RHEEM: Enabling Cross-Platform Data Processing

– May The Big Data Be With You! –

Divy Agrawal[2*] Sanjay Chawla[1] Bertty Contreras-Rojas[1] Ahmed Elmagarmid[1] Yasser Idris[1]
Zoi Kaoudi[1] Sebastian Kruse[3†] Ji Lucas[1] Essam Mansour[1] Mourad Ouzzani[1] Paolo Papotti[4*]
Jorge-Arnulfo Quiané-Ruiz[1] Nan Tang[1] Saravanan Thirumuruganathan[1] Anis Troudi[1]

[1]Qatar Computing Research Institute (QCRI), HBKU [2]UCSB [3]Hasso Plattner Institute (HPI) [4]Eurecom

http://da.qcri.org/rheem/

ABSTRACT

Solving business problems increasingly requires going beyond the limits of a single data processing platform (platform for short), such as Hadoop or a DBMS. As a result, organizations typically perform tedious and costly tasks to juggle their code and data across different platforms. Addressing this pain and achieving automatic cross-platform data processing is quite challenging: finding the most efficient platform for a given task requires quite good expertise for all the available platforms. We present RHEEM, a general-purpose cross-platform data processing system that decouples applications from the underlying platforms. It not only determines the best platform to run an incoming task, but also splits the task into subtasks and assigns each subtask to a specific platform to minimize the overall cost (e.g., runtime or monetary cost). It features (i) a robust interface to easily compose data analytic tasks; (ii) a novel cost-based optimizer able to find the most efficient platform in almost all cases; and (iii) an executor to efficiently orchestrate tasks over different platforms. As a result, it allows users to focus on the business logic of their applications rather than on the mechanics of how to compose and execute them. Using different real-world applications with RHEEM, we demonstrate how cross-platform data processing can accelerate performance by more than one order of magnitude compared to single-platform data processing.

PVLDB Reference Format:
Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. RHEEM: Enabling Cross-Platform Data Processing – May The Big Data Be With You!. *PVLDB*, 11 (11): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3236187.3236195

*Work done while working at QCRI.

†Work partially done while interning at QCRI.

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236195

1. THE DARK SIDE OF BIG DATA

The pursuit of comprehensive, efficient, and scalable data analytics as well as the *one-size-does-not-fit-all* dictum have given rise to a plethora of data processing platforms (*platforms* for short). These specialized platforms include DBMS, NoSQL, and MapReduce-like platforms. In fact, just under the umbrella of NoSQL, there are reportedly over 200 different platforms[1]. Each excels in specific aspects allowing applications to achieve high performance and scalability. For example, while Spark supports `Select` queries, Postgres can execute them much faster by using indices. However, Postgres is not as good as Spark for general purpose batch processing where parallel full scans are the key performance factor. Several studies have shown this kind of performance differences [20, 34, 40, 53, 61].

Diversity as Common Ground. Moreover, today's data analytics is moving beyond the limits of a single platform. For example: (i) IBM reported that North York hospital needs to process 50 diverse datasets, which run on a dozen different platforms [38]; (ii) Airlines need to analyze large datasets, which are produced by different departments, are of different data formats, and reside on multiple data sources, to produce global reports for decision makers [9]; (iii) Oil & Gas companies need to process large amounts of diverse data spanning various platforms [19,36]; (iv) Several data warehouse applications require data to be moved from a MapReduce-like system into a DBMS for further analysis [28, 56]; (v) Business intelligence typically requires an analytic pipeline composed of different platforms [58]; and (vi) Using multiple platforms for machine learning improves performance significantly [20, 40].

Status Quo. To cope with these new requirements, developers (or data scientists) have to write ad-hoc programs and scripts to integrate different platforms. This is not only a tedious, time-consuming, and costly task, but it also requires knowledge of the intricacies of the different platforms to achieve high efficiency and scalability. Some systems have appeared with the goal of facilitating platform integration [2, 4, 10, 12]. Nonetheless, they all require a good deal of expertise from developers, who still need to decide which processing platforms to use for each task at hand. Recent research has taken steps towards transparent cross-platform execution [15, 29, 34, 46, 58, 60], but lacks several important aspects. Usually these efforts do not automatically map

[1]`http://db-engines.com`

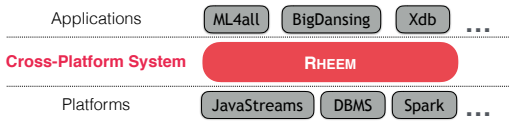

Figure 1: Rheem in the data analytics stack.

tasks to platforms. Additionally, they do not consider complex data movement (i.e., with data transformations) among platforms [29, 34]. Finally, most of the research focuses on specific applications [15, 46, 58].

Cross-Platform Data Processing. There is a clear need for a systematic approach to enable efficient *cross-platform data processing*, i.e., use of multiple data processing platforms. The Holy Grail would be to replicate the success of DBMSs for cross-platform data processing. Users simply send their tasks (or queries) expressing the logic of their applications, and the cross-platform system decides on which platform(s) to execute each task with the goal of minimizing its cost (e.g., runtime or monetary cost).

Challenges. Building such a cross-platform system is challenging on numerous fronts: (i) a cross-platform system not only has to effectively find all the suitable platforms for a given task, but also has to choose the most efficient one; (ii) cross-platform settings are characterized by high uncertainty as different platforms are autonomous and thus one has little control over them; (iii) the performance gains of using multiple platforms should compensate the added cost of moving data across platforms; (iv) it is crucial to achieve inter-platform parallelism to prevent slow platforms from dominating execution time; (v) the system should be extensible to new platforms and application requirements; and (vi) the system must be easy to use so that the development of data analytic tasks can be sped up.

Contributions. We present RHEEM[2], the first general-purpose cross-platform system to tackle all of the above challenges. The goal of RHEEM is to *enable* applications and users to run data analytic tasks *efficiently* on one or more data processing platforms. To do so, it decouples applications from platforms as shown in Figure 1. Applications issue their tasks to RHEEM, which in turn decides where to execute them. As of today, RHEEM supports a variety of platforms: Spark, Flink, JavaStreams, Postgres, GraphX, GraphChi, and Giraph. We are currently testing RHEEM in a large international airline company and in a biomedical research institute. In the former case, we aim at seamlessly integrating all data analytic activity governing an aircraft; In the latter case, we aim at reducing the effort scientists need for building data analytic pipelines while at the same time speeding up the running time. Note that several papers show different aspects of RHEEM: the vision behind it [17]; its optimizer [43]; its inequality join algorithm [42]; and a couple of its applications [40, 41]. A couple of demo papers showcase the benefits of RHEEM [16] and its interface [47]. This paper aims at presenting the complete design of RHEEM and how all its pieces work together.

In summary, we make the following contributions:

(1) We identify four situations in which applications require support for cross-platform data processing. For each case, we use a real application to show experimentally the benefits of cross-platform data processing using RHEEM. (Section 2)

(2) We present the data and processing model of RHEEM and show how it shields users from the intricacies of the underlying platforms. RHEEM provides flexible operator mappings that allow for better exploiting the underlying platforms. Also, its extensible design allows users to add new platforms and operators with very little effort. (Section 3)

(3) We discuss the key aspects that make RHEEM novel. It is the first to: (i) use a cost-based cross-platform optimizer that considers data movement costs; (ii) offer a progressive optimization mechanism to deal with inconsistent cardinality estimates; and (iii) provide a learning tool that alleviates the burden of tuning the cost model. (Section 4)

(4) We present the RHEEM interfaces whereby users can easily code and run any data analytic task on any data processing platform. In particular, we present a data-flow language (RheemLatin) and a visual integrated development environment (RHEEM Studio). (Section 5)

(5) We present additional experiments showing that RHEEM achieves its goals in terms of feasibility and performance improvements. RHEEM allows applications to run up to more than one order of magnitude faster than baselines and common practices. (Section 6)

(6) We summarize the lessons learned from our journey and show how these impacted the applications we built on top of RHEEM. We also discuss RHEEM's limitations. (Section 7)

Moreover, we discuss related work in Section 8 and conclude with some open problems in Section 9.

2. CROSS-PLATFORM PROCESSING

We identified four situations in which an application requires support for cross-platform data processing [39].

(1) *Platform-independence.* Applications run an entire task on a single platform but may require switching platforms for different input datasets or tasks usually with the goal of achieving better performance.

(2) *Opportunistic cross-platform.* Applications might also benefit performance-wise from using multiple platforms to run one single task.

(3) *Mandatory cross-platform.* Applications may require multiple platforms because the platform where the input data resides, e.g., PostgreSQL, cannot perform the incoming task, e.g., a machine learning task. Thus, data should be moved from the platform it resides to another platform.

