Optimizing Cross-Platform Data Movement

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

[1]*Qatar Computing Research Institute, HBKU*
{zkaoudi, jquianeruiz, schawla, brojas}@hbku.edu.qa

[2]*Hasso Plattner Institute, University of Potsdam*
{sebastian.kruse, felix.naumann}@hpi.de

***Abstract*—Data analytics are moving beyond the limits of a single data processing platform. A cross-platform query optimizer is necessary to enable applications to run their tasks over multiple platforms efficiently and in a platform-agnostic manner. For the optimizer to be effective, it must consider data movement costs across different data processing platforms. In this paper, we present the graph-based data movement strategy used by RHEEM, our open-source cross-platform system. In particular, we (i) model the data movement problem as a new graph problem, which we prove to be NP-hard, and (ii) propose a novel graph exploration algorithm, which allows RHEEM to discover multiple hidden opportunities for cross-platform data processing.**

I. Introduction

The research and industry communities have recently identified the need for *cross-platform data processing*: more and more applications require running their queries/tasks[1] on more than one data processing platform [3], [11], [12], [16]. Several systems have thus appeared with the goal of supporting cross-platform data processing [5], [7]–[9], [15]. The key component of all these systematic solutions is a *cross-platform query optimizer*, which finds the set of platforms that minimizes the execution cost of a given task. Performing a given task on multiple data processing platforms typically requires moving and transforming data across platforms. However, the above cross-platform solutions either do not consider data movement costs at all or do not consider different movement alternatives. As a result, they may hinder many cross-platform opportunities.

Considering data movement costs renders cross-platform query optimization quite challenging for many reasons: (i) there might be several alternative data movement strategies. A simple file-based strategy, such as in [9] and [17], may miss many opportunities for cross-platform data processing; (ii) the cost of each data movement strategy must be assessed so that the optimizer can explore the trade-off between selecting optimal execution operators and minimizing data movement costs; (iii) data movement might involve several intermediate steps to connect two operators of different processing platforms, as also stated in [15]; and (iv) data movement and transformation costs are platform-dependent.

In this paper, we delve into the data movement mechanism of RHEEM's optimizer [2], [4], an open source cross-platform system [1]. Our data movement mechanism is the first to tackle all of the above challenges. The idea is to model data movement as a graph problem and to devise an efficient graph algorithm to solve it. After giving an overview of our optimizer (Section II), we present our major contributions:

(1) We represent the space of possible communication steps as a graph and model cross-platform data movement as a new graph problem, which we prove to be NP-hard (Section III).
(2) We propose a novel graph exploration algorithm to efficiently solve this new graph problem (Section IV).
(3) We show that our data movement solution allows us to discover multiple hidden opportunities for cross-platform data processing, which were not possible before (Section V).

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

II. Cross-Platform Query Optimization

Let us first outline RHEEM [4] and its cost-based optimizer so as to establish our data movement mechanism's context.

RHEEM background. RHEEM decouples applications from data processing platforms with the goal of enabling cross-platform data processing. It receives as input a procedural RHEEM *plan*, its optimizer produces an *execution plan*, which is then given to the executor to run it. A RHEEM plan is essentially a directed data flow graph. The vertices are RHEEM *operators* and the edges represent the data flow among the operators, such as in Spark or Flink. 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. As of now, RHEEM supports neither nested loops nor control-flow operators. A complete list of the currently supported operators can be found in RHEEM's documentation [1]. Notice that only Loop operators accept feedback edges, thus enabling iterative data flows. A RHEEM plan without any loop operator is essentially a DAG. On the other side, an execution plan is also a data flow graph, but it differs from a RHEEM plan as (i) its vertices are platform-specific *execution operators* and (ii) it might comprise additional execution operators for data movement among platforms (e. g., a Broadcast operator). Conceptually, given a RHEEM plan, an execution plan indicates on which data processing platforms RHEEM must enact each RHEEM operator so that the execution cost is minimized. We refer the reader interested in further details about RHEEM to [1], [4].

