.2).

4.1 Inflation

While RHEEM operators declare certain data processing operations, they do not provide an implementation and are thus not executable. Therefore, our optimizer *inflates* the RHEEM plan with corresponding execution operators, each providing an actual implementation on a specific platform. A basic approach to determine corresponding execution operators for RHEEM operators are mapping dictionaries, such as in [37, 28]. This approach allows only for 1-to-1 *operator mappings* between RHEEM operators and execution operators – more complex mappings are precluded, though. However, different data processing platforms work with different abstractions: While databases employ relational operators and Hadoop-like systems build upon `Map` and `Reduce`, special purpose systems (e. g., graph processing systems) rather provide specialized operators (e. g., for the PageRank algorithm). Due to this diversity, 1-to-1 mappings are often insufficient and a flexible operator mapping technique is called for.

Graph-based operator mappings. To this end, we define operator mappings in terms of *graph mappings*, which, in simple terms, map a subgraph to a substitute subgraph. We formally define an operator mapping as follows.

Definition 1 (Operator mapping) *An operator mapping $p \rightarrow s$ consists of the graph pattern p and the substitution function s. Assume that p matches the subgraph G of a given* RHEEM *plan. Then, the operator mapping designates the substitute subgraph $G' := s(G)$ for this match via the substitution function.*

Usually, the matched subgraph G is a constellation of RHEEM operators and the substitute subgraph G' is a corresponding constellation of execution operators. However, G may comprise execution operators; and G' may be a constellation of RHEEM operators. The latter cases allow our optimizer to be able to choose among platforms that do not natively support certain operators. Figure 4(a) exemplifies some mappings for our k-means example.

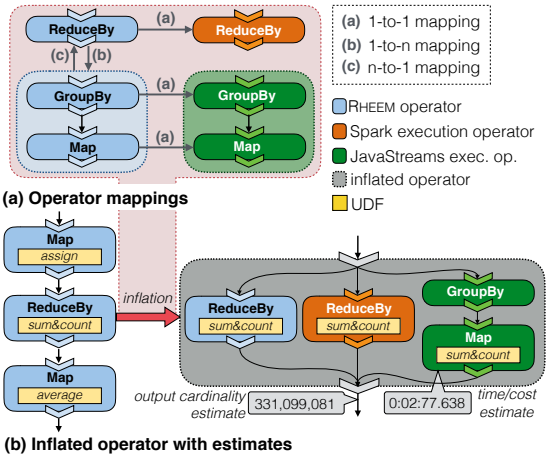


Figure 4: Rheem plan enrichment.

Example 3 (1-to-n mapping) *In Figure 4(a), the 1-to-n mapping transforms the* ReduceBy RHEEM *operator to a constellation of* GroupBy *and* Map RHEEM *operators, which in turn are transformed to Java Streams execution operators.*

In contrast to dictionary-based 1-to-1 mapping approaches, our graph-based approach provides a more powerful means to derive execution operators from RHEEM operators. Our approach also allows us to break down complex operators (such as a PageRank operator) and map it to platforms that do not support it natively.

Inflated operator. It is important to note that our optimizer does not apply operator mappings by simply replacing matched subgraphs G by their substitute subgraphs G', as doing so would cause two insufficiencies: First, this strategy would always create only a single execution plan, thereby precluding any cost-based optimization. Second, that execution plan would be dependent on the order in which the mappings are applied, because once a mapping is applied, other relevant mappings might become inapplicable. We overcome both insufficiencies by introducing *inflated operators* in RHEEM plans. An inflated operator replaces a matched subgraph and

comprises that matched subgraph *and* all the substitute graphs. The original subgraph is retained so that operator mappings can be applied *in any order*; and each inflated operator can contain multiple substitute graphs, thereby accounting for *alternative* operator mappings. Ultimately, an inflated operator expresses alternative subplans inside RHEEM plans.

Example 4 (Operator inflation) *Consider again our k-means example whose plan contains a* ReduceBy *operator. Figure 4(b) depicts the inflation of that operator. Concretely, the* RHEEM ReduceBy *operator is replaced by an inflated operator that hosts both the original and two substitute subgraphs.*

After our optimizer has exhaustively applied all its operator mappings, the resulting *inflated* RHEEM *plan* defines *all* possible combinations of execution operators to enact the original RHEEM plan – but without *explicitly* materializing those combinations. In other words, an inflated RHEEM plan is a highly compact representation of all execution plans.

4.2 Cost Estimation

To compare alternative execution plans entailed by an inflated RHEEM plan, our optimizer estimates and annotates cost values (e.g., expected execution time or monetary costs) to each execution operator (see Figure 4). This is done in two steps: At first, the optimizer estimates the cardinalities of the data flows (i. e., the edges) in the inflated RHEEM plan. Second, it assesses the costs of each execution operator based on its configuration as well as its input and output cardinalities. The optimizer performs above two steps without any user interaction as it comes with default functions and values for each of the above metrics.

Still, it is noteworthy that cardinality and cost estimation are extremely challenging problems – even in highly cohesive systems, such as relational databases, which have detailed knowledge on execution operator internals and data statistics [41]. Facing these challenges, RHEEM has several ways to improve its default cardinality and cost estimators: (i) users can provide hints to the optimizer whenever it is obvious (for instance, there should always be k centroids in a k-means plan); (ii) cardinality and cost estimators can be learned and refined from historical execution data; and (iii) our optimizer can also employ progressive optimization [43] whenever the two former options are not applicable. Nonetheless, cost estimation is a complex and complementary topic that goes beyond the scope of this paper. To put our optimizer into perspective, we briefly discuss some aspects in Appendices B and C, though. In any case, the operator costs form the basis to identify the most promising execution plan in the following steps.

5 Data Movement

Selecting optimal platforms for an execution plan might require to move data across platforms and transform them appropriately for the target platform. This leads to an inherent trade-off between choosing the optimal execution operators and minimizing data movement and transformation costs. Our optimizer must properly explore this trade-off to find the overall optimal execution plan.

However, planning and assessing communication is challenging for various reasons. The major challenge is that data movement can be complex; e. g., it might involve several intermediate steps to connect a source operator to multiple target operators. Furthermore, there might be several alternative data movement strategies, so it is necessary to determine the optimal one. Finally, the costs of each strategy must be assessed so that our optimizer can explore the above mentioned trade-off between selecting optimal execution operators and minimizing data movement costs.

To address these challenges, we represent the space of possible communication steps as a *channel conversion graph* (Section 5.1). This graph representation allows us to model the problem of finding the most efficient communication path among execution operators as a new graph problem: the *minimum conversion tree* problem (Section 5.2). We then devise a novel algorithm to efficiently solve this graph problem (Section 5.3).

5.1 Channel Conversion Graph

The channel conversion graph (*CCG* for short) is a graph whose vertices are data structure types (e. g., an RDD in Spark) and whose edges express conversions from one data structure to another. As with the operator mappings (see Section 4.1), RHEEM provides the CCG, which can be extended by developers if needed, e. g., when adding a new platform to RHEEM. Before formally defining the CCG, let us first explain how we model data structures (*communication channels*) and data transformation (*conversion operators*).

Communication channel. Data can flow among operators via communication channels (or simply *channels*), which form the vertices in the CCG. This can be for instance an internal data structure or stream within a data processing platform, or simply a file. For example, the yellow boxes in Figure 5 depict the standard communication channels considered by our optimizer for Java Streams and Spark. Note that communication channels can be *reusable*, i. e., they can be consumed multiple times, or non-reusable, i. e., once they are consumed they cannot be used anymore. For instance, a file is reusable, while a data stream is usually not.

Conversion operator. In certain situations, it becomes necessary to convert communication channels from one type to another, e. g., it might be necessary to convert an RDD to a file. Such conversions are handled by conversion operators, which form the edges in the CCG. Conversion operators are in fact regular execution operators: For example, RHEEM provides the SparkRDDToFile operator, which simply reads the RDD and writes it to a file. Intuitively, the associated communication costs are incurred neither by the RDD nor the file but by the conversion operator. Thus, given a cardinality estimate of the data to be moved, the optimizer computes the conversion costs as regular execution operator costs.

Channel conversion graph. We can now integrate communication channels and their conversions in a graph.

Definition 2 (Channel conversion graph) *A* CCG *is a directed graph* $G := (C, E, \lambda)$*, where the set of vertices* C *contains the channels,* E *comprises the directed edges indicating that the source channel can be converted to the target channel, and* $\lambda\colon E \to O$ *is a labeling function that attaches the appropriate conversion operator* $o \in O$ *to each edge* $e \in E$*.*

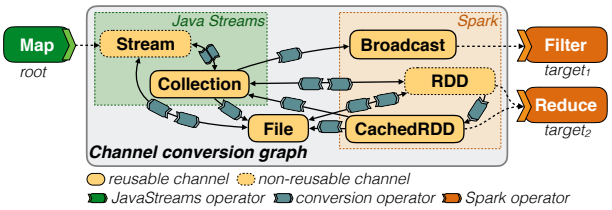


Figure 5: Example channel conversion graph along with source and target operators from different platforms.