(4) *Polystore.* Applications may require multiple platforms because the input data is stored on multiple data stores.

In contrast to existing systems [29, 30, 34, 58, 62], RHEEM helps users in *all* above cases. The design of our system has been mainly driven by four applications: a data cleaning application, *BigDansing* [41]; a machine learning application, *ML4all* [40]; a database application, *xDB*; and an end-to-end data discovery and preparation application, *Data Civilizer* [32]. We use these applications to showcase the benefits of performing cross-platform data processing, instead of single-platform data processing, in terms of both performance and ease of use. For clarity, the setup details of our below experiments are in Section 6, where we present additional experiments demonstrating the benefits of RHEEM.

[2]RHEEM is open source under the Apache Software License 2.0 and can be found at `https://github.com/rheem-ecosystem/rheem`.

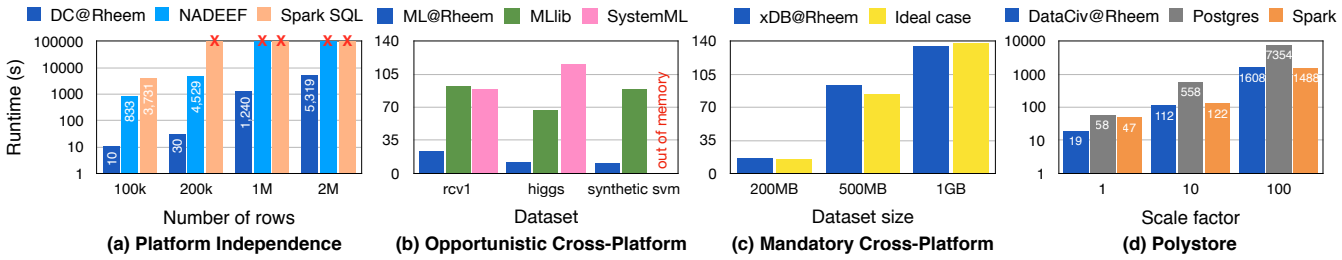

Figure 2: Benefits of the cross-platform data processing approach (using Rheem).

2.1 Platform Independence

Applications are usually tied to a specific platform. This may not constitute the ideal case for two reasons. First, as more efficient platforms become available, developers need to re-implement existing applications on top of these new platforms. For example, Spark SQL [14] and MLlib [13] are the Spark counterparts of Hive [6] and Mahout [7]. Migrating an application from one platform to another is a time-consuming and costly task and hence it is not always a viable choice. Second, for different inputs of a specific task, a different platform may be the most efficient one, so the best platform cannot be determined statically. For instance, running a specific task on a big data platform for very large datasets is often a good choice, while single-node platforms with only little overhead costs are often a better choice for small datasets [20]. Thus, enabling applications to seamlessly switch from one platform to another according to the input dataset and task is important. Rheem *dynamically determines the best platform to run an incoming task.*

Benefits. We use BigDansing [41] to demonstrate the benefits of providing platform independence. Users specify a data cleaning task with five logical operators: Scope (identifies relevant data), Block (defines the group of data among which an error may occur), Iterate (enumerates candidate errors), Detect (determines whether a candidate error is indeed an error), and GenFix (generates a set of possible repairs). Rheem maps these operators to Rheem operators to decide the best underlying platform. We show the power of supporting cross-platform data processing by running an error detection task on a widely used Tax dataset [31]. The task is based on the denial constraint $\forall t_1, t_2, \neg(t_1.\text{Salary} > t2.\text{Salary} \wedge t_1.\text{Tax} < t_2.\text{Tax})$, which states that there is an inconsistency between two tuples representing two different persons if one earns a higher salary but pays a lower tax. We considered NADEEF [25], a data cleaning tool, and SparkSQL, a general-purpose framework, as baselines and forced Rheem to use either Spark or JavaStreams per run.

Figure 2(a) shows the results (the red cross means we stopped the execution after 40 hrs). Overall, we observe that Rheem (DC@Rheem) allows data cleaning tasks to scale up to large datasets and be at least three orders of magnitude faster than baselines. One order of magnitude gain comes from the ability of Rheem to automatically switch platforms. Rheem used JavaStreams for small datasets speeding up the data cleaning task by avoiding Spark's overhead, while it used Spark for the largest datasets. Furthermore, in contrast to SparkSQL that cannot process inequality joins efficiently, Rheem's extensibility allowed us to plug in a more efficient inequality-join algorithm [42], thereby further improving over these baselines. In a nutshell, BigDansing benefited from Rheem because of its ability to effectively switch platforms and because of its extensibility to easily plug optimized algorithms. We demonstrated how BigDansing benefits from Rheem in [16].

2.2 Opportunistic Cross-Platform

While some applications can be executed on a single platform, there are cases where their performance would be sped up by using multiple platforms. For instance, users can run a gradient descent algorithm, such as SGD, on top of Spark relatively fast. Still, we recently showed that mixing it with JavaStreams significantly improves performance [40]. In fact, opportunistic cross-platform processing can be seen as the execution counter-part of *polyglot persistence* [55], where different types of databases are combined to leverage their individual strengths. However, developing such cross-platform applications is difficult: developers must know all the cases where it is beneficial to use multiple platforms and how exactly to use them. These opportunities are often very hard (if not impossible) to spot. Even worse, like in the platform independence case, they usually cannot be determined a priori. Rheem *finds and exploits opportunities of using multiple processing platforms.*

Benefits. Let us now take our machine learning application, *ML4all* [40], to showcase the benefits of using multiple platforms to perform one single task. ML4all abstracts three fundamental phases (namely preparation, processing, and convergence) found in most machine learning tasks via seven logical operators which are mapped to Rheem operators. In the preparation phase, the dataset is prepared appropriately along with the necessary initialization of the algorithm (`Transform` and `Stage` operators). The processing phase computes the gradient and updates the current estimate of the solution (`Sample`, `Compute`, and `Update` operators) while the convergence phase repeats the processing phase based on the number of iterations or other criteria (`Loop` and `Converge` operators). We demonstrate the benefits of using Rheem with a classification task over three benchmark datasets, using Stochastic Gradient Descent (SGD).

Figure 2(b) shows the results. We observe that, even though all systems use the same SGD algorithm, Rheem allows this algorithm to run significantly faster than competing Spark-based systems. This is because of two main reasons. First, this comes from opportunistically running the Compute, Update, Converge, and Loop operators on JavaStreams, thereby avoiding some of the Spark's overhead. Rheem runs the rest of the operators on Spark. MLlib and SystemML do not avoid such overhead by purely using Spark for the entire algorithm. Second, ML4all leverages Rheem's

extensibility to plug an efficient sampling operator, resulting in significant speedups. We demonstrated how ML4all further benefits from RHEEM in [16].

2.3 Mandatory Cross-Platform

There are cases where an application needs to go beyond the functionalities offered by the platform on which the data is stored. For instance, a dataset is stored on a relational database and a user needs to perform a clustering task on particular attributes. Doing so inside the relational database might simply be disastrous in terms of performance. Thus, the user needs to move the projected data out of the relational database and, for example, put it on HDFS in order to use Apache Flink [3], which is known to be efficient for iterative tasks. A similar situation occurs in complex data analytics applications with disparate subtasks. As an example, an application that extracts a graph from a text corpus to perform subsequent graph analytics may require using both a text and a graph analytics system. The required integration of platforms is tedious, repetitive, and particularly error-prone. Nowadays, developers write ad-hoc programs to move the data around and integrate different platforms. RHEEM *not only selects the right platforms for each task but also moves the data if necessary at execution time.*

Benefits. We use xDB[3], a system on top of RHEEM with database functionalities, to demonstrate the benefits of performing cross-platform data processing for the above situation. It provides a declarative language to compose data analytic tasks, while its optimizer produces a plan to be executed in RHEEM. We evaluate the benefits of RHEEM with the cross-community pagerank[4] task, which is not only hard to express in SQL but also inefficient to run on a DBMS. Thus, it is important to move the computation to another platform. In this experiment, the input datasets are on Postgres and RHEEM moves the data into Spark.

Figure 2(c) shows the results. As a baseline, we consider the ideal case where the data is on HDFS and RHEEM simply uses either JavaStreams or Spark to run the tasks. We observe that RHEEM allows xDB (xDB@Rheem) to achieve similar performance with the ideal case in all the situations, while fully automating the process. This is a remarkable result as RHEEM needs to move data out of Postgres to perform the tasks, in contrast to the ideal case.