Cross-platform optimizer. It is the cross-platform optimizer that allows RHEEM to produce an (platform-specific) execution plan for a given RHEEM plan. The main idea behind our optimizer 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 the total cost is minimized.

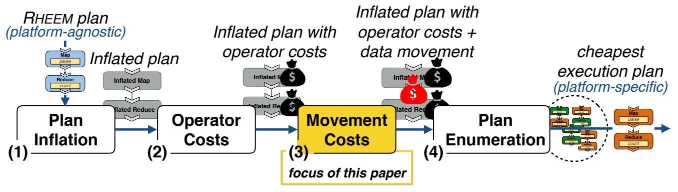

Fig. 1: Cross-platform optimization pipeline.

Figure 1 illustrates the workflow of our optimizer. **Step (1):** Given a RHEEM plan, the optimizer first *inflates* the input plan by applying a set of flexible m-to-n mappings. These mappings list 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 defines all possible combinations of executions operators of the original RHEEM plan. That is, an inflated RHEEM plan is a highly compacted representation of all execution plans. **Step (2):** The optimizer then *annotates* the resulting inflated RHEEM plan with estimates for both the costs of each execution operator and data cardinalities. The total cost estimate for an execution operator o depends on the cost of the resources it consumes (CPU, memory, disk, and network), defined as: $cost_o = t_o^{CPU} + t_o^{mem} + t_o^{disk} + t_o^{net}$. For instance, the cost function to estimate the CPU cycles required by the SparkFilter operator is $CPU_{SF} := c_{in}(\text{Filter}) \times \alpha + \beta$, where parameter c_{in} denotes the input cardinality of operator o, α denotes the number of required CPU cycles for each input tuple, and parameter β describes some fixed overhead for the operator start-up. RHEEM provides a cost learner to calibrate all cost parameters, such as α and β. **Step (3)**: Next, the optimizer takes a graph-based approach to determine how to move data efficiently between data processing platforms and annotates the inflated RHEEM plan with the corresponding costs. In this paper, we focus on this aspect. **Step (4)**: The optimizer uses all these annotations to determine the optimal execution plan via an enumeration algorithm.

III. Data Movement as a Graph Problem

In contrast to federated databases, a cross-platform setting typically has completely different data formats. Hence, different data transformations might take place to move data, which makes planning and assessing communication in cross-platform settings a unique problem. First, there might exist several alternatives to move data, e. g., from RDD to a file or to a Java object. Second, for a cross-platform optimizer to be effective, it must assess the cost of each strategy, including the strategy of not changing the platform. Third, data movement might involve several intermediate steps. We thus model the problem of finding the most efficient communication path among execution operators as a new graph problem.

Channel conversion graph. We represent the space of possible communication steps as a channel conversion graph (CCG for short). The vertices in a CCG are data structures (communication *channels*), and the edges are data transfer and transformations (*conversion operators*) from one data structure to another. In detail, data flows among operators via channels, which can be, for example, an internal data structure or a stream within a platform, or simply an external file. The yellow boxes in Figure 2 depict some standard channels in Java Streams, Postgres, Spark, and Flink. Note that channels can be *reusable*, i. e., they can be consumed multiple times, or non-reusable, i. e., they can be consumed once. For instance, a file is reusable while a stream is usually not. Conversion operators handle any necessary data transformation to move data from between two channels, e. g., convert an SQL query result to a data stream. They also deal with semantic integration issues, such as transforming data from one format to another (e. g., from CSV to TSV). Conversion operators are in fact regular execution operators. The benefit of using conversion operators for both data transfer and transformation is twofold: (i) there is less overhead in the execution pipeline; and (ii) as conversion operators are in fact regular execution operators, the conversion costs do not require a new cost model.