5.2 Minimum Conversion Tree Problem

CCGs allow us to model the problem of planning data movement as a *graph problem*. This approach is very flexible: If there is *any* way to connect execution operators via a sequence of conversion operators, we will discover it. Unlike other approaches [**?**, e. g.,]]gog2015musketeer,ires-bigdata, developers do not need to provide conversion operators for all possible source and target channels. It is therefore much easier for developers to add new platforms to RHEEM and make them interoperable with the other platforms. Let us further motivate the utility of CCGs for data movement with a concrete example.

Example 5 *Figure 5 shows an excerpt of* RHEEM*'s default CCG that is used to determine how to move data from a* JavaMap *execution operator (*root*) to a* SparkFilter *(*target$_1$*) and a* SparkReduce *execution operator (*target$_2$*), respectively. While the* root *produces a Java stream as output communication channel, target$_1$ and target$_2$ accept only a Spark broadcast and a (cached) RDD, respectively, as input communication channels. Multiple conversions are needed to serve the two Spark operators.*

We model such complex scenarios of finding the most efficient communication path from a root pro-

ducer to multiple target consumers as the *minimum conversion tree* (MCT) problem.

MINIMUM CONVERSION TREE PROBLEM. *Given a* root channel c_r, n target channel sets C_{t_i} $(0 < i \leq n)$, *and the CCG* $G = (C, E, \lambda)$, *find a subgraph* G' *(i. e., a minimum conversion tree), such that:*

(1) *G' is a directed tree with root c_r and contains at least one channel c_{t_i} for each target channel set C_{t_i}, where $c_{t_i} \in C_{t_i}$.*

(2) *Any non-reusable channel in G', must have a single successor, i. e., a conversion or a consumer operator.*

(3) *The sum of costs of all edges in G' is minimized, i. e., there is no other subgraph G'' that satisfies the above two conditions and has a smaller cost than G'. The cost of an edge e is the estimated cost for the associated conversion operator $\lambda(e)$.*

Example 6 *In the example of Figure 5, the root channel is $c_r :=$* Stream *and the target channel sets are $C_{t_1} :=$ {*Broadcast*}* *(for target$_1$) and $C_{t_2} :=$ {*RDD, CachedRDD*}* *(for target$_2$). A minimum conversion tree for this scenario could look as follows: The* Stream *root channel is converted to a Java* Collection. *This* Collection *is then converted twice; namely to a Spark* Broadcast *(thereby satisfying C_{t_1}) and to an* RDD *(thereby satisfying C_{t_2}). Note that this is possible only because* Collection *is reusable.*

Although our MCT problem seems related to other well-studied graph problems, it differs from all of them. At first, we note that it differs from the minimum spanning tree problem, because MCTs have a fixed root and need not span the whole CCG. Our problem is also different from the single-source multiple-destinations shortest paths (SMSP) problem: We seek to minimize the costs of the conversion tree as a whole rather than its individual paths from the root to the target channels. That is, the overall costs for a conversion tree are not determined by the most expensive conversion path but by the *sum* of the costs of all conversions. Still, one might argue that our problem at hand is an SMSP problem if conversion operators could be executed in parallel. However, we do not allow for parallel execution of conversion operators for two reasons. First, SMSP is incompatible with Property (2) of the MCT definition for non-reusable channels: SMSP searches *individually optimal* paths, but in MCT we *must* consider the interaction between paths because paths must not diverge at a non-reusable channel. Second, when considering monetary costs for hardware utilization as operator costs, those costs also have to be summed up. The Group Steiner Tree (GST) problem is closest to our MCT problem: There, n sets of vertices should be connected by a minimal tree [47]. However, this problem is typically considered on undirected graphs and without the notion of non-reusable channels. Furthermore, GST solvers are often designed only for specific types of graphs, such as planar graphs or trees. These disparities preclude the adaption of existing GST solvers to the MCT problem. However, the GST problem allows to show the NP-hardness of our MCT problem.

Theorem 1 *The MCT problem is NP-hard.*

Proof: The NP-hard problem of GST [47] can be reduced in polynomial time to an MCT problem. Recall a GST instance consists of a weighted graph G with positive edge weights, a root vertex r, and k subsets (groups) of vertices from G. The goal of GST is to find a tree G' on G that connects r with at least one vertex of each group. We convert an instance of GST to MCT as follows. We provide as input to MCT (i) a channel conversion graph that has exactly the same vertices and edges with G, (ii) the vertex r as root channel, (iii) the k groups as target channel sets, and (iv) the edge weights of the graph as conversion costs. This conversion is clearly of polynomial complexity. □

5.3 Finding Minimum Conversion Trees

Because the MCT problem differs from existing graph problems, we devise a new algorithm to solve it (Algorithm 1). Given a CCG G, a root channel c_r, and n target channel sets $\mathscr{C}_t := \{C_{t_1}, C_{t_2}, ..., C_{t_n}\}$, the algorithm proceeds in two principal steps. First, it

Algorithm 1: Minimum conversion tree search.

Input: conversion graph G, root channel c_r, target channel sets $\mathscr{C}_t$
Output: minimum conversion tree

1 $\mathscr{C}_t \leftarrow \texttt{kernelize}(\mathscr{C}_t)$;
2 $T_{c_r} \leftarrow \texttt{traverse}(G, c_r, \mathscr{C}_t, \emptyset, \emptyset)$;
3 **return** $T_{c_r}[\mathscr{C}_t]$;

▷ `Recursive traversal`
Input: channel conversion graph G, current channel c, target channel sets $\mathscr{C}_t$, visited channels C_v, satisfied target channel sets $\mathscr{C}_s$
Output: minimum conversion trees from c to subsets of $\mathscr{C}_t$

4 **Function** *traverse*$(G, c, \mathscr{C}_t, C_v, \mathscr{C}_s)$
5 $T \leftarrow \texttt{create-dictionary}()$;
6 $\mathscr{C}'_s \leftarrow \{C_{t_i} \in \mathscr{C}_t \mid c \in C_{t_i}\} \setminus \mathscr{C}_s$;
7 **if** $\mathscr{C}'_s \neq \emptyset$ **then**
8 **foreach** $\mathscr{C}''_s \in 2^{\mathscr{C}'_s} \setminus \emptyset$ **do** $T[\mathscr{C}''_s] \leftarrow \texttt{tree}(c)$;
9 **if** $\mathscr{C}_s \cup \mathscr{C}'_s = \mathscr{C}_t$ **then return** T ;
10 $C_v \leftarrow C_v \cup \{c\}$;
11 **if** *reusable*(c) **then** $\mathscr{C}_s \leftarrow \mathscr{C}_s \cup \mathscr{C}'_s$;
12 $\mathcal{T} \leftarrow \emptyset$;
13 **foreach** $(c \xrightarrow{o} c') \in G$ *with* $c' \notin C_v$ **do**
14 $T' \leftarrow \texttt{traverse}(G, c', \mathscr{C}_t, C_v, \mathscr{C}_s)$;
15 $T' \leftarrow \texttt{grow}(T', c \xrightarrow{o} c')$;
16 $\mathcal{T} \leftarrow \mathcal{T} \cup \{T'\}$;
17 **if** *reusable*(c) **then** $d \leftarrow |\mathscr{C}_t| - |\mathscr{C}_s|$ **else** $d \leftarrow 1$;
18 **foreach** $\mathbf{T} \in$ *disjoint-combinations*$(\mathcal{T}, d)$ **do**
19 $T \leftarrow \texttt{merge-and-update}(\mathbf{T}, T)$
20 **return** T;

simplifies the problem by modifying the input parameters (*kernelization*, Line 1). Then, it explores the graph (*channel conversion graph exploration*, Line 2) to find the MCT (Line 3). We discuss these two steps in more detail in the following.

Kernelization. In the frequent case that two (or more) target consumers target_i and target_j accept the same communication channels, i. e., $C_{t_i} = C_{t_j}$, with at most one non-reusable channel and at least one reusable channel, we can merge them into a single set by discarding the non-reusable channel: $C_{t_{i,j}} = \{c \mid c \in C_{t_i} \wedge c \text{ is reusable}\}$. The key point of this kernelization is that it decreases the number of target channel sets and thus, reduces the maximum degree (i. e., fanout) of the MCT, which is a major complexity driver of the MCT problem. In fact, in the case of only a single target channel set (see Figure 2) the MCT problem becomes a single-source single-destination shortest path problem, which we can solve with, e. g., Dijkstra's algorithm.

Example 7 (Merging target channel sets) *In Figure 5,* target_2 *accepts the channels* $C_{t_2} = \{\mathsf{RDD}, \mathsf{CachedRDD}\}$. *Assume there is a further consumer* target_3 *(e. g., a* SparkMap *operator) that accepts the same set of channels as* target_2. *In this case, we can merge their input channels into* $C_{t_{2,3}} = \{\mathsf{CachedRDD}\}$.