2.4 Polystore

In many organizations, data is collected in different formats and on heterogeneous storage platforms (*data lakes*). Typically, a data lake comprises various DBMSs, document stores, key-value stores, graph databases, and pure file systems. As most of these stores are tightly coupled with an execution engine, e.g., a DBMS, it is crucial to be able to run analytics over multiple platforms. For this, users perform not only tedious, time-intensive, and costly data migration, but also complex integration tasks for analyzing the data. RHEEM *shields the users from all these tedious tasks and allows them to instead focus on the logic of their applications.*

Benefits. A clear example that shows the benefits of cross-platform data processing in a polystore case is the Data Civilizer system [32]. Data Civilizer is a big data management system for data discovery, extraction, and cleaning from data lakes in large enterprises [27]. It constructs a graph that expresses relationships among data existing in heterogeneous data sources. Data Civilizer uses RHEEM to perform complex tasks over information that spans multiple data storages. We measure the efficiency of RHEEM for these polystore tasks with TPC-H query 5. In this experiment, we assume that the data is stored in HDFS (LINEITEM and ORDERS), Postgres (CUSTOMER, REGION, and SUPPLIER), and a local file system (NATION). Thus, this task performs join, groupby, and orderby operations across three different platforms. In this scenario, the common practice is to move the data into the database to enact the queries inside the database [28,56] or move the data entirely to HDFS and use Spark. We consider these two practices as the baseline. For a fairer comparison, we also set the "parallel query" and "effective IO concurrency" features of Postgres to 4.

Figure 2(d) shows the results. RHEEM (DataCiv@Rheem) is significantly faster, namely up to 5×, than the current practice. 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 perform quite similarly to the parallel version of Postgres. The pure execution time in Postgres for scale factor 100 amounts to 1,541 sec compared to 1,608 sec for RHEEM, which exploits Spark for data parallelism. We also observe that RHEEM has negligible overhead over the case where the developer writes ad-hoc scripts to move the data to HDFS for running the task on Spark. In particular, RHEEM is twice faster than Spark for scale factor 1 because it moves less data from Postgres to Spark.

3. RHEEM MODEL

First of all, let us emphasize that RHEEM is *not* yet another data processing platform. On the contrary, it is designed to work between applications and platforms (as shown in Figure 1), helping applications to choose the right platform(s) for a given task. RHEEM is the first general-purpose cross-platform system that shields users from the intricacies of the underlying platforms and let them focus only on the logic of their applications. We define the RHEEM data and processing models in the following.

Data Quanta. The RHEEM data model relies on *data quanta*, the smallest processing units from the input datasets. A data quantum can express a large spectrum of data formats, such as database tuples, edges in a graph, or the full content of a document. This flexibility allows applications and users to define a data quantum at any granularity level, e.g., at the attribute level rather than at the tuple level for a relational database. This fine-grained data model allows RHEEM to work in a highly parallel fashion, if necessary, to achieve better scalability and performance.

Rheem Plan. RHEEM accepts as input a RHEEM *plan*: a directed data flow graph whose vertices are RHEEM *operators* and whose edges represent data flows among the operators. A RHEEM operator is a platform agnostic data transformation over its input data quanta, e.g., a Map operator transforms an individual data quantum while a Reduce operator aggregates input data quanta into a single output data quantum. Only Loop operators accept feedback edges, which allows iterative data flows to be expressed. Users

[3] `https://github.com/rheem-ecosystem/xdb`

[4] This task basically intersects two community-DBpedia datasets and runs pagerank on the resulting dataset.

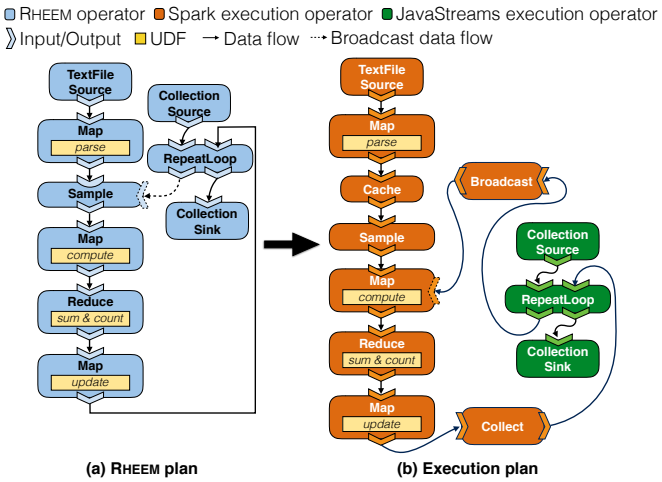

Figure 3: SGD example.

or applications can refine the behavior of operators with a UDF. Optionally, applications can also attach the selectivities of the operators through a UDF. RHEEM comes with default selectivity values in case they are not provided. A RHEEM plan must have at least one source operator, i.e., an operator reading or producing input data quanta, and one sink operator per branch, i.e., an operator retrieving or storing the result. Intuitively, data quanta are flowing from source to sink operators, thereby being manipulated by all inner operators. As our processing model is based on primitive operators, RHEEM plans are highly expressive. This is in contrast to other systems that accept either declarative queries [34, 62] or coarse-granular operators [29].

EXAMPLE 1. *Figure 3(a) shows a* RHEEM *plan for the stochastic gradient descent algorithm (SGD). Initially, the dataset containing the data points is read via a* TextFileSource *operator and parsed using a* Map *operator while the initial weights are read via a* Collection *source operator. After the* RepeatLoop *operator, the weights are fed to the* Sample *operator, where a set of input data points is sampled. Next,* Map(compute) *computes the gradient for each sampled data point. Note that as* Map(compute) *requires all weights to compute the gradient, the weights are broadcasted at each iteration to the* Sample *operator (denoted by the dotted line). Then, the* Reduce *operator computes the sum and count of all gradients. The next* Map *operator uses these sum and count values to update the weights. This process is repeated until the loop condition is satisfied. The resulting weights are output in a collection sink.*

Execution Plan. Given a RHEEM plan as input, RHEEM uses a cost-based optimization approach to produce an *execution plan* by selecting one or more platforms to efficiently execute the input plan. The cost can be any user-specified cost, e.g., runtime or monetary cost. The resulting execution plan is again a data flow graph, where the vertices are now *execution operators*. An execution operator implements one or more RHEEM operators with platform-specific code. For instance, the Cache Spark execution operator in RHEEM implements the Cache RHEEM operator by calling the RDD.cache() operation of Spark. An execution plan may also comprise additional execution operators for data movement (e.g., data broadcasting) or data reuse (e.g., data caching). Additionally, each execution operator has attached a UDF where its cost is specified. RHEEM learns such costs from execution logs using machine learning. We discuss more details in Section 4.5.

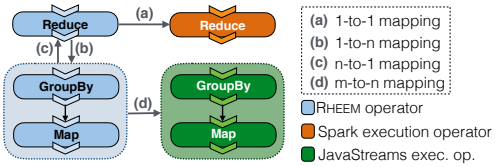

Figure 4: Operator mappings.

EXAMPLE 2. *Figure 3(b) shows the SGD execution plan produced by* RHEEM *when Spark and JavaStreams are the only available platforms. This execution plan exploits high parallelism for the large dataset of input data points and avoids the extra overhead incurred by big data processing platforms for the smaller collection of weights. Note that the execution plan also contains three execution operators for transferring (*Broadcast, Collect*) and making data quanta reusable across the platforms (*Cache*).*

Operator Mappings. To produce an execution plan, RHEEM relies on flexible *m-to-n mappings* to map RHEEM operators to execution operators. Supporting *m*-to-*n* mappings is particularly useful as it allows to map whole subplans of RHEEM operators to subplans of execution operators. Additionally, a subplan of RHEEM (or execution) operators can map to another subplan of RHEEM (respectively execution) operators. As a result, we can handle different abstraction levels among platforms, e.g., to emulate RHEEM operators that are not natively supported by a specific platform. This is not possible in other systems, such as [29].

EXAMPLE 3. *Figure 4 illustrates the mapping for the* Reduce RHEEM *operator. This operator directly maps to the* Reduce *Spark execution operator via a 1-to-1 mapping (mapping (a)). However, it does not have a direct mapping to a JavaStreams execution operator. Instead, it maps to a set of* RHEEM *operators (*GroupBy *and* Map*) via a 1-to-n mapping (mapping (b)) and vice-versa (n-to-1 mapping (c)). In turn, this set of* RHEEM *operators maps to a set of JavaStreams execution operators (*GroupBy *and* Map*) via an m-to-n mapping (mapping (d)).*

Data movement. Data flows among operators via *communication channels* (or simply *channels*). A channel can be any internal data structure within a data processing platform (e.g., RDD for Spark or Collection for JavaStreams), or simply a file. In the case of two execution operators of different platforms connected within a plan, it is necessary to convert the output channel of one to the input channel of the other (e.g., from RDD to Collection). These conversions are handled by conversion operators, which in fact are regular execution operators. For example, we can convert a Spark RDD channel to a JavaStreams Collection channel using the SparkCollect operator (see Figure 3(b)). We represent the space of data movement paths across all platforms as a *channel conversion graph*, where the channels form its vertices and the *conversion operators* form its directed edges connecting one source channel to a target channel. The channel conversion graph size depends on the number of available platforms and their available channels.

