Visionary Paper

Road to Freedom in Big Data Analytics

Divy Agrawal* Sanjay Chawla Ahmed Elmagarmid Zoi Kaoudi Mourad Ouzzani
Paolo Papotti Jorge-Arnulfo Quiané-Ruiz Nan Tang Mohammed J. Zaki*

Data Analytics Center, Qatar Computing Research Institute, HBKU

ABSTRACT

The world is fast moving towards a data-driven society where data is the most valuable asset. Organizations need to perform very diverse analytic tasks using various data processing platforms. In doing so, they face many challenges; chiefly, platform dependence, poor interoperability, and poor performance when using multiple platforms. We present RHEEM, our vision for big data analytics over diverse data processing platforms. RHEEM provides a three-layer *data processing* and *storage* abstraction to achieve both platform independence and interoperability across multiple platforms. In this paper, we discuss our vision as well as present multiple research challenges that we need to address to achieve it. As a case in point, we present a data cleaning application built using some of the ideas of RHEEM. We show how it achieves *platform independence* and the performance benefits of following such an approach.

1. WHY TIED TO ONE SINGLE SYSTEM?

Data analytic tasks may range from very simple to extremely complex pipelines, such as data extraction, transformation, and loading (ETL), online analytical processing (OLAP), graph processing, and machine learning (ML). Following the dictum "one size does not fit all" [23], academia and industry have embarked on an endless race to develop data processing platforms for supporting these different tasks, *e.g.,* DBMSs and MapReduce-like systems. Semantic completeness, high performance, and scalability are key objectives of such platforms. While there have been major achievements in these objectives, users still face two main roadblocks.

The **first roadblock** is that applications are tied to a single processing platform, making the migration of an application to new and more efficient platforms a difficult and costly task. Furthermore, complex analytic tasks usually require the combined use of different processing platforms. As a result, the common practice is to develop several specialized analytic applications on top of different platforms. This requires users to manually combine the results to draw a conclusion. In addition, users may need to re-implement existing applications on top of faster processing platforms when these become available. For example, Spark SQL [3] and MLlib [2] are the Spark counterparts of Hive [24] and Mahout [1].

The **second roadblock** is that datasets are often produced by different sources and hence they natively reside on different storage platforms. As a result, users often perform tedious, time-intensive, and costly data migration and integration tasks for further analysis.

Let us illustrate these roadblocks with an Oil & Gas industry example [13]. A single oil company can produce more than 1.5TB of diverse data per day [6]. Such data may be structured or unstructured and come from heterogeneous sources, such as sensors, GPS devices, and other measuring instruments. For instance, during the exploration phase, data has to be acquired, integrated, and analyzed in order to predict if a reservoir would be profitable. Thousands of downhole sensors in exploratory wells produce real-time seismic data for monitoring resources and environmental conditions. Users integrate these data with the physical properties of the rocks to visualize volume and surface renderings. From these visualizations, geologists and geophysicists formulate hypotheses and verify them with ML methods, such as regression and classification. Training of the models is performed with historical drilling and production data, but oftentimes users have to go over unstructured data, such as notes exchanged by emails or text from drilling reports filed in a cabinet. Thus, an application supporting such a complex analytic pipeline has to access several sources for historical data (relational, but also text and semi-structured), remove the noise from the streaming data coming from the sensors, and run both traditional (such as SQL) and statistical analytics (such as ML algorithms) over different processing platforms.

Similar examples can be drawn from many other domains such as healthcare: *e.g.,* IBM reported that North York hospital needs to process 50 diverse datasets, which are on a dozen different internal systems [15]. These emerging applications clearly show the need for complex analytics coupled with a diversity of processing platforms, which raises two major research challenges.

Data Processing Challenge. Users are faced with various choices on where to *process* their data, each choice with possibly orders of magnitude differences in terms of performance. However, users have to be intimate with the intricacies of the processing platform to achieve high efficiency and scalability. Moreover, once a decision is taken, users may end up being tied up to a particular platform. As a result, migrating the data analytics stack to a more efficient processing platform often becomes a nightmare. Thus, there is a need to build a system that offers *data processing platform independence*. Furthermore, complex analytic applications require executing tasks over different processing platforms to achieve high performance. For example, one may aggregate large datasets with traditional queries on top of a relational database such as PostgreSQL, but ML tasks might be much faster if executed on Spark [28]. How-

*Work done while at QCRI.

ever, this requires a considerable amount of manual work in selecting the best processing platforms, optimizing tasks for the chosen platforms, and coordinating task execution. Thus, this also calls for *multi-platform task execution*.