Lemma 1 *A solution for a kernelized MCT problem also solves the original MCT problem.*

Proof: *Assume an original MCT problem* M_o *with identical target channel sets* $C_{t_1}, \ldots, C_{t_k}$ *and a kernelized MCT problem* M_k *for which those identical* C_{t_i} *have been merged to a single target channel set* C^{t*}. *Now let* t_k *be an MCT for* M_k. *Obviously,* t_k *is also a conversion tree for* M_o, *but it remains to show that it is also minimum. For that purpose, we assume that* t_k *was not minimum for* M_o; *in consequence, there has to be some other MCT* t_o *for* M_o. *If* t_o *satisfies all target channel sets of* M_o *(i. e., the* C_{t_i}*) via the same communication channel* c, *then* t_o *would also be an MCT for* M_k, *which contradicts our assumption. Specifically,* c *must be a reusable channel, as it satisfies multiple target channel sets. In contrast, if* t_o *satisfies the target channel sets of* M_o *with different channels, then there has to be at least one reusable channel* c' *among them, because we kernelize only such target channel sets that have* at most *one non-reusable channel. Because* c' *alone can already satisfy all target channel sets of* M_o, *it follows that* t_o *produces more target channels than necessary and is therefore not minimal – which also contradicts our assumption.* □

Channel conversion graph exploration. After the original MCT problem has been kernelized, Algorithm 1 proceeds to explore the CCG, thereby building the MCT from "its leaves to the root": Generally

speaking, our algorithm searches – starting from the root channel c_r – across the CCG for communication channels that satisfy the target channel sets $\mathscr{C}_t$; It then backtracks the search paths, thereby incrementally building up the MCT. Concretely, the `traverse` function implements this strategy via recursion – in other words, each call of this function represents a recursive traversal step through the CCG. The objective of each such step is to build up a dictionary T (Line 5) that associates subsets of the target channel sets, i. e., $\mathscr{C}_s \subseteq \mathscr{C}_t$, with *partial* conversion trees (PCTs) from the currently visited channel to those target channels $\mathscr{C}_s$. While backtracking from the recursion, these PCTs can then be merged successively until they form the final MCT. We explain these steps now in detail with the help of an example.

Example 8 *Assume we are solving the MCT problem in Figure 5, i. e.,* $c_r :=$ Stream, $C_{t_1} :=$ {Broadcast}*, and* $C_{t_2} :=$ {RDD, CachedRDD}*. Also, assume that we have already made one recursion step from the* Stream *to the* Collection *channel. That is, in our current invocation of* `traverse` *we visit* $c :=$ Collection*, on our current path we have visited only* $C_v =$ {Stream} *and did not reach any target channel sets, i. e.,* $\mathscr{C}_s := \emptyset$.

Visit channel (Lines 6–9). The `traverse` function starts by collecting all so far unsatisfied target channel sets $\mathscr{C}'_s$, that are satisfied by the currently visited channel c (Line 6). If there is any such target channel set (Line 7), we create a PCT for any combinations of those target channel sets in $\mathscr{C}'_s$ (Line 8). At this point, these PCTs consist only of c as root node, but will be "grown" during backtracking from the recursion. If we have even satisfied *all* target channel sets on our current traversal path, we can immediately start backtracking (Line 9). For the Example 8, $c =$ Stream does not satisfy any target channel set, i. e., we get $\mathscr{C}'_s = \emptyset$ and need to continue.

Forward traversal (Lines 10–16). In the second phase, the `traverse` function does the *forward* traversal. For that purpose, it marks the currently visited channel c as visited; and if c is reusable *and* satisfies some target channel sets $\mathscr{C}'_s$, it marks those sets also as satisfied (Lines 10–11). This is important to let the recursion eventually terminate. Next, the algorithm traverses forward by following all CCG edges starting at c and leading to an unvisited channel (Lines 13–14). For the Example 8, we accordingly visit Broadcast, RDD, and File. Each recursive call yields another dictionary T' of PCTs. For instance, when invoking `traverse` on Broadcast, we get $T'[C_{t_1}] =$ Broadcast (a PCT consisting only of Broadcast as root). At this point, we add the followed edge to this PCT to "grow" it (Line 16) and obtain the PCT Collection → Broadcast. We store all those "grown" PCTs in $\mathcal{T}$.

Merge PCTs (Lines 17–20). As a matter of fact, none of the PCTs in $\mathcal{T}$ might have reached all target channel sets. For instance, the above mentioned PCT Collection → Broadcast is the only one to satisfy C_{t_1}, but it does not satisfy C_{t_2}. Thus, the third and final phase of the `traverse` function merges certain PCTs in $\mathcal{T}$. Specifically, the `disjoint-combinations` function (Line 18) enumerates all combinations of PCTs in $\mathcal{T}$ that (i) originate from different recursive calls of `traverse`; (ii) do not overlap in their satisfied target channel sets; and (iii) consist of 1 to d different PCTs. While the former two criteria ensure that we enumerate all combinations of PCTs that may be merged, the third criterion helps us to avoid enumerating *futile* combinations: When the current channel c is not reusable, it must not have multiple consuming conversion operators, so d is set to 1 (Line 17). In any other case, any PCT must not have a degree larger than the number of not satisfied target channels sets; otherwise the enumerated PCTs would overlap in their satisfied target channel sets. Note that the value of d can be lowered by kernelization, which reduces the number of target channel sets. For the Example 8, we have three outgoing conversion edges from $c =$ Collection but only two non-satisfied target channel sets, namely C_{t_1} and C_{t_2}. As a result, we can avoid merging PCTs from all three edges *simultaneously*, as the resulting PCT could not be minimal. Eventually, the `merge-and-update` function combines the PCTs into a new PCT and, if there is no PCT in T already that reaches the same target channel sets and has lower costs, the new PCT is added to T (Line 19). Amongst others, we merge the PCTs Collection → Broadcast and Collection → RDD in our

example. When we backtrack (Line 20), the resulting PCT will be "grown" by the edge Stream → Collection and form the eventual MCT.

Complexity and Correctness. Our algorithm solves the MCT problem exactly (see Theorem 2). This comes at the cost of exponential complexity: There are $(n-1)!$ ways to traverse a full CCG of n channels and we might need to maintain 2^k partial trees in the intermediate steps, where k is the number of target channel sets. However, in practical situations, our algorithm finishes in the order of milliseconds, as the CCG comprises only tens of channels and is very sparse. Also, the number of target channel sets k is mostly only 1 or 2 and can often be diminished by the kernelization. More importantly, our algorithm avoids performance penalties from inferior data movement plans. However, if it ever runs into performance problems, one may consider making it approximate. Inspiration could be drawn from existing algorithms for GST [31, 25]. Yet, we evaluate our algorithm's scalability in Section 7.6 and show that it gracefully scales to a reasonable number of platforms.

Theorem 2 *Given a channel conversion graph, Algorithm 1 finds the minimum conversion tree if it exists.*

Proof: As per Lemma 1, the kernelization does not change the solution of an MCT problem, so we proceed to prove the correctness of the graph traversal algorithm – by induction. Let h be the height of the MCT. If $h = 1$, the conversion tree, which is composed of only a root (cf. Algorithm 1, Line 8), is always minimal as any conversion operator incurs non-negative costs. Assume an MCT of height h. We prove that our algorithm can output a tree of height $h+1$ that is also minimal. When merging PCTs two facts hold: (i) any subtree in the MCT must be an MCT (with its own root), otherwise this subtree has a cheaper alternative and the overall conversion tree cannot be minimal; and (ii) we consider all valid combination of PCTs in the merging phase and hence will not miss out the most efficient combination. Thus, given an MCT with height h, the tree with height $h+1$ will also be minimal. □

6 Plan Enumeration

The goal of our optimizer is to find the optimal plan, i. e., the plan with the smallest estimated cost. More precisely, for each inflated operator in an inflated plan, it needs to select one of its alternative execution operators, such that the overall execution cost is minimized. Finding the optimal plan, however, is challenging because of the exponential size of the search space. A plan with n operators, each having k execution operators, will lead to k^n possible execution plans. This number quickly becomes intractable for growing n. For instance, a cross-community PageRank plan, which consists of n=27 operators, each with k=5, yields $2,149,056,512$ possible execution plans. One could apply greedy pruning to reduce the search space significantly. For example, we could pick only the most cost-efficient execution operators for each inflated operator and prune all plans with other execution operators, but such a greedy approach could not guarantee to find the optimal execution plan, because it neglects data movement and platform start-up costs.

Thus, it is worthwhile to spend a bit more computation time in the optimization process in order to gain significant performance improvements in the task execution. We take a principled approach to solve this problem: We define an algebra to formalize the enumeration (Section 6.1) and propose a lossless pruning technique (Section 6.2). We then exploit this algebra and pruning technique to devise an efficient enumeration algorithm (Section 6.3).

6.1 Plan Enumeration Algebra

Inspired by the relational algebra, we define the plan enumeration search space along with traversal operations algebraically. This approach enables us to: (i) define the enumeration problem in a simple, elegant manner; (ii) concisely formalize our enumeration algorithm; and (iii) explore design alternatives. Let us first describe the data structures and operations of our algebra.

Data structures. Our enumeration algebra needs only one principal data structure, the *enumeration* $E = (S, SP)$, which comprises a set of *execution sub-*

plans SP for a given *scope* S. While the scope is the set of inflated operators that an enumeration is considering, the subplans select execution operators for each inflated operator in S, including execution operators for the data movement (see Section 5). Intuitively, one can think of an enumeration as a relational table whose schema corresponds to its scope and whose tuples correspond to its possible execution subplans.