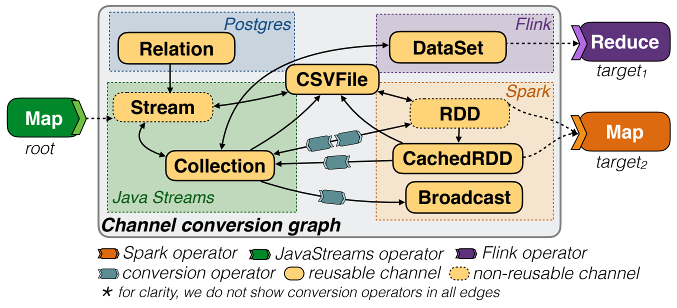

Fig. 2: A channel conversion graph along with root and target operators from different platforms.

Definition 1 (Channel conversion graph): A *CCG* is a directed graph $G := (C, E, \lambda)$, where the set of vertices C is 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 $e \in E$.

CCG yields another major benefit: it allows us to model the problem of finding the most efficient communication path among execution operators 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. Let us further motivate the utility of CCGs for data movement with a concrete example.

Example 1: Figure 2 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 FlinkReduce (*target*$_1$) and a SparkMap (*target*$_2$) execution operator. While the *root* produces a Java Stream as output channel, $target_1$ and $target_2$ accept only a Flink DataSet and a (cached) RDD, respectively, as input channels. Multiple conversions are needed to serve the two target operators.

Minimum conversion tree. We model such scenarios of finding the most efficient communication path from a root producer 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' 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}, for $c_{t_i} \in C_{t_i}$;*
***(2)** any non-reusable channel in G', must have a single successor, i. e., a conversion or a consumer operator;*
***(3)** there is no other subgraph G'' that satisfies the above two conditions and has a smaller cost (i. e., the sum of costs of all its edges) than G'. The cost of an edge e is the estimated cost for the associated conversion operator $\lambda(e)$.*

Example 2: In Figure 2, the root channel is $c_r := \mathsf{Stream}$ and the target channel sets are $C_{t_1} := \{\mathsf{DataSet}\}$ (for target_1) and $C_{t_2} := \{\mathsf{RDD}, \mathsf{CachedRDD}\}$ (for target_2). The minimum conversion tree for this scenario could be: The Stream root channel is converted to a Java Collection, which is converted twice, namely to a Flink DataSet (thereby satisfying C_{t_1}) and to an RDD (thereby satisfying C_{t_2}). Note that this is possible only because Collection is reusable.

Theorem 1: The MCT problem is NP-hard.

Due to space limitations, all proofs can be found in [13].

Related work. The MCT problem seems related to the minimum spanning tree and single-source multiple-destinations shortest paths, but it differs from them for two main reasons. First, MCTs have a fixed root and need not span the whole CCG. Second, MCT seeks to minimize the costs of the conversion tree as a whole rather than its individual paths from the root to the target channels. It is the Group Steiner Tree (GST) problem [14] that is the closest to our MCT problem. 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. Moreover, [10] focuses on providing new channels to move data more efficiently, rather than finding the best path for data movement. Thus, it is complementary to our work.

IV. DATA MOVEMENT ALGORITHM

We introduce MCTfinder, our algorithm to solve the MCT problem. Given a CCG G, a root channel c_r, and n target channel sets $\mathscr{C}_t := \{C_{t_1}, ..., C_{t_n}\}$, MCTfinder proceeds in two steps: (i) it simplifies the problem by modifying the input parameters (*kernelization*). (ii) it exhaustively explores the graph (*channel conversion graph exploration*) to find the MCT.

Kernelization. In the frequent case that several target consumers, e. g., target_i and target_j, accept the same channels, $C_{t_i} = C_{t_j}$, 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 rationale is that it is most cost-efficient to find a single reusable channel that serves as many consumers as possible. Doing so decreases the number of target channel sets and thus, reduces the maximum degree (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, the MCT problem becomes a single-source single-destination shortest path problem. We can thus solve it with, e. g., Dijkstra's algorithm.

Example 3 (Merging target channel sets): In Figure 2, target_2 accepts the channels $C_{t_2} = \{\mathsf{RDD}, \mathsf{CachedRDD}\}$. Assuming that target_1 is a *SparkReduce* operator instead, which accepts the same set of channels as target_2, we could then merge their input channels into $C_{t_{1,2}} = \{\mathsf{CachedRDD}\}$.