Data Storage Challenge. Data processing platforms are typically tightly coupled with a specific *storage* solution. Moving data from a certain storage (*e.g.,* a relational DB) to a more suitable processing platform for the actual task (*e.g.,* Spark on HDFS) requires shuffling data between different systems. Such shuffling may end up dominating the execution time. Moreover, different departments in the same organization may go for different storage engines due to legacy as well as performance reasons. Dealing with such heterogeneity calls for *data storage independence*.

To tackle these two challenges, we envision a system, called RHEEM[1], that provides both platform independence and interoperability (Section 2). In the following, we first discuss our vision for the data processing abstraction (Section 3), which is fully based on user-defined functions (UDFs) to provide adaptability as well as extensibility. This processing abstraction allows both users to focus only on the logic of their data analytic tasks and applications to be independent from the data processing platforms. We then discuss how to divide a complex analytic task into smaller subtasks to exploit the availability of different processing platforms (Section 4). As a result, RHEEM can run simultaneously a single data analytic task over multiple processing platforms to boost performance. Next, we present our first attempt to build an instance application based on some of the ideas of RHEEM and the resulting benefits (Section 5). We then show how we push down the processing abstraction idea to the storage layer (Section 6). This storage abstraction allows both users to focus on their storage needs and the processing platforms to be independent from the storage engines.

Some initial efforts are also going into the direction of providing data processing platform independence [11, 12, 21] (Section 7). However, our vision goes beyond the data processing. We not only envision a data processing abstraction but also a data storage abstraction, allowing us to consider data movement costs during task optimization. We give a research agenda highlighting the challenges that need to be tackled to build RHEEM in Section 8.

2. OUR VISION

We envision a system that frees applications and users from being tied to a single data processing platform (platform independence) and provides interoperability across different platforms (multi-platform task execution). We discuss these two aspects in the following. We discuss data storage independence in Section 6.

Processing Platform Independence. Whenever a new platform that achieves better performance than existing ones becomes available, one is enticed to move to the new platform. However, such move does not usually come without pain. There is a clear need for a system that frees us from the burden and cost of re-implementing applications from one platform to another. Mahout [1] and MLlib [2] clearly illustrate this need, as all ML algorithms initially implemented in Hadoop had to be re-implemented in Spark. In addition, there are cases where, for the same task but with a different input, one platform is better than another. Thus, the system we envision should not only provide platform independence, but also should be able to select the best available platform to execute a given task in order to deliver better performance.

Multi-Platform Task Execution. We are witnessing the emergence of complex data analytic pipelines in many different domains [4, 6, 13, 15]. These pipelines require first combining multiple processing platforms to perform each task of the process and then integrating the results. For instance, many companies are already adopting a lambda architecture, which combines both batch and stream processing. Our vision goes beyond batch or stream processing to any kind of data analytics paradigm. We envision a system that eases the integration among different processing platforms by automatically dividing a task into subtasks and determining the underlying platform for each subtask.

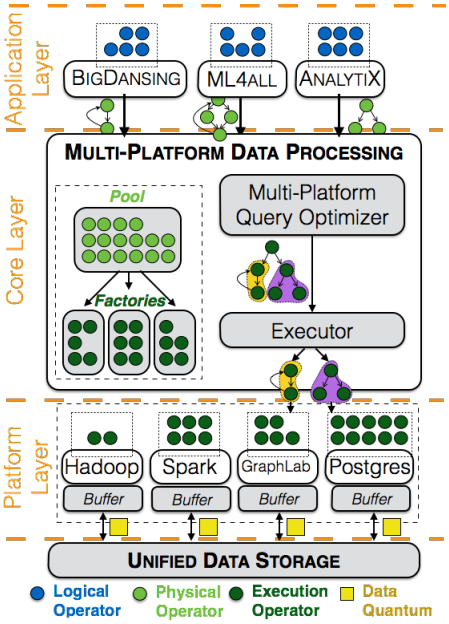

Figure 1: RHEEM data processing abstraction.