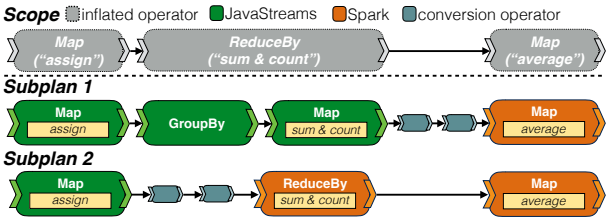


Figure 6: An example enumeration with two subplans.

Example 9 (Enumeration) *Figure 6 depicts an enumeration for the operators from Figure 4. It comprises two subplans for a scope of three inflated operators.*

Notice that if the scope contains all the inflated operators of a RHEEM plan, then the corresponding subplans form complete execution plans. This admits the following problem formalization.

PLAN ENUMERATION PROBLEM. *Given a* RHEEM *plan, let* $E = (S, SP)$ *be its complete enumeration. The goal is to efficiently identify a subplan* $sp_k \in SP$ *such that* $cost(sp_k) \leq cost(sp_i)\ \forall sp_i \in SP$*, where* $cost(sp_i)$ *comprises the costs of execution, data movement, and platform initializations of* sp_i*.*

Algrebra operations. Our enumeration algebra comprises two main operations, *Join* ($\bowtie$) and *Prune* (σ), both of which allow to manipulate enumerations. In few words, *Join* connects two small enumerations to form a larger one, while *Prune* scraps inferior subplans from an enumeration for efficiency reasons. Let us briefly establish these two operations before detailing how they can be used to enumerate complete execution plans.

Definition 3 (Join) *Given two disjoint enumerations* $E_1 = (S_1, SP_1)$ *and* $E_2 = (S_2, SP_2)$ *(i. e.,* $S_1 \cap S_2 = \emptyset$*), we define a join* $E_1 \bowtie E_2 = (S, SP)$ *where* $S := S_1 \cup S_2$ *and* $SP := \{\texttt{connect}(sp_1, sp_2) \mid sp_1 \in SP_1$ *can be connected to* $sp_2 \in SP_2\}$*. The* `connect` *function connects* sp_1 *and* sp_2 *by adding conversion operators between operators of the two subplans as explained in Section 5.*

Example 10 (Merging subplans) *The enumeration in Figure 6 could be created by joining an enumeration with scope* $S_1 = \{\mathsf{Map}(\text{“assign”}), \mathsf{ReduceBy}(\text{“sum\&count”})\}$ *with an enumeration with scope* $S_2 = \{\mathsf{Map}(\text{“average”})\}$*. In particular, the* `connect` *function adds conversion operators to link the two* Maps *in Subplan 1.*

Definition 4 (Prune) *Given an enumeration* $E = (S, SP)$*, we define a pruned enumeration* $\sigma_\pi(E) := (S, SP')$*, where* $SP' := \{sp \in SP \mid sp$ *satisfies* $\pi\}$ *and* π *is a configurable pruning criterion.*

Applying the algebra. Let us now draft a basic enumeration algorithm based on the *Join* and *Prune* operations. For each inflated operator o, we create a singleton enumeration $E = (\{o\}, SP_o)$, where SP_o are the executable subplans provided by o. We then join these singleton enumerations one after another to obtain an exhaustive enumeration for the complete RHEEM plan. By pruning the enumerations before joining them, we can drastically reduce the number of intermediate subplans, which comes with according performance benefits. That being said, this algorithm still lacks two important details, namely a concrete pruning rule and an order for the joins. We present our choices on these two aspects in the remainder of this section.

6.2 Lossless Pruning

We devise a novel strategy for the *Prune* operation that is *lossless*: it will not prune a subplan that is part of the optimal execution plan. As a result, the optimizer can find the optimal execution plan without an exhaustive enumeration of all execution plans. Overall, our pruning technique builds upon the notion of

boundary operators, which are inflated operators of an enumeration with scope S that are *adjacent* to some inflated operator *outside* of S.

Example 11 (Boundary operators) *In the scope of the enumeration from Figure 6,* Map *("assign") and* Map *("average") are boundary operators, because they have adjacent operators outside the scope; namely* RepeatLoop *and* Map *("parse") (cf. Figure 2).*

Having explained the boundary operators, we proceed to define our lossless pruning strategy that builds upon them.

Definition 5 (Lossless Pruning) *Let* $E = (S, SP)$ *be an enumeration and* $S_b \subseteq S$ *be the set of its* boundary *operators. The lossless pruning removes all* $sp \in SP$ *for which there is another* $sp' \in SP$ *that (i) contains the same execution operators for all* S_b *as* sp*, (ii) employs the same platforms as* sp*, and (iii) has lower cost than* sp*.*

Example 12 (Lossless Pruning) *For our example enumeration from Figure 6, the lossless pruning discards either Subplan 1 or Subplan 2 (whichever has the lower cost), because (i) the two subplans contain the same boundary execution operators (*JavaMap *("assign") and* SparkMap *("average")); and (ii) they need to initialize the same platforms (Java Streams and Spark).*

This pruning technique effectively renders the enumeration a dynamic programming algorithm by establishing the principle of optimality for certain subplans. Let us now demonstrate that this pruning rule is indeed lossless.

Lemma 2 *The lossless pruning does not prune a subplan that is contained in the optimal plan w.r.t. the cost model.*

Proof: See Appendix **??** for the proof. □

6.3 Enumeration algorithm

Algorithm 2 shows our enumeration algorithm with the lossless pruning strategy. Given an inflated RHEEM plan, we first create a singleton enumeration for each inflated operator (Line 1). We then need to repeatedly join and prune these enumerations to obtain the optimal execution plan. However, we aim at maximizing the pruning effectiveness by choosing a good order to join the enumerations. Thus, we first identify *join groups* (Line 2). A join group indicates a set of plan enumerations to be joined. Initially, we create a join group for each inflated operator's output, so that each join group contains (i) the enumeration for the operator with that output, E_{out}, and (ii) the enumerations for all inflated operators that consume that output as input, E^i_{in}. For instance in the inflated plan of Figure 2, the enumerations for Map ("assign") and ReduceBy ("sum & count") form an initial join group. While the join order is not relevant to the correctness of the enumeration algorithm, joining only adjacent enumerations is beneficial to performance: It minimizes the number of boundary operators in the resulting enumeration, which in turn makes our lossless pruning most effective (see Definition 5, Criterion (i)). To further promote this effect, we order the join groups ascending by the number of boundary operators (Line 3). Then, we greedily poll the join groups from the queue, execute the corresponding join, and prune the join product (Lines 4–6). Also, in any other join group that includes one of the joined enumerations, i. e., E_{out} or any E^i_{in}, we need to replace those joined enumerations with the join product $E_{\bowtie}$ (Lines 7–9). Note that these changes make it necessary to re-order the affected join products in the priority queue (Line 10). Eventually, the last join product is a full enumeration for the complete RHEEM plan. Its lowest cost subplan is the optimal execution plan (Line 11).

It is worth noting that our lossless pruning is related to the concept of *interesting sites* [39] in distributed relational query optimization, especially to the *interesting properties* [50] in general. We can easily extend our pruning rule to account for properties other than boundary operators. For example, we already do consider platform start-up costs in our cost model (see the plan enumeration problem statement in Section 6.1). As a result, we avoid pruning subplans with start-up costs that might be redeemed over the whole plan. Let us now establish the correctness

Algorithm 2: RHEEM plan enumeration

Input: RHEEM inflated plan R
Output: Optimal execution plan $sp_{\min}$

```
1  E ← {(({o}, SP_o) : o is an inflated operator ∈ R} ;
2  joinGroups ← find-join-groups(E) ;
3  queue ← create-priority-queue(joinGroups) ;
4  while |queue| > 0 do
5      joinGroup = {E_out, E_in^1, E_in^2, ...} ← poll(queue)
       ;
6      E_⋈ ← σ(E_out ⋈ E_in^1 ⋈ E_in^2 ⋈ ...) ;
7      foreach joinGroup' ∈ queue do
8          if joinGroup ∩ joinGroup' ≠ ∅ then
9              update(joinGroup' with E_⋈) ;
10             re-order(joinGroup in queue);
11 sp_min ← the subplan in E_⋈ with the lowest cost ;
```

of our enumeration algorithm.

Theorem 3 *The enumeration Algorithm 2 determines the optimal execution plan w. r. t. the cost estimates.*

Proof: As Algorithm 2 applies a lossless pruning technique (as per Lemma 2) to an otherwise *exhaustive* plan enumeration, it detects the optimal execution plan. □

7 Experiments

We implemented our proposed cross-platform optimizer in the open-source cross-platform system RHEEM[4]. For the sake of simplicity, we henceforth refer to our optimizer simply as RHEEM. We have carried out several experiments to evaluate the effectiveness and efficiency of RHEEM. Note that our work is the first to provide cross-platform optimization for general-purpose tasks (see Section 8 for related work). We thus compared it vis-a-vis individual platforms and common practices.