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

Channel conversion graph exploration. After kernelization, MCTfinder proceeds to explore the CCG, thereby building the MCT from "its leaves to the root": Intuitively, it 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. Overall, MCTfinder's graph traversal is composed of three main parts: (i) it visits a new channel, checks if it belongs to any target channel set, and potentially creates a partial singleton conversion tree; (ii) then it traverses forward, thereby creating *partial* MCTs from the currently visited channel to any subset of target channel sets; and (iii) it merges the partial MCTs from the steps (i) and (ii) and returns the *merged* MCTs. The algorithm terminates when the partial MCTs form the final MCT.

Theorem 2: Given a CCG, MCTfinder finds the minimum conversion tree if it exists.

Complexity and correctness. MCTfinder solves the MCT problem exactly, which 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, MCTfinder 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 kernelization. More importantly, MCTfinder avoids performance penalties from inferior data movement plans. Still, one may consider making it approximate, inspired from existing algorithms for the Group Steiner Tree problem [6].

V. VALIDATION

Our data movement solution is an integral part of the optimizer of RHEEM, our open-source cross-platform system [1]. For the sake of simplicity, we henceforth refer to the optimizer simply as RHEEM. We demonstrate how our solution enables RHEEM to: (i) *spot hidden opportunities for cross-platform processing that improve performance*, (ii) *outperform the common practice of moving data through files*, and (iii) *still scale graceful for large* RHEEM *plans*.

A. Setup

We ran all our experiments on a cluster of 10 machines: each with one 2 GHz Quad Core Xeon processor, 32 GB main memory, 500 GB SATA hard disks, a 1 Gigabit network card

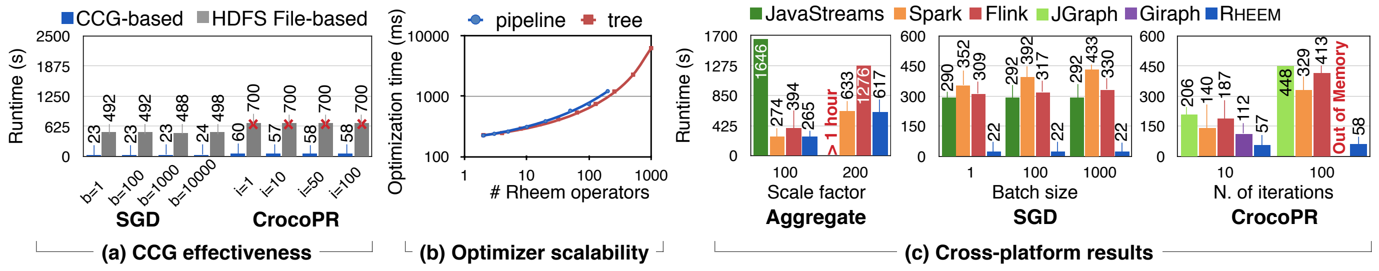

Fig. 3: Experimental results.

running the 64-bit platform Linux Ubuntu 14.04.05. We considered three tasks from three different domains: `Aggregate` (TPC-H Q1 - OLAP), `SGD` (stochastic gradient descent - ML), and `CrocoPR` (cross-community pagerank - Graph Mining). These tasks run over three different datasets: `Aggregate` on TPC-H data, `SGD` on HIGSS (7.4GB), and `CrocoPR` on a subset of DBpedia (2GB). We considered the following platforms: Java's Streams (JavaStreams), Spark 2.4.0 (Spark), Flink 1.7.1 (Flink), Giraph 1.2.0 (Giraph), a self-written Java graph library (JGraph), and HDFS 2.6.5 to store all datasets. All the numbers we report are the average of three runs.