RHEEM. The foundation of our vision is a three-layer data processing abstraction that sits between user applications and data processing platforms (*e.g.,* Hadoop or Spark). Figure 1 depicts these three layers: the *application* layer models all application-specific logic; the *core* layer provides the intermediate representation between applications and processing platforms; and the *platform* layer embraces the underlying processing platforms. In contrast to DBMSs, RHEEM decouples physical and execution levels. This separation allows applications to express physical plans in terms of algorithmic needs only, without being tied to a particular processing platform. The communication among these levels is enabled by operators defined as UDFs. These three layers allow RHEEM to provide applications with platform independence. Providing platform independence is the first step towards realizing multi-platform task execution, which is crucial to achieve the best performance at all times. For example, Figure 2 shows the benefits of running the SVM algorithm on different datasets from LIBSVM[2] with only one hundred iterations, as a Spark job and as a plain Java program. We observe that, for small datasets, executing SVM as a plain Java program is up to one order of magnitude faster than executing it on Spark. Indeed, this performance gap gets bigger with the number of iterations. Using Spark pays off for big datasets only. Such results show a great potential for platform independence and ultimately multi-platform execution. RHEEM will be able to receive a complex analytic task, seamlessly divide it into subtasks, schedule each task on the best processing platform, monitor task execution, and aggregate results for users or applications. Achieving our vision requires tackling several challenges that we will discuss throughout

[1]Rheem is a native gazelle species in Qatar.

[2]http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

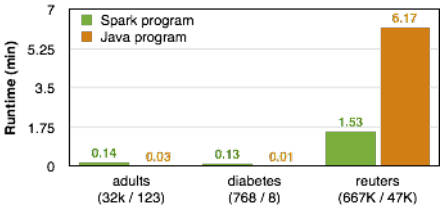

Figure 2: SVM on Spark and Java.

the paper and then summarize in Section 8.

To show the benefits of our RHEEM vision, we have fully developed one data cleaning application based on it [19]. While this initial instance provides only platform independence, its performances are encouraging and already demonstrate the advantages of our vision (see Section 5).

Similar to the data processing abstraction, we envision a three-level data storage abstraction to uphold data processing tasks. The data storage abstraction is also composed of an *application*, a *core*, and an *execution* level. The RHEEM data storage abstraction functions symmetrically as the data processing abstraction. We shall further discuss this storage abstraction in Section 6.

3. DATA PROCESSING ABSTRACTION

In this section, we detail the abstraction layers of RHEEM and show how users can interact with the system at each layer.

3.1 Abstraction Layers

RHEEM provides a set of operators at each layer, namely, logical operators, physical operators, and execution operators. The input of the application layer is the logical operators provided by users (or generated by a declarative query parser) and the output is a physical plan. The physical plan is then passed to the core layer where multi-platform optimizations take place to produce an execution plan. In contrast to a DBMS, RHEEM decouples the physical level from the execution one. This separation allows applications to express a physical plan in terms of algorithmic needs only, without being tied to a particular processing platform.

Application Layer. A logical operator is an abstract UDF that acts as an *application-specific* unit of data processing. One can see a logical operator as a template where users provide the logic of their tasks. Such abstraction enables both ease-of-use, by hiding implementation details from users, and high performance, by allowing several optimizations, *e.g.,* seamless distributed execution.

A logical operator works on *data quanta*, which are the smallest units of data elements from the input datasets. For instance, a data quantum represents a tuple in the input dataset or a row in a matrix. This fine-grained data model allows RHEEM to apply operators in a highly parallel fashion and thus achieve better performance.

Example 1: Consider a developer who wants to offer end users logical operators to implement various ML algorithms. The developer can define three basic operators: (i) Initialize, for initializing algorithm-specific parameters, *e.g.,* initializing cluster centroids, (ii) Process, for the computations required by the ML algorithm, *e.g.,* finding the nearest centroid of a point, (iii) Loop, for specifying the stopping condition. Users implement algorithms such as SVM, K-means, and linear/logistic regression with them. □

The application optimizer translates logical operators into physical operators that will form the *physical plan* at the core layer.

Core Layer. This layer is the heart of RHEEM. It exposes a pool of *physical* operators, each representing an algorithmic decision for executing an analytic task. A physical operator is a *platform-independent* implementation of a logical operator. These operators are available to the developer to deploy a new application on top of RHEEM. Developers can still define new operators as needed.

Example 2: In the above ML example, the application optimizer maps Initialize to a Map physical operator and Process to a GroupBy physical operator. RHEEM provides two different implementations for GroupBy: the SortGroupBy (sort-based) and HashGroupBy (hash-based) operators from which the optimizer of the core level will have to choose. □

Once an application has produced a physical plan for a given input task, RHEEM divides this physical plan into *task atoms*, *i.e.,* sub-tasks, which are the units of execution. A task atom (a part of the execution plan) is a sub-task to be executed on a single data processing platform. It will then translate the task atoms into an *execution plan* by optimizing each task atom according to a target platform. Finally, it schedules each task atom to be executed on its corresponding platform. Therefore, in contrast to DBMSs, RHEEM produces execution plans that can run on multiple platforms.