In particular, we focus on evaluating RHEEM in all four cross-platform use cases (see Appendix **??**) and aim at answering the following questions: ***(Platform Independence)*** *Can* RHEEM *choose the best platform for a given task?* (Section 7.2); ***(Opportunistic Cross-Platform)*** *Does* RHEEM *spot hidden opportunities for cross-platform processing that improve performance?* (Section 7.3); ***(Polystores)*** *How well does the optimizer perform in a data lake setting?* (Section 7.4); and ***(Mandatory Cross-Platform)*** *Can the optimizer complement the functionalities of one platform with another to perform a given task?* (Section 7.5). Last but not least, we also evaluate the design of our optimizer itself: ***(Internals)*** *How efficient and scalable is our optimizer?* (Section 7.6).

[4]Link omitted because of the double-blind policy.

7.1 Setup

Hardware. We ran all our experiments on a cluster of 10 machines. Each node has one 2 GHz Quad Core Xeon processor, 32 GB main memory, 500 GB SATA hard disks, a 1 Gigabit network card and runs 64-bit platform Linux Ubuntu 14.04.05.

Processing & Storage Platforms. We considered the following platforms: Java's Stream library (JavaStreams), PostgreSQL 9.6.2 (PSQL), Spark 1.6.0 (Spark), Flink 1.3.2 (Flink), GraphX 1.6.0 (GraphX), Giraph 1.2.0 (Giraph), a simple self-written Java graph library (JGraph), and HDFS 2.6.0 to store files. We used all these systems with their default settings and configured the RAM of each platform to 20 GB. We obtained all the cost functions required by our optimizer as described in Appendix B.

Tasks and datasets. We have considered a broad range of data analytics tasks from different areas, namely text mining (TM), relational analytics (RA), machine learning (ML), and graph mining (GM). Details on the datasets and tasks are shown in Table 1. These tasks and datasets individually highlight different features of RHEEM and together demonstrate its general applicability. Note that, to allow RHEEM to choose most of the available platforms, all tasks' input datasets (with the exception of `Polystore`'s) are stored on HDFS (except when specified otherwise). To challenge RHEEM and evaluate its effectiveness, we focused primarily on medium-sized datasets, so

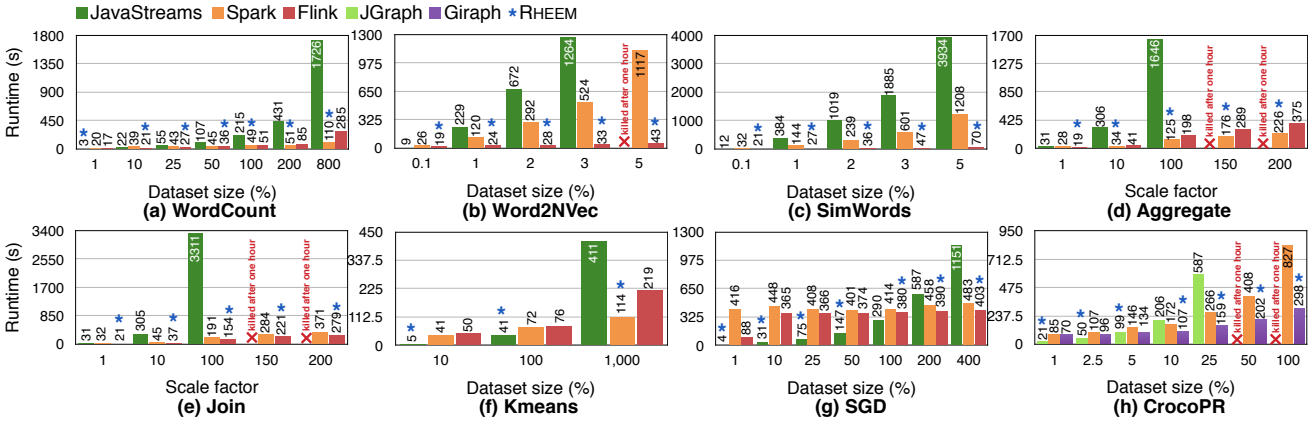


Figure 7: Platform independence: Rheem avoids all worst execution cases and chooses the best platform for almost all tasks.

that all platforms are competitive performance-wise. Nonetheless, RHEEM scales to large datasets when provided with scalable processing platforms. Note that all the numbers we report are the average of three runs.

Table 1: Tasks and datasets.

Task	Description	Dataset	Default store
`WordCount` (TM)	count distinct words	*Wikipedia abstracts* (3GB)	HDFS
`Word2NVec` (TM)	word neighborhood vectors	*Wikipedia abstracts* (3GB)	HDFS
`SimWords` (TM)	word neighborhood clustering	*Wikipedia abstracts* (3GB)	HDFS
`Aggregate` (RA)	aggregate query (TPC-H Q1)	*TPC-H* (1-100GB)	HDFS
`Join` (RA)	2-way join (TPC-H Q3)	*TPC-H* (1-100GB)	HDFS
`PolyJoin` (RA)	n-way join (TPC-H Q5)	*TPC-H* (1-100GB)	Postgres, HDFS, LFS
`Kmeans` (ML)	clustering	*USCensus1990* (361MB)	HDFS
`SGD` (ML)	stochastic gradient descent	*HIGGS* (7.4GB)	HDFS
`CrocoPR` (GM)	cross-community pagerank	*DBpedia pagelinks* (20GB)	HDFS

Repeatability. To ensure repeatability, we provide the code of all our experimental applications (RHEEM plans), SQL queries, datasets, and a detailed guideline on how to reproduce our experiments at [*link omitted for double-blind review*].

7.2 Platform Independence

We start our experiments by evaluating how well RHEEM selects a single data processing platform to execute a given task.

Experiment setup. For this experiment, we forced RHEEM to use a single platform when executing a task. Then, we checked if our optimizer chose the one with the best runtime. We ran all the tasks of Table 1 with increasing dataset sizes. Note that we do not run `PolyJoin` as it cannot be performed using a single platform. For `CrocoPR`, `K-means`, and `SGD`, we loop the algorithms for 10, 100, and 1,000 times, respectively. To further stress the optimizer, for `K-means`, we increased its input dataset ten times (i. e., 3.6GB).

Results. Figure 7 shows the execution times for all our data analytic tasks and for increasing dataset sizes[5]. The stars denote the platform selected by our optimizer. First of all, let us stress that the results show significant differences in the runtimes of the different platforms: even between Spark and Flink, which are big data platform competitors. For example, Flink can be up to 26x faster than Spark and Spark can be twice faster than Flink for the tasks we considered in our evaluation. Therefore, it is crucial for an optimizer to prevent tasks from falling into such non-

[5]For the non-synthetic datasets, we created samples of increasing size.

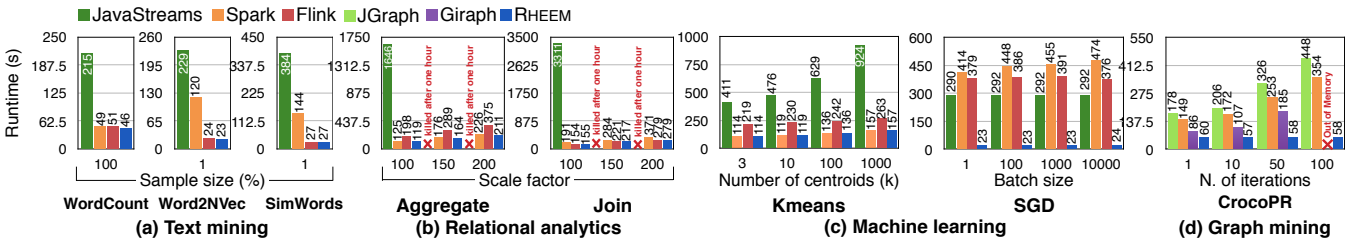


Figure 8: Opportunistic cross-platform: Rheem improves performance by combining multiple data processing platforms.

obvious worst cases. The results, in Figure 7, show that our optimizer indeed makes robust platform choices whenever runtimes differ substantially. This effectiveness of the optimizer for choosing the right platform transparently prevents applications from using suboptimal platforms. For instance, it prevents running: (i) `Word2NVec` and `SimWords` on Spark for 5% of its input dataset. Spark performs worse than Flink for `Word2NVec` and `SimWords` because it employs only 2 compute nodes (one for each input data partition), while Flink uses all 10; (ii) `SimWords` on Java for 1% of its input dataset ($\sim$30MB); as `SimWords` performs many CPU-intensive vector operations, using JavaStreams (i. e., a single compute node) simply slows down the entire process; (iii) `WordCount` on Flink for 800% of its input dataset (i. e., 24GB), where, in contrast to Spark, Flink suffers from a slower data reduce mechanism[6] ; and (iv) `CrocoPR` on JGraph for more than 10% of its input dataset as it simply cannot efficiently process large datasets. We also observe that RHEEM generally chooses the right platform even for the difficult cases where the execution times are quite similar on different platforms. For example, it always selects the right platform for `Aggregate` and `Join` even if the execution times for Spark and Flink are quite close to each other. Only in few of these difficult cases the optimizer fails to choose the best platform, such as in `Word2NVec` and `SimWords` for 0.1% of input data. This is because the accuracy of our optimizer is very sensitive to uncertainty factors, such as cost model calibration and cardinality estimates. These factors are also quite challenging to estimate even for controlled settings, such as in databases. Still, despite these two cases, all these results allow us to conclude that *our optimizer chooses the best platform for almost all tasks and it prevents tasks from falling into worst execution cases.*

[6]Flink uses a sorting-based aggregation, which – in this case – appears to be inferior to Spark's hash-based aggregation.