B. Results

Figure 3(a) shows the effectiveness of our graph-based data movement solution when compared to the common practice of moving data via files. We observe that for `SGD` and `CrocoPR`, it is always more than one order of magnitude faster, in fact for `CrocoPR`, we decided to kill the process after 700 seconds. We saw similar results for `Aggregate`, but we omit them due to space limitations. This clearly shows the importance of our data movement approach.

Figure 3(b) illustrates the optimization time of our optimizer when increasing the number of RHEEM operators. For this, we generated synthetic RHEEM plans with two topologies we found to be at the core of many data analytic tasks: *pipeline* and *tree*. The results show the high efficiency of our optimizer for both topologies: it optimized plans with 100 operators in less than 1 sec and with 1,000 in only 6 secs, even though finding a plan with 1,000 operators is very unlikely.

Now let us show how our data movement mechanism helps RHEEM to efficiently combine multiple platforms. Figure 3(c) reports these results. Overall, we find that RHEEM outperforms all single-platform executions: It is up to 20$\times$ faster than Spark, up to 15$\times$ faster than Flink, up to 13$\times$ faster than JavaStreams, up to 2$\times$ faster than Giraph. There are several reasons for having this large improvement. For `Aggregate`, RHEEM selects Flink in combination with Spark, which allows it to run this task slightly faster than the fastest baseline platform. It 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. For `SGD`, RHEEM handles the model parameters, which is typically tiny ($\sim$0.1KB for our input 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, JGraph, and JavaStreams, even if Giraph is the fastest baseline platform (for 10 iterations). This happens because after the preparation phase of this task, the input dataset for the PageRank operation on JGraph is $\sim$544 MB only. These results clearly show that RHEEM can spot hidden cross-platform opportunities that improve performance significantly.

VI. CONCLUSION

We presented a cross-platform data movement mechanism that finds the best way to transfer data among multiple data processing platforms. In particular, we modeled the problem as a new graph problem and proposed an efficient and scalable algorithm to solve it. Our evaluation showed that our data movement mechanism enabled RHEEM's optimizer to discover cross-platform opportunities, which were not possible before.

REFERENCES

[1] Rheem project: http://da.qcri.org/rheem.
[2] D. Agrawal et al. Rheem: Enabling Multi-Platform Task Execution. In *SIGMOD*, pages 2069–2072, 2016.
[3] D. Agrawal et al. Road to freedom in big data analytics. In *EDBT*, pages 479–484, 2016.
[4] D. Agrawal et al. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! *PVLDB*, 11(11):1414–1427, 2018.
[5] E. Begoli et al. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*, pages 221–230, 2018.
[6] C. Chekuri et al. A greedy approximation algorithm for the Group Steiner problem. *Discrete Applied Mathematics*, 154(1):15–34, 2000.
[7] K. Doka et al. Mix 'n' match multi-engine analytics. In *IEEE BigData*, pages 194–203, 2016.
[8] A. Elmore et al. A demonstration of the BigDAWG polystore system. *PVLDB*, 8(12):1908–1911, 2015.
[9] I. Gog et al. Musketeer: All for one, one for all in data processing systems. In *EuroSys*, pages 1–16, 2015.
[10] B. Haynes et al. PipeGen: Data pipe generator for hybrid analytics. In *SoCC*, pages 470–483, 2016.
[11] Z. Kaoudi et al. A Cost-based Optimizer for Gradient Descent Optimization. In *SIGMOD*, pages 977–992, 2017.
[12] Z. Kaoudi and J.-A. Quiané-Ruiz. Cross-Platform Data Processing: Use Cases and Challenges. In *ICDE (tutorial)*, 2018.
[13] S. Kruse et al. RHEEMix in the Data Jungle – A Cross-Platform Query Optimizer. arXiv: 1805.03533 https://arxiv.org/abs/1805.03533, 2018.
[14] 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.
[15] A. Simitsis et al. Optimizing analytic data flows for multiple execution engines. In *SIGMOD*, pages 829–840, 2012.
[16] M. Stonebraker. The case for polystores. ACM SIGMOD Blog, 2015.
[17] J. Wang et al. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*, 2017.