Platform Layer. At this layer, *execution* operators (in an execution plan) define how a task is executed on the underlying processing platform. In other words, an execution operator is the *platform-dependent* implementation of a physical operator. RHEEM relies on existing data processing platforms to actually run input tasks.

Example 3: Again in the above ML example, the MapPartitions and ReduceByKey execution operators for Spark are one way to perform Initialize and Process. □

In contrast to a logical operator, an execution operator works on multiple data quanta rather than a single one, which enables the processing of multiple data quanta with a single function call.

Flexible operator mappings. Defining mappings between execution and physical operators is the developers' responsibility whenever a new platform is plugged to the core. Our goal is to rely on a mapping structure to model the correspondences between operators together with context information. Such context is needed for the effective and efficient execution of each operator. For instance, the Process logical operator maps to two different physical operators (SortGroupBy and HashGroupBy). In this case, a developer could use the context to provide hints to the optimizer for choosing the right physical operator at run time. Developers will provide only a declarative specification of such mappings; the system will use them to translate physical operators to execution operators.

3.2 User Interaction

We distinguish between two types of users: *end-users*, who interact with the applications, and *developers*, who interact with the system at all the three layers. We discuss below how developers define operators (UDFs) at every layer of the abstraction.

Application layer. At this layer, developers model a data processing application by specifying a set of abstract logical operators. End-users implement these operators to express their analytic tasks. RHEEM provides an abstract LogicalOperator that defines the method applyOp. Logical operators of any application extend LogicalOperator and provide an implementation of applyOp. RHEEM invokes this method at runtime to apply a logical operator. In addition to logical operators, an application developer could also expose a declarative language for users to define their tasks (*e.g.,* queries). The application is then responsible for translating a declarative query into a logical plan. Then, the application optimizer translates the logical plan into a physical plan.

Core layer. RHEEM provides a pool of physical operators for applications to produce physical plans. To enable extensibility, the

system also provides an abstract PhysicalOperator, with the abstract method applyOp, for developers to define their own physical operators. Developers define a new physical operator to fill two different needs. First, developers define a *wrapper* operator to execute the logical operator together with some physical details, such as algorithmic decisions and schema details. The wrapper operator follows the signature of the logical operator. Second, since the output of a specific operator might not fit as input of a subsequent operator, developers define *enhancer* operators to fill possible gaps between wrapper operators. For example, an application for K-means clustering might only expose the GetCentroid (for getting the closest centroid of a data point) and SetCentroids (for computing the new centroids) logical operators. GetCentroid outputs a data point and its closest centroid, while SetCentroids requires a centroid and all its closest data points. Here, the developer provides a GroupBy enhancer operator between GetCentroid and SetCentroid.

Platform layer. To model a data processing platform, developers extend the abstract ExecutionOperator and implement its applyOp method. There are two main scenarios. If a new physical operator has been defined, *e.g.,* because the developer is adding a new application, then it must be supported with a corresponding execution operator in the actual execution platform. In a different scenario, the developer is adding a new platform to the execution layer. In this case, every physical operator must be supported with a corresponding execution operator in the new execution platform. RHEEM uses these execution operators to produce an execution plan and pushes "down" execution details to the underlying platform, such as data distribution, parallel execution, and data storage. At the end, the target processing platform simply performs an execution plan in its own data and processing model.

4. MULTI-LAYER OPTIMIZATIONS

In contrast to traditional data management systems, which are tied to a specific data processing platform, RHEEM's goal is not only platform independence but also multi-platform execution. To efficiently deal with both aspects, we envision optimizations at each layer: (i) at the application layer, we validate an input task, translate it into a logical plan, and then produce an optimized physical plan, (ii) at the core layer, we translate a physical plan into an execution plan by dividing the query into *task atoms*, and (iii) at the platform layer, we further refine a task atom based on the actual platform.

4.1 Application-Layer Optimizations

In our envisioned system, users will be able to express their tasks either procedurally (via logical operators) or declaratively (using a query language). Given an input task, the application optimizer builds a logical plan and performs some pre-defined optimizations, such as operator push-down. Once the logical plan is built, the application optimizer produces an optimized physical plan by translating each logical operator into a wrapper physical operator. Recall that a wrapper receives a logical operator as input. Additionally, the application optimizer might use enhancer physical operators to boost performance; for example, it may plug-in a GroupBy physical operator followed by a CrossProduct operator to perform a cross product inside a group only. This avoids a costly cross product over the entire input dataset. As an example of this kind of optimization we refer to [19]. Then, the application sends the optimized physical plan to the core layer.