7.3 Opportunistic Cross-Platform

We continue our evaluation by studying the efficiency of RHEEM when using multiple platforms for a task. We especially examine if it can spot hidden opportunities for the use of multiple platforms.

Experiment setup. We now allow RHEEM to use any platform combination. We use the same tasks and datasets with three differences: we ran (i) Kmeans on 10x its entire dataset for a varying number of centroids, (ii) SGD on its entire dataset for different batch sizes, and (iii) CrocoPR with 10% of its input dataset and for a varying number of iterations.

Results. Figure 8 shows the results for these experiments. Overall, we find that in the worst case RHEEM matches the performance of any single platform execution, but in several cases considerably improves over single-platform executions. We observe it to be up to 20× faster than Spark, up to 17× faster than Flink, up to 21× faster than JavaStreams, up to 6× faster than Giraph. There are several reasons for having this large improvement. In detail, for `SGD`, RHEEM decided to handle the model parameters (typically a very small dataset) with JavaStreams while it processed the data points (typically a large dataset) with Spark. For `CrocoPR`, surprisingly our optimizer uses a combination of Flink and JGraph, even if Giraph is the fastest baseline platform. This is because after

the preparation phase of this task, the input dataset for the PageRank operation on JGraph is a couple of megabytes only. Additionally, for `WordCount`, RHEEM detected that moving the result data from Spark to JavaStreams and afterwards shipping it to the driver application is slightly faster than Spark (which is the fastest baseline platform for this task). This is because when moving data to JavaStreams RHEEM uses the action Rdd.collect(), which is more efficient than the operation Spark uses to move data to the driver (Rdd.toLocalIterator()). For `Aggregate`, our optimizer selects Flink and Spark, which allows it to run this task slightly faster than the fastest baseline platform, which is Spark for this task. Our optimizer achieves this improvement by (i) exploiting the fast stream data processing mechanism native in Flink for the projection and selection operations, and (ii) avoiding the slow data reduce mechanism of Flink by using Spark for the ReduceBy operation. These are surprising results per-se. They show not only that RHEEM can outperform state-of-the-art platforms by using combinations of them, but also that it can spot hidden opportunities for cross-platform execution.

To further stress the importance of finding hidden cross-platform execution opportunities, we ran a subquery (`JoinX`) of `PolyJoin`: This query joins the relations `SUPPLIER` and `CUSTOMER` (which are stored on Postgres) on the attribute `nationkey` and aggregates the join results on the same attribute. For this additional experiment, we compare RHEEM with the execution of `JoinX` on Postgres, which is the obvious platform to run this kind of queries.

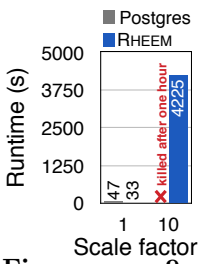


Figure 9: JoinX

The results are displayed in Figure 9. Remarkably, we observe that RHEEM significantly outperforms Postgres, even though the input data is stored there. In fact, RHEEM is more than twice as fast as Postgres for a scale factor of 10. This is because it simply pushes down the projection operation into Postgres and then moves the data into Spark to perform the join and aggregation operations, thereby leveraging the parallelism offered by Spark. We thus do confirm that *our optimizer is indeed able to identify hidden opportunities to improve performance as well as to perform much more robustly (than when selecting a single platform) by using multiple platforms.*

7.4 Polystores

We now evaluate the efficiency of RHEEM in polystore settings, where the input datasets are dispersed across several systems.

Experiment setup. We consider the `PolyJoin` task, which takes the `CUSTOMER`, `LINEITEM`, `NATION`, `ORDERS`, `REGION`, and `SUPPLIER` TPC-H tables as input. We stored the `LINEITEM` and `ORDERS` tables in HDFS, the `CUSTOMER`, `REGION`, and `SUPPLIER` tables in Postgres, and the `NATION` table in a local file system (LFS). In this scenario, the common practice is to move the data into a relational database in order to enact the analytical queries inside the database [27, 51]. Therefore, we consider such a typical case as the baseline. That is, we move all HDFS and LFS data into Postgres and then perform `PolyJoin` over Postgres. We measure the data migration time as well as the query execution time as the total runtime for this baseline. In contrast, RHEEM can process the input datasets directly on the data stores where they reside.

Results. Figure 10(a) shows the results: RHEEM is significantly faster, namely up to 5×, than the current practice. In particular, we observed that loading data into Postgres is already approximately 3× slower than it takes RHEEM to complete the entire task. Even when discarding data migration times, RHEEM can still outperform Postgres. For example, the pure execution time in Postgres for a scale factor of 100 amounts to 2,159 seconds compared to 1,608 seconds for RHEEM. This shows the *substantial benefits of our optimizer in polystore scenarios, not only in terms of performance but also in terms of ease-of-use, as users do not have to write ad-hoc scripts anymore to integrate different data processing platforms.*

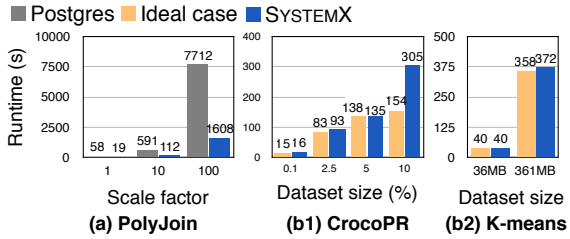


Figure 10: (a) Polystore & (b) Mandatory multi-platform.

7.5 Mandatory Cross-Platform

We proceed to evaluate the effectiveness of Rheem to complement the functionalities of disparate processing platforms.

Experiment setup. To evaluate this feature, we consider the `CrocoPR` and `Kmeans` tasks. In contrast to previous experiments, we assume both input datasets (*DBpedia* and *USCensus1990*) to be on Postgres. As the implementation of these tasks on `Postgres` would be very impractical and of utterly inferior performance, it is important to move the computation to a different processing platform. In these experiments, we consider as baseline the *ideal case* where the data resides in the HDFS instead and Rheem uses either JavaStreams or Spark to run the tasks.

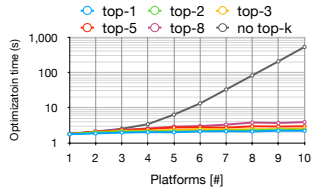
Results. Figure 10(b) shows the results. We observe that Rheem achieves similar performance with the ideal case in almost all scenarios. This is a remarkable result, as it needs to move data out of Postgres to a different processing platform, in contrast to the ideal case. Only for `CrocoPR` and only for the largest dataset we measured a slowdown of Rheem w. r. t. the ideal case, which is because Rheem reads data in parallel in the ideal case (i. e., when reading from HDFS), which is not possible when reading from Postgres. Nevertheless, it is more efficient (and practical) than writing ad-hoc scripts to move data out of Postgres and running the task on a different platform. In particular, we observed that the optimizer pushes down projections and selections to Postgres, and thus reduces the amount of data to be moved. These results show that *our optimizer frees users from the burden of complementing the functionalities of diverse platforms, without sacrificing performance.*

7.6 Optimizer Scalability

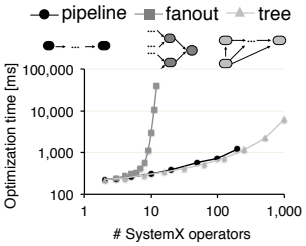
We finish our experimental study by evaluating the scalability of our optimizer in order to determine whether it operates efficiently on large Rheem plans and for large numbers of platforms.

Experiment setup. We start by evaluating our optimizer's scalability in terms of number of supported platforms and then proceed to evaluate it in terms of number of operators in a Rheem plan. For the former, we consider hypothetical platforms with full Rheem operator coverage and three communication channels each. For the latter, we generated Rheem plans with three basic topologies that we found to be at the core of many data analytic tasks: *pipeline*, *fanout*, and *tree*. Notice that most iterative analytics also fall into these three topologies.

Results. Figure 11(a) details the scalability of our optimizer for `Kmeans` when increasing the number of supported platforms – the results for the other tasks are similar. As expected, the optimization time increases along with the number of platforms. This is because (i) the CCG gets larger, challenging our MCT algorithm, and (ii) our lossless pruning has to retain more alternative subplans. Still, we observe that our optimizer (the *no top-k* series in Figure 11(a)) performs well for a practical number of platforms: it takes less than 10 seconds when having 5 different platforms. Yet, when adding a simple top-k pruning strategy, our optimizer gracefully scales with the number platforms, e. g., for k=8 it takes less than 10 seconds when having 10 different platforms. Note that our algebraic formulation of the plan enumeration problem allows to easily augment our optimizer with a top-k pruning strategy (see Section 6.1): We just specify an additional rule for the *Prune* operator. Additionally, the resulting pruning is superior to plain top-k pruning, as the preceeding lossless pruning still removes redundant, inferior subplans. Let us now proceed to evaluate our optimizer's scalability w. r. t. the number of operators in a task. Figure 11(b) depicts the above mentioned plan



(a) Scalability w.r.t platforms



(b) Enumeration scalability

Figure 11: Optimization scalability.

topologies along with our experimental results. It is clearly visible that the optimizer scales to very large plans for the pipeline and tree topologies. In contrast, we get a different picture for the fanout topology: The optimizer processed plans with at most 12 operators[7] within a time frame of 5 minutes. Such plans entail hard MCT problems and allow for only very little (lossless) pruning. However, we encountered only much smaller fanouts in real-world tasks. We can thus conclude that *our optimizer can scale to a realistic number of platforms and to a reasonable number of operators in a* Rheem *plan.* The interested reader can find further insights into the optimizer's internals in Appendix D.