We expect to have at least one channel (vertex) per platform. Our optimizer is responsible for selecting the optimal path to connect two (or more) channels (see Section 4.1 for more details). Unlike other approaches [29, 34], developers do not need to provide conversion operators for all combinations of source and target channels. It is therefore much easier for developers to add new platforms to RHEEM.

Extensibility. We have designed RHEEM to address extensibility as a first-class citizen rather than as "nice-to-have" feature. Users add new RHEEM and execution operators by merely extending or implementing few abstract classes/interfaces. RHEEM provides template classes to facilitate the development for different operator types. Users also add operator mappings by simply implementing an interface and specifying a graph pattern that matches the RHEEM operator. As a result, users can plug a new platform by providing: (i) its execution operators and their mappings and (ii) the communication channels that are specific to the new platform (e.g., RDDChannel for Spark) with at least one conversion operator from the new channel to an existing one and vice versa. Very often these operators are simply source and sink operators that the developer needs to implement anyway (e.g., SparkCollect). For instance, when adding a new platform, e.g., an array database, it is sufficient to provide one conversion operator to transform the channel of the new platform to the channel of one of the supported platforms, e.g., a relational database. Then, using the channel conversion graph the optimizer can find a path to convert the new channel to any channel of the supported platforms, e.g., a graph database. Certainly, providing a new conversion operator may improve performance but it is not a necessary condition for the system to work. Users neither have to modify the RHEEM code nor integrate the newly added platform with all the already supported platforms. Thus, RHEEM reduces the "effort complexity" of adding n new platforms to a cross-platform system with already m registered platforms from quadratic $O((n+m)^2)$ to linear $O(n)$.

4. RHEEM INTERNALS

In this section, we give the details of the RHEEM internals. Figure 5 depicts the RHEEM ecosystem, i.e., the RHEEM core architecture together with three main applications built on top of it. Users provide a RHEEM plan to the system (Step (1) in Figure 5), using Java, Scala, Python, REST, RheemLatin, or RHEEM Studio API (yellow boxes in Figure 5). The *cross-platform optimizer* compiles the RHEEM plan into an execution plan (Step (2)), which specifies the processing platforms to use; the *executor* schedules the resulting execution plan on the selected platforms (Step (3)); the *monitor* collects statistics and checks the health of the execution (Step (4)); the *progressive optimizer* re-optimizes the plan if the cardinality estimates turn out to be inaccurate (Step (5)); and the *cost learner* helps users in building the cost model offline. In the following, we explain each of these components using the pseudocode in Algorithm 1, which shows the entire data processing pipeline.

4.1 Optimizer

The cross-platform optimizer (Line 1 in Algorithm 1) is at the heart of RHEEM. It is responsible for selecting the most efficient platform for executing each single operator

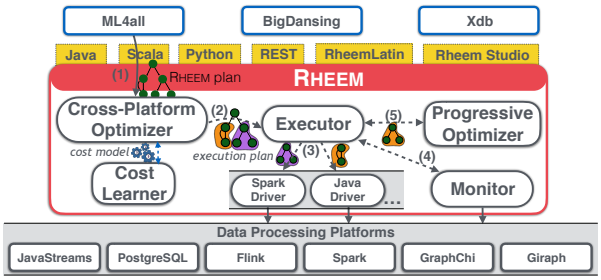

Figure 5: Rheem's ecosystem and architecture.

Algorithm 1: Cross-platform data processing

Input: RHEEM plan *rheemPlan*

```
1 exPlan ← Optimize(rheemPlan)
2 monitor ← StartMonitor(exPlan)
3 finished ← ExecuteUntilCheckpoint(exPlan, monitor)
4 while ¬finished do
5     updated ← UpdateEstimates(exPlan, monitor)
6     if updated then exPlan ← ReOptimize(exPlan)
7     finished ← ResumeExecution(exPlan, monitor)
```

in a RHEEM plan. That is, it does not perform any logical or physical optimization (such as operator reordering, or choosing the operator implementation by considering interesting orders or partitioning). This kind of optimization takes place at the application or platform level. A discussion about these levels of optimization can be found in [17].

Although a rule-based optimizer could determine how to split and execute a plan, e.g., based on its processing patterns [34, 62], such an approach is neither practical nor effective. First, by setting rules, one may make only very simplistic decisions based on the different cardinality and complexity of each operator. 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 cumbersome as the number of rules grows rapidly [62].

We thus pursue a more flexible cost-based approach: *we split a given* RHEEM *plan into subplans and determine the best platform for each subplan so that the total plan cost is minimized.* Below, we give the four main phases of the optimizer, namely *plan inflation*, *cost estimates annotation*, *data movement planning*, and *plan enumeration*. Technical details about these can be found in [43].

Plan Inflation. At first, the optimizer passes the RHEEM plan through an *inflation phase*. It inflates each RHEEM operator of the input plan by applying a set of operator mappings as described in Section 3. Recall that these mappings determine how each of the platform-agnostic RHEEM operators can be implemented on various platforms with execution operators. Note that the inflation process does not simply replace a RHEEM operator with an execution operator via these mappings but it keeps both the RHEEM operator and *all* mapped execution operators. For instance, the gray box in Figure 6 shows the inflated Reduce operator from the SGD plan of Figure 3(a) after applying the mappings of Figure 4. This approach allows the optimizer to later enumerate all possible execution plans without depending on the order in which the mappings are applied. In fact, an inflated plan is

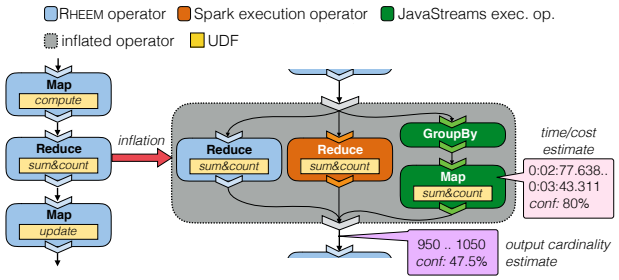

Figure 6: Inflated operator.

a highly compact representation of all execution plans.

Cardinality and Cost Estimates. Next, the optimizer needs to annotate the inflated plan with the cost of each execution operator. RHEEM uses a modular and fully UDF-based cost model and represents cost estimates as intervals with a confidence value (see pink box in Figure 6). Having interval-based estimates allows it to perform on-the-fly re-optimization as we will see in Section 4.4. The cost (e.g., wallclock time or monetary cost) of an execution operator depends on (i) its resource usage (CPU, memory, disk, and network) and (ii) the unit costs of each resource (e.g., how much one CPU cycle costs). The unit costs depend on hardware characteristics (such as number of nodes and CPU cores) which are encoded in a configuration file for each platform. The resource usage of each execution operator o for metric m is estimated by a dedicated cost function r_o^m, which depends on its input cardinality c_{in}. Given that the output cardinality of one operator is equal to the input cardinality of its subsequent operators, RHEEM uses the following approach for estimating the output cardinality of each operator (purple box in Figure 6). It first computes the output cardinalities of the source operators via sampling. It then traverses the inflated plan in a bottom-up fashion to estimate the output cardinality of all subsequent operators in the inflated plan. For this, each RHEEM operator is associated with a cardinality estimator function, which considers its properties (e.g., selectivity and number of iterations) and input cardinalities. Note that applications (and developers) can input the selectivity of the operators in their RHEEM plans via a UDF. There are ways [37, 54] that can be used to help developers to provide the selectivity for their UDFs, but this is part of our future work. Finally, the parameters of all cost functions r_o^m can be learned offline using RHEEM's cost learner module (see Section 4.5 for details). Note that as a result of having a modular and fully UDF-based cost model, one can easily define her own objective criteria for optimizing RHEEM plans. Additionally, the separation of the cost functions from the cost model parameters allows the optimizer to be portable across different deployments.

Data Movement Planning. Given an inflated plan with its estimates, the optimizer also needs to estimate how to best move data quanta among execution operators of different platforms and what the cost of such data movement is. It is crucial to consider data movement at optimization time because of several reasons: (i) its cost can be very high; (ii) its cost must be minimized to be able to scale to several platforms; and (iii) it typically involves operations that go beyond a simple byte copy (caching, data transformations, 1-to-n communication, among others). Existing systems simply fall short of addressing these real-world data movement problems. As already mentioned, we take a graph-based approach and represent the space of data movement paths across platforms as a channel conversion graph. This allows us to model the problem of finding the most efficient communication path among execution operators as a graph problem, which we proved in [43] that is NP-hard and solved it with a novel algorithm that relies on kernelization. Our data movement approach can discover all ways to connect execution operators of different platforms via a sequence of communication channels, if one exists. After the best data movement strategy is found, its cost is attached to the inflated plan.