4.2 Core-Layer Optimizations

The optimizations at the core layer are the responsibility of the multi-platform task optimizer (see Figure 1). RHEEM receives a physical plan from an application and passes it to the multi-platform task optimizer to generate a plan for execution on the underlying processing platforms. We envision the multi-platform task optimizer to deal with five aspects. First, the optimizer should consider operators as first-class citizens. That is, its optimization process should be fully based on UDFs optimization techniques. We will base our solutions on different existing optimization techniques, such as Manimal [16], PACTs [25], and SOFA [22], but we also need to devise new optimization techniques to support different processing platforms. Second, the optimizer should consider rules and cost models for its optimizations as plugins and not hard-coded as in traditional database optimizers. In other words, these two aspects should be decoupled from the optimizer in order to allow for extensibility when new processing platforms are added by developers. Third, it has to consider inter-platform cost models to effectively take into account the cost of moving data and computation across underlying processing platforms. The main difficulty here comes from the fact that underlying frameworks are typically highly heterogeneous in terms of both data representations and processing paradigms. Fourth, it should divide a physical plan into task atoms according to the supported underlying data processing platforms. Recall that it is the underlying processing platform that ensures the execution of task atoms. The main challenge in this aspect is to find a way to divide a task into atoms seamlessly from users. Last, but not least, it should also apply traditional physical optimizations, whenever possible. Examples are shared scans and optimized data access paths, such as index access. Achieving this is difficult as such optimizations should be general in order to be efficient on any processing platform.

Once an execution plan is built, the multi-platform task optimizer passes it to the Executor (see Figure 1) for: (i) scheduling the resulting execution plan on the selected data processing frameworks, (ii) monitoring the progress of plan execution, (iii) coping with failures, and (iv) aggregating and returning results to users.

4.3 Platform-Layer optimizations

Once at a target processing platform, we envision a third optimization phase that uses plugged-in platform-specific optimization tools. For instance, if the selected platform for a task atom is Hadoop, we could further optimize an execution plan by using Starfish [14]. Notice that the data processing platform itself can also perform some additional optimizations, *e.g.,* if an execution plan is given as input to Spark in the form of a Shark query [26].

5. APPLICATIONS

Clearly, a large number of applications benefit from our vision. As a proof of concept, we present here a data cleaning application we developed using part of RHEEM's vision, mainly platform independence. We are currently developing two other applications: a machine learning application and a graph processing application.

5.1 Data Cleaning in RHEEM

Demand. Ensuring high quality data is challenging because of the variety of data dirtiness, such as typos, duplicates, and missing values. However, detecting errors is a combinatorial problem that quickly becomes expensive with the size of the data, thus limiting the applicability of cleaning systems.

Our solution. We built BIGDANSING, a Big Data Cleansing on top of RHEEM [19]. The two distinct features of BIGDANSING are its *ease-of-use* and *high scalability*; both natural consequences of the RHEEM abstraction vision. BIGDANSING models data quality rules with five operators, namely Scope, Block, Iterate, Detect, and GenFix. These operators allow users to capture the semantics

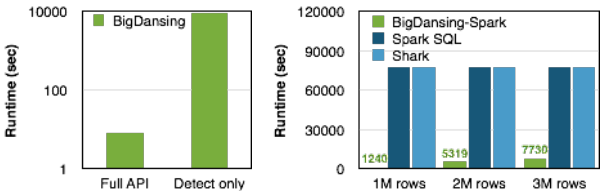

Figure 3: RHEEM execution times for violations detection.

of error detection and possible repairs generation at the application layer (see [19] for details).

Ease-of-use. The developer of the application has to come up with the physical plan of the cleaning process (preferably via an automatic optimization process). The detection part of BIGDANSING is composed of a sequence of five physical operators, which are fed with the logics coming from the corresponding logical operators. Similarly, the logical operators require only a few lines of code [19], allowing for ease-of-use.

High scalability. Figure 3 shows a comparison of the performance for a single Detect UDF versus our operators for the same task. The left-side subfigure clearly shows the benefits of the abstraction with operators that enables finer granularity for the distributed execution. The right-side subfigure shows a comparison of BIGDANSING against state-of-the-art approaches on Spark. We observe that RHEEM enables orders of magnitude better performance than baselines, which we had to stop after 22 hours. Here, as an example of extensibility, we extended the set of physical RHEEM operators with a new join operator (called IEJoin [20]) to boost performance.