8 Related Work

In the past years, the research and industry communities have proposed many data processing platforms [26, 5, 10, 55, 16]. In contrast to all these works, we do not provide a new processing platform but an optimizer to automatically combine and choose among such different platforms.

Cross-platform task processing has been in the spotlight very recently. Some works have proposed different solutions to decouple data processing pipelines from the underlying platforms [32, 30, 28, 53, 42, 14]. Although their goals are similar, all these works differ substantially from our optimizer as none of them considers data movement costs, which is crucial in cross-platform settings. Note that some complementary works [46, 33] focus on improving data movement among different platforms, but they do not provide a cross-platform optimizer. Moreover, each of these systems *additionally* differs from our optimizer in various ways. Musketeer's main goal is to decouple query languages from execution platforms [32]. Its main focus lies on converting queries via a fixed intermediate representation and thus mostly targets platform independence (cf. Appendix **??**). BigDAWG [30] comes with no optimizer and requires users to specify where to run cross-platform queries via its `Scope` and `Cast` commands. In [28] the authors present a cross-platform system intended for optimizing complex workflows. It allows only for simple one-to-one operator mappings and does not consider optimization at the atomic operator granularity. The authors in [53] focus on ETL workloads making it hard to extend their proposed solution with new operators and other analytic tasks. DBMS+ [42] is limited by the expressiveness of its declarative language and hence it is neither adaptive nor extensible. Furthermore, it is unclear how DBMS+ abstracts underlying platforms seamlessly. Tensorflow [14] follows a similar idea but for cross-device execution of machine learning tasks and thus it is orthogonal to Rheem.

Query optimization has been the focus of a great amount of literature [36]. However, most of these works focus on relational-style query optimization, such as operator re-ordering and selectivity estimation, and cannot be directly applied to our system. More closely to our work is the optimization for federated DBMSs where adaptive query processing and re-optimization is of great importance [18, 19, 43]. Nevertheless, the solutions of such works are tailored for relational algebra and assume tight control over the execution engine, which is not applicable to our case. Finally, there is a body of work on UDF-based data flow optimization, such as [48, 35]. Such optimizations are complementary to our optimizer and

[7] The tasks from Table 1 have 17 operators on average.

one could leverage them to better incorporate UDFs in our cost models.

MapReduce-based integration systems mainly aims at integrating Hadoop with RDBMS, such as [40, 27]. However, these works cannot be easily extended to deal with more diverse data analytic tasks and different processing platforms. Furthermore, there are works that automatically decide whether to run a MapReduce job locally or in a cluster, such as FlumeJava [23]. Although such an automatic choice is crucial for some tasks, it does not generalize to data flows with other platforms.

Finally, **federated databases** have been studied since almost the beginnings of the database field itself [52]. Garlic [22], TSIMMIS [24], and InterBase [21] are just three examples. However, all these works significantly differ from ours in that they consider a single data model and push query processing to where the data is.

9 Conclusion

We presented a cross-platform optimizer that automatically allocates a data analytic task to a combination of data processing platforms in order to minimize its execution costs. Our optimizer takes into consideration the nature of the incoming task, platform characteristics, and data movement costs in order to select the most efficient platforms for a given task. In particular, we proposed (i) novel strategies to map platform-agnostic tasks to concrete execution strategies; (ii) a new graph-based approach to plan data movement among platforms; and (iii) an algebraic formalization and novel solution to select the optimal execution strategy. Our extensive evaluation with real-word workloads and datasets showed that our optimizer allows data analytic tasks to run up to more than one order of magnitude faster than on any single platform.

Moreover, our insights spawn many interesting questions, such as how to reduce inter-platform data movement costs and how to tackle the cardinality and cost estimation problem. Particularly, the latter is a relevant topic for many novel, optimizer-backed data processing platforms. In the Appendices B and C we outline some basic approaches to be further pursued.

References

[1] Apache Drill. `https://drill.apache.org`.

[2] Apache Flume. `https://flume.apache.org/index.html`.

[3] Apache Hive: A data warehouse software for distributed storage. `http://hive.apache.org`.

[4] Apache Mahout. `http://mahout.apache.org`.

[5] Apache Spark: Lightning-fast cluster computing. `http://spark.apache.org`.

[6] Data-driven healthcare organizations use big data analytics for big gains. IBM Software white paper.

[7] Demonstration of Rheem. Because of double-blind review, this reference is available upon request.

[8] Java 8 Streams library. `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`.

[9] Luigi project. `https://github.com/spotify/luigi`.

[10] PostgreSQL. `http://www.postgresql.org`.

[11] PrestoDB Project. `https://prestodb.io`.

[12] Spark MLlib: `http://spark.apache.org/mllib`.

[13] Spark SQL programming guide. `http://spark.apache.org/docs/latest/sql-programming-guide.html`.

[14] M. Abadi et al. TensorFlow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[15] 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 *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 479–484, 2016.

[16] A. Alexandrov et al. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[17] A. Baaziz and L. Quoniam. How to use big data technologies to optimize operations in upstream petroleum industry. *International Journal of Innovation (IJI)*, 1(1):19–25, 2013.

[18] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2005.

[19] S. Babu, P. Bizarro, and D. J. DeWitt. Proactive re-optimization with Rio. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 936–938, 2005.

[20] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment (PVLDB)*, 9(13):1425–1436, 2016.

[21] O. A. Bukhres, J. Chen, W. Du, A. K. Elmagarmid, and R. Pezzoli. Interbase: An execution environment for heterogeneous software systems. *IEEE Computer*, 26(8):57–69, 1993.

[22] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the International Workshop on Research Issues in Data Engineering - Distributed Object Management (RIDE-DOM)*, pages 124–131, 1995.

[23] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[24] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Information Processing Society of Japan (IPSJ)*, pages 7–18, 1994.

[25] C. Chekuri, G. Even, and G. Kortsarz. A greedy approximation algorithm for the Group Steiner problem. *Discrete Applied Mathematics*, 154(1):15–34, 2000.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[27] D. J. DeWitt, A. Halverson, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in Polybase. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1255–1266, 2013.

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

[29] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. The BigDAWG polystore system. *SIGMOD Record*, 44(2):11–16, 2015.

[30] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al. A demonstration of the BigDAWG polystore system. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1908–1911, 2015.

[31] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the Group Steiner Tree problem. *Journal of Algorithms*, 37(1):66–84, 2000.

[32] 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 *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 1–16. ACM, 2015.

[33] B. Haynes, A. Cheung, and M. Balazinska. PipeGen: Data pipe generator for hybrid analytics. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 470–483, 2016.

[34] A. Hems, A. Soofi, and E. Perez. How innovative oil and gas companies are using big data to outmaneuver the competition. Microsoft white paper, 2014.

[35] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1256–1267, 2012.

[36] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[37] P. Jovanovic, A. Simitsis, and K. Wilkinson. Engine independence for logical analytic flows. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1060–1071, 2014.

[38] Z. Kaoudi, J.-A. Quiane-Ruiz, S. Thurumuruganathan, S. Chawla, and D. Agrawal. A Cost-based Optimizer for Gradient Descent Optimization. In *SIGMOD*, 2017.

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

[40] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping up big data query processing with a multistore system. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1591–1602, 2014.

[41] V. Leis et al. How good are query optimizers, really? *Proceedings of the VLDB Endowment (PVLDB)*, 9(3):204–215, 2015.

[42] H. Lim, Y. Han, and S. Babu. How to fit when no one size fits. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2013.

[43] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 659–670, 2004.

[44] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[45] K. Noyes. For the airline industry, big data is cleared for take-off. `http://fortune.com/2014/06/19/big-data-airline-industry`.

[46] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkopt, S. P. Amarasinghe, and M. Zaharia. A common runtime for high performance data analysis. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2017.

[47] G. Reich and P. Widmayer. Beyond Steiner's problem: A VLSI oriented generalization. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 196–210, 1989.

[48] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An extensible logical optimizer for UDF-heavy data flows. *Information Systems*, 52:96–125, 2015.

[49] P. J. Sadalage and M. Fowler. *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Addison-Wesley Professional, 2012.

[50] 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 *Proceedings of the International Conference on Management of Data (SIGMOD)*, SIGMOD '79, pages 23–34, 1979.

[51] S. Shankar, A. Choi, and J.-P. Dijcks. Integrating Hadoop data with Oracle parallel Processing. Oracle white paper, 2010.

[52] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[53] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 829–840, 2012.

[54] M. Stonebraker. The case for polystores. ACM SIGMOD Blog.

[55] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *Proceedings of the VLDB Endowment (PVLDB)*, 9(5):420–431, 2016.

A Coding SystemX Apps

Let us outline how users can build an application on top of Rheem. Rheem can be programmed with two principal APIs, namely for Java and Scala. Additionally, Rheem provides a declarative language based on PigLatin that enables users to specify their Rheem plans declaratively. In the following, we focus on the Scala API.