Plan Enumeration. At last, the optimizer determines the optimal way of executing an input RHEEM plan based on the cost estimates of its inflated plan. Doing so is very challenging because of the exponential size of the search space. A plan with n operators, each having k execution operators, leads to k^n possible plans. This number quickly becomes intractable for a growing n. For instance, a cross-community PageRank plan which consists of $n = 27$ RHEEM operators, each with $k = 5$, yields $2,149,056,512$ possible execution plans. Furthermore, when enumerating all possible execution plans, the optimizer must consider the previously computed data movement costs as well as the start-up costs of data processing platforms. Thus, instead of taking a simple greedy approach that neglects data movement and platform start-up costs, we follow a principled approach: we use an enumeration algebra and propose a lossless pruning technique. Our enumeration algebra has two manipulation operators, namely *Join*, for building a complete execution plan by connecting smaller subplans, and *Prune* (σ), for scraping inferior subplans from an enumeration according to a pruning technique. Our pruning technique is guaranteed to not prune a subplan that is part of the optimal execution plan by keeping only the cheapest subplan from a set of execution subplans having the same initial and ending execution operators. As a result, the optimizer can output the optimal execution plan without an exhaustive enumeration.

4.2 Executor

The executor receives an execution plan from the optimizer to run it on the selected data processing platforms (Lines 3 and 7 in Algorithm 1). For example, the optimizer selected the Spark and JavaStreams platforms for our SGD example in Figure 3(a). Overall, the executor follows well-known approaches to parallelize a task over multiple compute nodes, with only few differences in the way it divides an execution plan. In particular, it divides an execution plan into *stages*. A stage is a subplan where (i) all its execution operators are from the same platform; (ii) at the end of its execution, the platforms need to give back the execution control to the executor; and (iii) its terminal operators materialize their output data quanta in a data structure, instead of being pipelined into the next operator.

In our SGD example of Figure 3(b), the executor divides the execution plan into six stages as illustrated in Figure 7. Note that Stage3 contains only the RepeatLoop operator as the executor must have the execution control to evaluate the loop condition. This is why the executor also separates Stage1 from Stage5. Then, it dispatches the stages to the relevant platform drivers, which in turn submit the stages as a job to the underlying platforms. Stages are connected by data flow dependencies so that stages with no dependencies

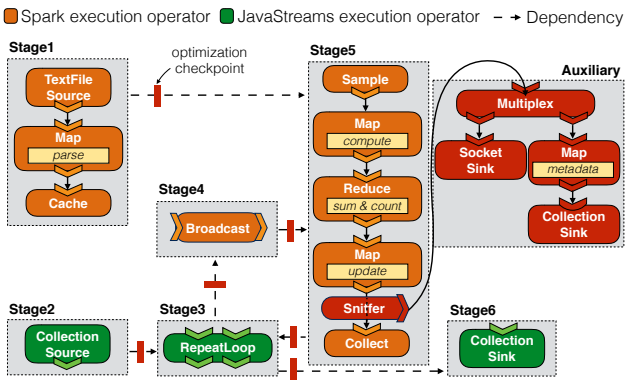

Figure 7: Stage dependencies for SGD.

(e.g., Stage1 and Stage2) are dispatched first in parallel and any other stage is dispatched once its input dependencies are satisfied (e.g., Stage3 after Stage2). Lastly, as data exploration is a key piece in the field of data science, the executor *optionally* allows applications to run in an *exploratory mode* where they can pause and resume the execution of a task at any point. This is done by injecting *sniffers* into execution plans and attaching *auxiliary* execution plans (see Figure 7).

4.3 Monitor

Recall that the cross-platform optimizer operates in a setting that is characterized by high uncertainty. For instance, the semantics of UDFs and data distributions are usually unknown because of the little control over the underlying platforms. This uncertainty can cause poor cardinality and cost estimates and hence can negatively impact the effectiveness of the optimizer [45]. To compensate this uncertainty, RHEEM registers the execution of a plan with the monitor (Line 2 in Algorithm 1). The monitor collects light-weight execution statistics for the given plan, such as data cardinalities and operator execution times. It is also aware of lazy execution strategies used by the underlying platforms and assigns measured execution time correctly to operators. RHEEM uses these statistics to improve its cost model and re-optimize ongoing execution plans in case of poor cardinality estimates. Additionally, the monitor is responsible for checking the health of the execution. For instance, if it finds a large mismatch between the real output cardinalities and the estimated ones, it pauses the execution plan and sends it to the progressive optimizer.

4.4 Progressive Optimizer

To mitigate the effects of bad cardinality estimates, RHEEM employs a *progressive query optimization* approach. The key principle is to re-optimize the plan whenever the cardinalities observed by the monitor greatly mismatch the estimated ones [48]. Applying progressive query optimization in our setting comes with two main challenges. First, we have only limited control over the underlying platforms, which makes plan instrumentation and halting executions difficult. Second, re-optimizing an ongoing execution plan must efficiently consider the results already produced.

We tackle these challenges by using *optimization checkpoints*. An optimization checkpoint tells the executor to pause the plan execution in order to consider a re-optimization of the plan beyond the checkpoint. The progressive optimizer inserts optimization checkpoints into execution plans wherever (i) cardinality estimates are uncertain (having a wide interval or low confidence) or (ii) the data is at rest (e.g., a Java collection or a file). For instance, the optimizer inserts an optimization checkpoint right after Stage1 as the data is at rest because of the Cache operator (see Figure 7). When the executor cannot dispatch a new stage anymore without crossing an optimization checkpoint, it pauses the execution and gives the control to the progressive optimizer. The latter gets the actual cardinalities observed so far by the monitor and re-computes all cardinalities from the current optimization checkpoint (Line 5 in Algorithm 1). In case of a mismatch, it re-optimizes the remaining of the plan (from the current optimization checkpoint) using the new cardinalities (Line 6). It then gives the new execution plan to the executor, which resumes the execution from the current optimization checkpoint (Line 7). RHEEM can switch between execution and progressive optimization any number of times at a negligible cost.

4.5 Cost Model Learner

Profiling operators in isolation might be unrealistic whenever platforms optimize execution across multiple operators, e.g., by pipelining. Indeed, we found cost functions derived from isolated benchmarking to be insufficiently accurate. We thus take a different approach.

Learning the Cost Model. Recall that each execution operator o is associated with a number of resource usage functions (r_o^m, where m is CPU, memory, disk, or network). For instance, the cost function to estimate the CPU cycles required by the JavaFilter operator is $r^{CPU}_{JavaFilter} := c_{in} \times (\alpha + \beta) + \delta$, where parameters α and β denote the number of required CPU cycles for each input data quantum in the operator itself and in its UDF, and parameter δ describes some fixed overhead for the operator start-up and scheduling. We then multiply each of these resource usage functions r_o^m with the time required per unit (e.g., msec/CPU cycle) to get the time estimate t_o^m. The total cost estimate for operator o is defined as: $f_o = t_o^{CPU} + t_o^{mem} + t_o^{disk} + t_o^{net}$. However, obtaining the parameters for each resource, such as the α, β, δ values for CPU, is not trivial. We, thus, use execution logs to *learn* these parameters in an offline fashion and model the cost of individual execution operators as a *regression problem*. Note that the execution logs contain the runtimes of execution stages (i.e., pipelines of operators as defined in Section 4.2) and not of individual operators. Let $(\{(o_1, C_1), (o_2, C_2), \dots (o_n, C_n)\}, t)$ be an execution stage, with o_i, $0 < i \leq n$, where o_i are execution operators, C_i are input and output true cardinalities, and t is the measured execution time for the entire stage. Furthermore, let $f_i(\mathbf{x}, C_i)$ be the total cost function for execution operator o_i with $\mathbf{x}$ being a vector with the parameters of all resource usage functions (e.g., CPU cycles, disk I/O per data quantum). We are interested in finding $\mathbf{x}_{\min} = \arg\min_{\mathbf{x}} \; loss\left(t, \; \sum_{i=1}^{n} f_i(\mathbf{x}, C_i)\right)$. 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 bounds of the cost 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 stages: we minimize the weighted arithmetic mean of the losses of multiple execu-

tion stages. In particular, we use as stage weights the sum of the relative frequencies of the stages' operators among all stages, so as to deal with skewed workloads that contain certain operators more often than others. Finally, we apply a genetic algorithm [50] to find $\mathbf{x}_{\min}$. In contrast to other optimization algorithms, genetic algorithms impose only few restrictions on the loss function to be minimized. Hence, our cost learner can deal with arbitrary cost functions. Applying this technique allows us to calibrate the cost functions with only little additional effort.