In particular, we demonstrate how to code our k-means example from Figure 2 in Listing 1. For clarity, we omit some UDF implementations, but the full application is available online.[8] When developing a Rheem application, the user has to initially create a Rheem context and specify the available processing platform plugins (Lines 2–4). A platform plugin defines a set of execution operators (including data movement) for a given processing platform along with driver code that integrates the platform with Rheem' execution and monitoring facilities.

```
1  // Initialization.
2  val context = new SystemXContext(new Configuration)
3    .withPlugin(Spark.basicPlugin)
4    .withPlugin(JavaStreams.basicPlugin)
5  // K-means plan.
6  val builder = new PlanBuilder(context)
7  val points =
8       builder.readTextFile("hdfs://my/data/points.csv")
9    .map(parsePoints)
10 val finalCentroids =
11      builder.loadCollection(createRandomCentroids())
12   .repeat(50, { centroids =>
13     points.map(selectNearestCentroid).withBroadcast(centroids)
14       .reduceByKey(_.centroidId, _ + _)
15       .map(_.average)
16   }).collect()
```

Listing 1: Rheem code for k-means.

Once the context is defined, the user can initiate a new Rheem plan (Line 6) and define its operators and its topology. Depending on their type, an operator can have zero or more data flow inputs and outputs, with the particularity that outputs can be connected to multiple operators as inputs. Rheem' API adapts the proven design from, e.g., the Scala standard library and Apache Spark: Each method call instantiates a new operator with the callee as input operator. For instance, the `map` call in Line 13 creates a new Map operator that takes as input the ReduceBy operator from Line 12. Eventually, one or more sink operators collect the output data. Our example defines a CollectionSink in Line 14. Only at this point, the plan is completely specified and Rheem can start optimizing and executing it.

Observe that users write such platform-agnostic code without any knowledge of the underlying platforms. It is Rheem that will figure out on which platform to execute each of the operators to achieve the best performance. If one wants to force the system to execute a given operator on a specific platform, she can do so by invoking `withTargetPlatform(...)` on the operator. In this case the optimizer will then pick among different execution operators of the specific platform, if there exist more than one. She can

[8] URL omitted because of double-blind policy.

also force the system to use a specific execution operator via the `customOperator(...)` method. This also allows to supply RHEEM with custom operators without having to extend the API.

Finally, it is noteworthy that the API accepts optimization hints, e. g., the selectivity of a filter UDF. RHEEM does not require or assume the presence of hints, but if they are available, they can improve performance.

B Learning Cost Functions

In a naïve approach, one might individually profile each operator to build a cost model. However, it is unclear what data characteristics the profiling tool should establish as well as measuring operators in isolation might be unrealistic. It is true that most platforms optimize execution across multiple operators, e. g., by pipelining or by reusing certain data partition schemes.

We thus take a different approach. We measure the runtime execution times of actual execution plans and use the gathered information to *learn* the cost functions of each execution operator. Still, getting precise cost functions at the execution operator level is quite challenging because most data processing platforms employ lazy-execution, i. e., they execute multiple operators together. We can measure the execution times of such *execution blocks* only and not by operator. We thus model the cost of individual execution operators as a *regression problem*. Let $(\{(o_1, C_1), (o_2, C_2), \dots (o_n, C_n)\}, t)$ be an execution block, with o_i, $0 < i \leq n$, where the o_i are execution operators, the C_i are input and output cardinalities, and t is the measured execution time for the entire block. Furthermore, let $f_i(\mathbf{x}, C_i)$ be the cost function for the execution operator o_i with parameters $\mathbf{x}$. We are interested in finding $\mathbf{x}_{\min}$:

$$\mathbf{x}_{\min} = \arg\min_{\mathbf{x}} loss\left(t,\ \sum_{i=1}^{n} f_i(\mathbf{x}, C_i)\right)$$

Specifically, we use a *relative* loss function defined as $loss(t, t') = \left(\frac{|t-t'|+s}{t+s}\right)^2$, where t' is the geometric mean of the lower and upper bound of the time estimate produced by $\sum f_i(\mathbf{x}, C_i)$ and s is a regularizer inspired by additive smoothing that tempers the loss for small t. Note that we can easily generalize this optimization problem to multiple execution blocks: Then we minimize the weighted arithmetic mean of the losses of multiple execution blocks. In particular, we use as block weights the sum of the relative frequencies of the blocks' operators among all blocks, so as to deal with skewed workloads that contain certain operators more often than others. Finally, we minimize our loss function with a genetic algorithm [44]. In contrast to other optimization algorithms, genetic algorithms impose only few restrictions on the function to be minimized. That way, we can deal with arbitrary cost functions.

C Progressive Optimization

Appendix B shows how to use historical execution data to learn or refine cost functions for execution operators. The same principle is applicable to cardinality estimation. Improving the cardinality estimates is valuable as it allows our optimizer to make more precise and well-informed decisions. A complementary approach to improve cardinality estimates is the *progressive query optimization* [43], which can easily be incorporated into our optimizer.

The key idea is to monitor actual cardinalities of an execution plan and re-optimize it on-the-fly whenever the observed cardinalities greatly mismatch the estimated ones. In particular, we extend the optimizer to attach a confidence value to the output cardinality and cost estimates. Then, we insert *optimization checkpoints* into RHEEM plans, where (i) cardinality estimates are uncertain and (ii) the data is "at rest" (e. g., a Java collection or a file). Once the executor encounters a checkpoint, it pauses the execution and uses the measured cardinalities to re-optimize the execution plan from that point and beyond. In fact, the regular optimization pipeline as detailed throughout Sections 4–6 can be re-used with only minor modifications. Then, RHEEM resumes the execution with the re-optimized plan.

We evaluated the utility of progressive optimization in our optimizer by setting up a scenario with

poor cardinality estimates, which is plausible in real-world scenarios. For this experiment, we underestimated the selectivity of the operator in the `WordCount` task that splits each line of an input file into separate words. We observe that our progressive optimization technique allows RHEEM to run more than one order of magnitude faster than without: we measured a runtime of 15 seconds without re-optimization and only 1 second if the progressive optimization is enabled. This is because RHEEM can detect the incorrect cardinality estimates and revise its decision of using Spark, thereby saving Spark's start-up costs (∼12 sec. in our cluster).

D Optimizer Internals

Besides the scalability results presented in Section 7.6, we also conducted several experiments to further evaluate the efficiency of our optimizer. We start by analyzing the importance of the order, in which our enumeration algorithm process join groups (see Section 6.3). As we observe in Figure 12(a) for the tree topology, ordering the join groups order *can* indeed be crucial. On the other hand, the process of ordering the join groups does not seem to exert any measurable influence on the optimization time.

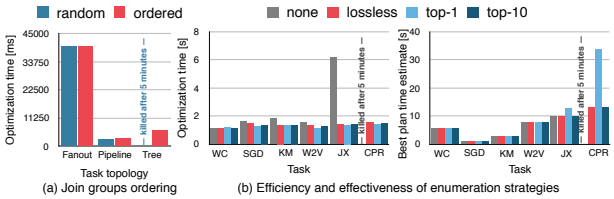


Figure 12: Optimizer internals.

Additionally, we compare our lossless pruning strategy (Section 6) with several alternatives, namely no pruning at all and top-k pruning[9] that retains the k best subplans when applied to an enumeration. Figure 12(b) shows the efficiency results of all pruning strategies (on the left) as well as their effectiveness (on the right), i. e., the estimated execution times of their optimized plans. Note that we did not use the actual plan execution times to assess the effectiveness of our enumeration strategy in order to eliminate the influence of the calibration of the cost functions. As a first observation, we see that pruning is crucial overall: An exhaustive enumeration was not possible for `CrocoPR` (`CPR`). On the other hand, we found that the top-1 strategy, which merely selects the best alternative for each inflated operator, is pruning too aggressively and fails (3 out of 7 times) to detect the optimal execution plan. While the numbers now seem to suggest that the remaining lossless and top-10 pruning strategies are of the same value, there is a subtle difference, though: The lossless strategy *guarantees* to find the optimal plan (w.r.t. the cost estimates) and is, thus, superior. For large, complex RHEEM plans, as discussed in the above paragraph, a combination of the lossless pruning followed by a top-k pruning might be a valuable pruning strategy. While the former keeps intermediate subplans diverse, the latter removes the worst plans. This flexibility is a direct consequence of our algebraic approach to the plan enumeration problem.

We end the evaluation of our optimizer's internals by analyzing where the time is spent throughout the optimization process. Figure **??** breaks down the runtime of our optimizer in its several phases for several tasks. At first, we note that the average optimization time amounts to slightly more than a second, which is several orders of magnitude smaller than the time savings from the previous experiments. The lion's share of the runtime is the source inspection, which obtains cardinality estimates for the source operators of a RHEEM plan (e. g., for inspecting an input file). This could be improved, e. g., by a metadata repository or via caches. In contrast, the enumeration and MCT discovery finished in the order of tens of milliseconds, even though they are of exponential complexity.

[9]This is the same pruning as in Section 7.6. However, while in Section 7.6 we used top-k pruning to *augment* our lossless pruning, here we consider it *independently*.