Logs Generation. Clearly, the more execution logs are available, the better RHEEM can tune the cost model. Thus, RHEEM comes with a log generator. It first creates a set of RHEEM plans by composing all possible combinations of RHEEM operators forming a particular topology. We found that most data analytic tasks in practice follow three different topologies: *pipeline* (e.g., batch tasks), *iterative* (e.g., ML tasks), and *merge* (e.g., SPJA tasks). It then generates all possible executions plans for the previously created set of RHEEM plans. Next, it creates different configurations for each execution plan, i.e., it varies the UDF complexity, output cardinalities, input dataset sizes, and data types. Once it has generated all possible plans with different configurations, it executes them and logs their runtime.

5. BUILDING A RHEEM APPLICATION

RHEEM provides a set of native APIs for developers to build their applications. These include Java, Scala, Python, and REST. Examples of using these APIs can be found in the RHEEM repository[5]. The code developers have to write is fully agnostic of the underlying platforms. Still, in case the user wants to force RHEEM to execute a given operator on a specific platform, she can invoke the withTargetPlatform method. Similarly, she can force the system to use a specific execution operator via the customOperator method, which further enables users to employ custom operators without having to extend the API.

Although the native APIs are quite popular among developers, many users are not proficient using these APIs. Thus, RHEEM also provides two APIs that target non-expert users: a data-flow language (*RheemLatin*) and a visual IDE (RHEEM *Studio*). The salient feature of all these APIs is that they are all platform-agnostic. It is RHEEM that figures out on which platform to execute each of the operators.

RheemLatin. RHEEM provides a data-flow language (RheemLatin) for users to specify their tasks [47]. Our goal is to provide ease-of-use to users without compromising expressiveness. RheemLatin follows a procedural programming style to naturally fit the pipeline paradigm of RHEEM. It draws its inspiration from PigLatin [51] and hence it has PigLatin's grammar and supports most PigLatin's keywords. In fact, one could see it as an extension of PigLatin for cross-platform settings. For example, users can specify the platform for any part of their queries. More importantly, it provides a set of configuration files whereby users can add new keywords to the language together with their mappings to RHEEM operators. As a result, users can easily adapt RheemLatin for their applications. Listing 1 illustrates how one can express our SGD example with RheemLatin.

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

```
1  import '/sgd/udfs.class' AS taggedPointCounter;
2  lines = load 'hdfs://myData.csv';
3  points = map lines -> {taggedPointCounter.parsePoints(lines)};
4  weights = load taggedPointCounter.createWeights();
5  final_weights = repeat 50 {
6      sample_points = sample points -> {taggedPointCounter.getSample()}
           with broadcast weights;
7      gradient = map sample_points ->
           {taggedPointCounter.computeGradient()};
8      gradient_sum_count = reduce gradient -> {gradient.sumcount()};
9      weights = map gradient_sum -> {gradient_sum_count.average()} with
           platform 'JavaStreams';}
10 store final_weights 'hdfs://output/sgd';
```

Listing 1: SGD task in RheemLatin.

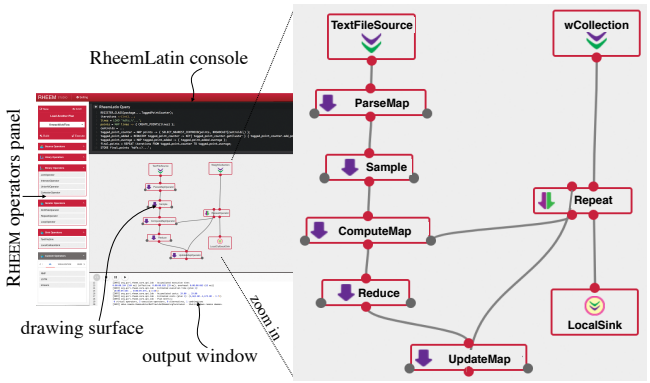

Figure 8: SGD task in the Rheem Studio.

Rheem Studio. Although the native APIs and RheemLatin cover a large number of users, some might still be unfamiliar with programming and data-flow languages. To this end, RHEEM provides a visual IDE (RHEEM Studio) where users can compose their data analytic tasks in a *drag and drop* fashion [47]. Figure 8 shows the RHEEM Studio's GUI. The GUI is composed of four parts: a panel containing all RHEEM operators, the drawing surface, a console for writing RheemLatin queries, and the output terminal. The right-side of Figure 8 shows how operators are connected for an SGD plan. The studio provides default implementations for any of the RHEEM operators, which enables users to run common data analytic tasks without writing code. Yet, expert users can provide a UDF by double-clicking on any operator.

6. DON'T UNDERESTIMATE THE FORCE

We carried out several additional experiments to the ones we presented earlier in Section 2. Our goal in these additional experiments is to further show the benefits of the cross-platform data processing approach over the single-platform data processing approach. Due to space limitations, we focus on the platform-independence and opportunistic cross-platform cases. We also evaluate our progressive optimization and data exploration techniques as well as compare RHEEM with Musketeer [34].

6.1 Experiments Setup

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. While we use RHEEM for the cross-platform data processing approach, we use the following commonly used data

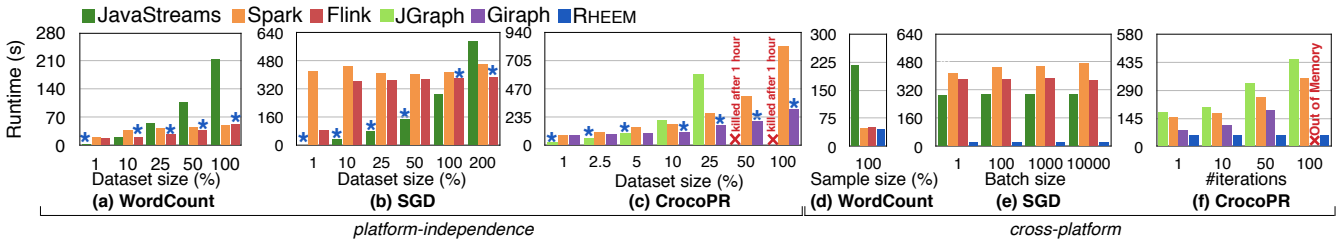

Figure 9: (a)-(c) Platform independence: Rheem always selects the best data processing platform; (d)-(f) Opportunistic cross-platform: Rheem improves performance by combining multiple platforms.

Table 1: Tasks and datasets.

Task	Description	Rheem operators	Dataset
`WordCount`	count distinct words	4	*Wikipedia abstracts* (3GB)
`SGD`	stochastic gradient descent	9	*HIGGS* (7.4GB)
`CrocoPR`	cross-community pagerank	27	*DBpedia pagelinks* (24GB)

processing platforms for the single-platform data processing approach: Java's Stream library (JavaStreams), PostgreSQL 9.6.2 (Postgres), Spark 1.6.0 (Spark), Flink 1.3.2 (Flink), GraphX 1.6.0 (GraphX), Giraph 1.2.0 (Giraph), a Java graph library (JGraph), and HDFS 2.6.0 to store files. We used all these platforms with their default settings and configured the maximum RAM of each platform to 20 GB. We disabled the RHEEM stage parallelization feature to have only one single platform running at any time. We obtained all the cost functions required by our optimizer as described in Section 4.5. We considered common data analytic tasks from three important areas, namely text mining, machine learning, and graph mining. Details on the datasets (stored on HDFS) and tasks are shown in Table 1. We loop `SGD` for 1,000 and `CrocoPR` 10 times.

6.2 Experiments Results

Platform Independence. We start our experimental study by evaluating how well RHEEM selects a platform to execute a task. For this experiment, we forced RHEEM to use a single platform when executing a task and checked if it chooses the one with the best runtime.

Figures 9(a)-(c) show the results for increasing dataset sizes[6]. The blue stars indicate the platform chosen by RHEEM. The first observation is that there is no single platform that outperforms all other platforms for all cases. In fact, the differences in the runtime values are significant. For example, Flink can be more than 4× faster than Spark and Spark can be twice faster than Flink for the different tasks we considered in our evaluation. The results show that supporting platform independence indeed prevents tasks from falling into such non-obvious worst cases. In detail, our system prevents: (i) `WordCount` on JavaStreams for 100% of its input dataset (i.e., 3GB), where JavaStreams suffers from a single threaded data access; (ii) `SGD` on Spark and Flink for 25% of its input dataset, where the overhead of both platforms still dominates execution time; and (iii) `CrocoPR` on JGraph for more than 10% of its input dataset as it simply cannot efficiently process large datasets. More importantly, RHEEM always selects the best platform in all cases, even if the execution time is quite similar on different platforms. These results clearly show the importance of providing platform independence.

> RHEEM selects the most efficient platform for all tasks we considered.

Opportunistic Cross-Platform. We now evaluate if our system is able to (i) use multiple platforms to reduce execution times and (ii) spot "hidden" opportunities for the use of multiple platforms. Thus, we re-enable RHEEM to use any platform combination for performing one single task. We consider the same tasks as in the previous set of experiments but with two differences. First, we now study `SGD` on its entire dataset only and for different batch sizes. Second, we run `CrocoPR` on 10% of its input dataset and for a varying number of iterations. Additionally, to further stress the importance of finding hidden cross-platform execution opportunities, we ran a subquery (a `Join` task) of the TPC-H Q5. This additional task joins the relations `SUPPLIER` and `CUSTOMER` (which are stored on Postgres) on the attribute `nationkey` and aggregates the results on the same attribute.

We show the results of this experiment in Figures 9(d)-(f) and 10(a). We overall observe that performing cross-platform data processing considerably improves over single-platform data processing. In detail, we observe RHEEM is: up to 20× faster than Spark; up to 17× faster than Flink; up to 12× faster than JavaStreams; up to 8× faster than JGraph; up to 6× faster than Giraph; and more than 3× faster than Postgres. There are several reasons for having this large improvement. For `SGD`, RHEEM mixes Spark with JavaStreams. It uses Spark execution operators when dealing with a large number of data points, i.e., for the Sample operator and all operators outside the loop. In contrast, it uses Java execution operators for computing and updating the gradient of a single data point, which is done inside the loop. When running `CrocoPR`, our system surprisingly uses a combination of Flink and JGraph, even though Giraph is the fastest baseline platform. The reason is that after the preparation phase on Flink, the input dataset for the PageRank operation on JGraph is a couple of megabytes only. For `WordCount`, our system slightly outperforms Spark and Flink by bringing the results to the driver application via JavaStreams rather than directly from Spark, which is the fastest baseline platform for this task. 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()). Figure 10(a) confirms the importance of the cross-platform data processing approach. We compared RHEEM with the execution of `Join`

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

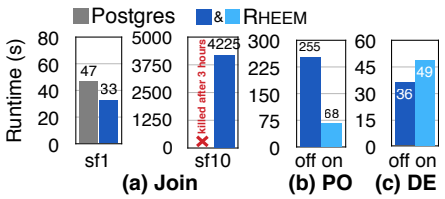

Figure 10: (a) Join, (b) Progressive Optimizer, and (c) Data Exploration

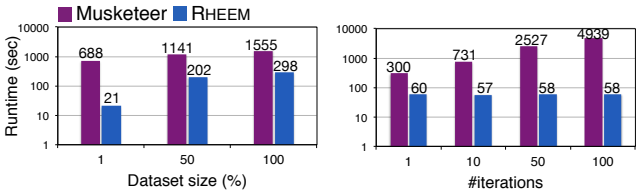

Figure 11: Rheem outperforms Musketeer by more than one order of magnitude.

on Postgres, which is the obvious platform to run this kind of queries. Surprisingly, RHEEM significantly outperforms Postgres, even though the input data is stored on Postgres. It is at least twice faster than Postgres for a scale factor of 10. This is because it executes the projection on Postgres and moves only the projected data into Spark to leverage its parallelism to perform the join and aggregation operations.

> RHEEM is able to identify hidden opportunities to improve performance by using multiple platforms.

Progressive Optimization. We proceed to evaluate the effectiveness of the progressive optimizer (PO) by extending the `Join` task. We added a low-selective selection predicate on the names of the suppliers and customers. Then, to simulate the usual cases where users cannot provide accurate selectivity estimates, we provide a high selectivity *hint* to RHEEM for this filter operator. Figure 10(b) shows the results for this experiment. We clearly observe the benefits of our progressive optimizer. In detail, RHEEM first generates an execution plan using Spark and JavaStreams. It uses JavaStreams for all the operators after the Filter because it sees that Filter has a very high selectivity. However, RHEEM figures out that Filter is in fact low selective. Thus, it runs the re-optimization process and changes on-the-fly all JavaStreams operators to Spark operators. This allows it to speed up performance by almost 4 times.

> RHEEM further improves performance notably by re-optimizing plans on-the-fly at a negligible cost.

Data Exploration. We also evaluate the cost that RHEEM incurs in exploratory mode, where preliminary results are returned to users in less than 2 seconds and a task can be paused at and resumed from any part of its plan. We modify the `WordCount` task to compute how many words have less than 10 characters and equal or more than 10 characters. We insert a sniffer right before the reduce operator of this task to keep track of both sets of words. Figure 10(c) shows the results of this experiment. We observe that RHEEM not only enables the underlying platforms to support data exploration, but it does so with an overhead of $\sim$36% only.

> RHEEM enables underlying platforms to run in exploratory mode at a low cost.

Comparison with Musketeer. Lastly, we compared RHEEM with its closest competitor, Musketeer [34]. For this experiment, we considered `CrocoPR` because the authors reported this task to be a case where Musketeer chooses multiple platforms. Figure 11 shows the results in log scale when varying the dataset sizes for 10 iterations and the number of iterations for 10% of the dataset. Overall, we observe the superiority of RHEEM over Musketeer, especially as the number of iterations increases: RHEEM is up to 85 times faster than Musketeer. Note that, in contrast to Musketeer, RHEEM keeps its runtime constant as the number of iterations increases. This is because: (i) Musketeer, among other things, checks dependencies, compiles and package the code, and writes the output to HDFS at each iteration (or stage), which comes with a high overhead; (ii) RHEEM executes the page rank part of the task (i.e., after the data preparation) on JavaStreams, which allows it to perform each iteration with almost zero overhead.

7. DISCUSSION

We first discuss our system's limitations and then summarize the lessons learned while building and using RHEEM.

7.1 Limitations

RHEEM does not support any stream processing platform. While users can easily supply new batch processing platforms, stream processing requires to extend RHEEM's core. We plan to do so by following the lambda architecture paradigm [49]. In addition, RHEEM currently relies on the fault-tolerance of the underlying platforms and is, thus, susceptible to failures while moving data across platforms. We plan to incorporate some basic fault-tolerance mechanism at the cross-platform level. Other issues that remain to be addressed include: adding methods that speed up inter-platform communications, such as the one proposed in [35], integrating RHEEM with resource managers to incorporate changes in the availability of computing resources, and supporting simultaneous execution of RHEEM jobs.

7.2 Lessons Learned

We now give the main takeaway messages from our journey in building RHEEM and several RHEEM applications.

Extensibility As A First Class Citizen. A cross-platform environment is quite dynamic as new platforms are constantly appearing and applications often emerge with new requirements. Thus, a cross-platform system has to be easily extensible and flexible enough to adapt to these constant changes with little effort. Having this in mind, we designed RHEEM in a way that extending it with new RHEEM or execution operators does not require developers to worry about any of RHEEM's internal implementation details. Depending on the complexity of the execution operator, it can take from one hour (for simple operators already supported by the underlying platform, such as Map) to one day (for more complex operators) to add a new operator with its mappings. For example, we had to design a new algorithm for inequality join [42] for BigDansing [41] and provide its implementation as a new join operator. Similarly, we had to implement new sample operators that are IO-efficient for

ML4all [40]. We experienced the same when developing Data Civilizer: we could easily add a new operator to wrap any dataset as a relational table. In all cases, it took the application developers one day for adding a couple of Java classes to the codebase. Furthermore, the addition of a new platform always takes constant time, i.e., regardless of how many platforms are already supported. For instance, supporting Giraph, which has a graph-based data model, took two additional days for providing its communication channels and conversion operators (after adding all its execution operators and mappings). This experience constitutes strong evidence of the benefit of an extensible architecture, where users can plug in operators with little effort.

Learning Cost Models. We also realized that using a traditional approach of isolated profiling to build a cost model is simply not effective for several reasons. First, one has little control over the underlying platforms. Second, cross-platform parallelism is difficult to take into account. Third, isolated profiling does not reflect at all how different platforms interact with each other, including the data movement costs. This led us to adopt a *learning* approach to build the Rheem's cost model. We used machine learning to learn the cost model from actual execution logs. This not only removed the complexity of calibrating the cost model but also made Rheem easier to deploy on any cross-platform setting.

Rapid Prototyping and Portability. Several works have shown that rapid prototyping and portability are crucial factors when building big data systems [24, 33, 59]. We confirmed these observations when developing our Rheem applications. We had to iterate several approaches to address the encountered problems and take good design decisions. Rheem enabled us to do so in a period of time that was not possible before. For example, when building the data integration sub-system of Data Civilizer, we needed only one week to implement and play with several different approaches to detect similar data elements existing on different data storages. This was possible because Rheem allowed us to focus on the logic of the different approaches rather than on the tedious platform integration chores. Only building the program to integrate and coordinate the underlying platforms would not be possible to do in few weeks. Rheem also provided us with application portability, which allowed us to try out different underlying platforms.

Flexible High-Level Languages. The emergence of new platforms and applications and their ensuing requirements very often influences the way high-level languages have to be designed. A high-level language must be flexible enough to adapt to the needs of applications. This was one of the driving principles when designing RheemLatin and the Rheem Studio [47]. In fact, we exploited such a flexibility to adapt both high-level languages to two specific applications: ML4all and Rheem Studio.

8. RELATED WORK

The research and industry communities have proposed a myriad of different data processing platforms [5, 8, 11, 18, 26, 63]. In contrast, we do not provide a data processing platform but a novel system on top of them.

Cross-platform data processing has been in the spotlight only very recently. Some works focus only on integrating different data processing platforms with the goal of alleviating users from their intricacies [1, 2, 10, 12, 30]. However, they still require expertise from users to decide when to use a specific data processing platform. For example, BigDAWG [30] requires users to specify where to run tasks via its Scope and Cast commands, which already require expertise from users. Only few works share a similar goal with us [29,34,46,58,62]. However, they substantially differ from Rheem. Two main differences are that they consider neither data movement costs nor progressive task optimization techniques, although both aspects are crucial in cross-platform settings. Additionally, each of these works differs from Rheem in various ways. Musketeer maps task patterns to specific underlying platforms, hence it is not clear how one can efficiently map a task when having similar platforms (e.g., Spark vs. Flink or Postgres vs. MySQL). Similarly, in Myria [62], it is hard to allocate tasks when having similar platforms because it comes with a rule-based optimizer, which additionally makes it hard to maintain. IReS [29] supports only 1-to-1 mappings between abstract tasks and their implementations, which limits expressiveness and optimization opportunities. QoX focuses only on ETL workloads [58]. DBMS+ [46] is limited by the expressiveness of its declarative language and hence it is neither adaptive nor extensible. Other complementary works focus on improving data movement across different platforms [35] or libraries by using a common intermediate representation and executing the scripts in LLVM [52]. However, none of them address the cross-platform optimization problem. Tensorflow [15] follows a similar idea, but for cross-device execution of machine learning tasks and thus it is orthogonal to Rheem. In fact, Rheem could use TensorFlow as an underlying platform.

The research community has also studied the problem of federating relational databases [57]. Garlic [22], TSIMMIS [23], and InterBase [21] are just three examples. However, all these works significantly differ from Rheem in that they consider a single data model and simply push query processing to where the data is. Other works integrate Hadoop with an RDBMS [28,44], however, one cannot easily extend them to deal with more diverse tasks and platforms.

To the best of our knowledge, Rheem is the first system to support all four cross-platform use cases (see Section 2).

9. MAY THE BIG DATA BE WITH YOU!

Given today's data analytic ecosystem, supporting cross-platform data processing has become rather crucial in organizations. We have identified four different situations in which an application requires or benefits from cross-platform data processing. Driven by these cases, we built Rheem, a cross-platform system that decouples applications from data processing platforms to achieve efficient task execution over multiple platforms. Rheem follows a cost-based optimization approach for splitting an input task into subtasks and assigning each subtask to a specific platform, such that the cost (e.g., runtime or monetary cost) is minimized. Our experience while building Rheem raised several interesting questions that need to be addressed in the future, namely: *How can we (i) reduce the inter-platform data movement costs? (ii) address the cardinality and cost estimation problem? (iii) efficiently support fault-tolerance across platforms? (iv) add new platforms automatically? and (v) improve data exploration in cross-platform settings?*

10. REFERENCES

[1] Apache Beam. https://beam.apache.org.

[2] Apache Drill. https://drill.apache.org.

[3] Apache Flink. https://flink.apache.org.

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

[5] Apache HBase. http://hbase.apache.org/.

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

[7] Apache Mahout. http://mahout.apache.org.

[8] Apache Spark: Lightning-Fast Cluster Computing. http://spark.incubator.apache.org/.

[9] Fortune magazine. http://fortune.com/2014/06/19/big-data-airline-industry/.

[10] Luigi Project. https://github.com/spotify/luigi.

[11] PostgreSQL. http://www.postgresql.org/.

[12] PrestoDB Project. https://prestodb.io.

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

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

[15] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.

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

[17] D. Agrawal et al. Road to Freedom in Big Data Analytics. In *EDBT*, pages 479–484, 2016.

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

[19] A. Baaziz and L. Quoniam. How to use big data technologies to optimize operations in upstream petroleum industry. In *21st World Petroleum Congress*, 2014.

[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. *PVLDB*, 9(13):1425–1436, 2016.

[21] O. A. Bukhres et al. InterBase: An Execution Environment for Heterogeneous Software Systems. *IEEE Computer*, 26(8):57–69, 1993.

[22] M. J. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM*, pages 124–131, 1995.

[23] S. S. Chawathe et al. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, pages 7–18, 1994.

[24] H. Chen, R. Kazman, and S. Haziyev. Strategic prototyping for developing big data systems. *IEEE Software*, 33(2):36–43, 2016.

[25] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[27] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The Data Civilizer System. In *CIDR*, 2017.

[28] 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 *SIGMOD*, pages 1255–1266, 2013.

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

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

[31] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2):6:1–6:48, 2008.

[32] R. C. Fernandez, D. Deng, E. Mansour, A. A. Qahtan, W. Tao, Z. Abedjan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. A Demo of the Data Civilizer System. In *SIGMOD*, pages 1639–1642, 2017.

[33] M. Franklin. Making sense of big data with the berkeley data analytics stack. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 1–2, 2015.

[34] I. Gog et al. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, 2015.

[35] B. Haynes, A. Cheung, and M. Balazinska. PipeGen: Data Pipe Generator for Hybrid Analytics. In *SoCC*, pages 470–483, 2016.

[36] A. Hems, A. Soofi, and E. Perez. How innovative oil and gas companies are using big data to outmaneuver the competition. Microsoft White Paper, http://goo.gl/2Bn0xq, 2014.

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

[38] IBM. Data-driven healthcare organizations use big data analytics for big gains. White paper, http://goo.gl/AFIHpk.

[39] Z. Kaoudi and J.-A. Quiané-Ruiz. Cross-Platform Data Processing: Use Cases and Challenges. In *ICDE (tutorial)*, 2018.

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

[41] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A System for Big Data Cleansing. In *SIGMOD*, pages 1215–1230, 2015.

[42] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and P. Kalnis. Lightning Fast and Space Efficient Inequality Joins. *PVLDB*, 8(13):2074–2085, 2015.

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

https://arxiv.org/abs/1805.03533, 2018.

[44] 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 *SIGMOD*, pages 1591–1602, 2014.

[45] V. Leis et al. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[46] H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *CIDR*, 2013.

[47] J. Lucas, Y. Idris, B. Contreras-Rojas, J.-A. Quiané-Ruiz, and S. Chawla. Cross-Platform Data Analytics Made Easy. In *ICDE*, 2018.

[48] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[49] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning, 2015.

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

[51] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, 2008.

[52] S. Palkar, J. J. Thomas, A. Shanbhag, M. Schwarzkopt, S. P. Amarasinghe, and M. Zaharia. Weld: A Common Runtime for High Performance Data Analysis. In *CIDR*, 2017.

[53] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD*, pages 165–178, 2009.

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

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

[56] S. Shankar, A. Choi, and J.-P. Dijcks. Integrating Hadoop Data with Oracle Parallel Processing. Oracle White Paper, http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-integrating-hadoop-data-with-or-130063.pdf, 2010.

[57] 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.

[58] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. In *SIGMOD*, pages 829–840, 2012.

[59] C. Statchuk, N. Madhavji, A. Miranskyy, and F. Dehne. Taming a tiger: Software engineering in the era of big data & continuous development. In *CASCON*, 2015.

[60] M. Stonebraker. The Case for Polystores. http://wp.sigmod.org/?p=1629, 2015.

[61] D. Tsoumakos and C. Mantas. The Case for Multi-Engine Data Analytics. In *Euro-Par*, pages 406–415, 2013.

[62] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*, 2017.

[63] F. Yang, J. Li, and J. Cheng. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *PVLDB*, 9(5):420–431, 2016.