5.2 When to Use RHEEM?

It should be clear at this point how the proposed three layers enable better performance and more freedom in developing applications with respect to existing solutions. However, there is a trade-off. A developer who decides to use RHEEM instead of one of the alternative systems may need to implement new operators as required by the target application. This is because our pool of default physical operators is not as exhaustive as the operators provided by the specific underlying platforms. For example, we had to implement the IEJoin operator in RHEEM to boost the performance of our data cleaning application. While this may require extra effort from a developer, we believe that this a reasonable price to pay for platform independence when performance is crucial.

6. DATA STORAGE ABSTRACTION

So far we have focused our attention on the data processing side of our vision. Symmetrically to the data processing, we envision a data storage abstraction to provide interoperability among different data storage platforms.

Figure 4 illustrates the RHEEM data storage abstraction. Each layer of abstraction has a set of operators (*i.e.,* UDFs): logical operators (*l-store*) at the application layer, physical operators (*p-store*) at the core layer, and execution operators (*x-store*) at the platform layer. At the application layer different storage applications, *e.g.,* Dropbox, or data processing platforms, *e.g.,* Hadoop or PostgreSQL, output a physical storage plan in a homogeneous format defined by RHEEM. Then, at the core layer, RHEEM takes the physical storage plan as input and produces an optimized execution storage plan. Finally, at the platform layer, a data storage platform stores or accesses a dataset according to the execution storage plan. Note that an execution storage plan is composed of *storage atoms*, *i.e.,* the counterpart of task atoms, which are processed by a different data storage platform. It is worth noting that while a data quantum is the data unit itself (*e.g.,* a tuple), a storage atom is the

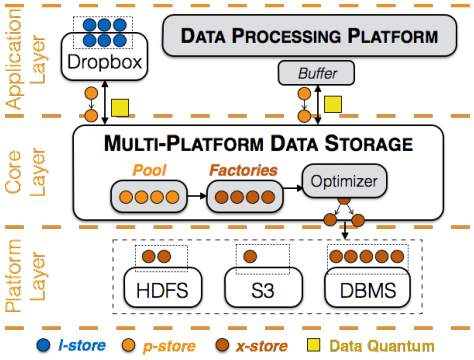

Figure 4: RHEEM data storage abstraction.

minimum unit of data quanta transformation (*e.g.,* projection).

The benefit of such a data storage abstraction is twofold. First, it provides interoperability across storage platforms to simplify users specification about how to store or transform their datasets from one platform to another. Second, it offers opportunities to fully optimize a data flow in order to further improve the performance of data processing tasks. Still, two main challenges make this problem hard: (i) how to unify the abstraction for data storage and access over multiple platforms, and (ii) how to seamlessly decide where and how to store data. Our current efforts into this direction include Cartilage [18], which is a unified data storage representation. In summary, Cartilage introduces the notion of data transformation plans, analogous to logical query plans, that specify a sequence of data transformations that should be applied to raw data as it is uploaded into a storage system. This allows for intermediate storage optimizations based on users and applications needs. For example, WWHow! [17] is a first effort for a unified data storage optimizer.

Embracing hot data. Accessing data through a unified data storage might degrade the performance of processing platforms because the data might not be in the required format. Thus, we envision processing platforms or storage applications with specialized buffers for embracing frequently accessed data in their native format.

7. RELATED WORK

The closest work to us is Musketeer [12], which provides an intermediate representation between applications and data processing platforms. While Musketeer has the merit of proposing an optimizer for the supported applications and platforms, it considers neither the costs of data movement across processing platforms nor the fact that multiple platforms may be able to perform the same job. Furthermore, it lacks the extensibility that we advocate with our proposal. In fact, only Musketeer developers can integrate new processing platforms or applications. This is in fact similar to integrating a new storage system on an existing processing platform, such as Spark or Hadoop MapReduce. In contrast, in RHEEM, users can achieve these tasks with mappings and new physical operators.

DBMS$^+$ [21] is another work that aims at embracing several processing and storage platforms for declarative processing. However, DBMS$^+$ is not adaptive and extensible as it is limited by the expressiveness of its declarative language. Furthermore, it is unclear how it abstracts underlying platforms seamlessly. BigDAWG [11] has recently been proposed as a federated system that enables users to run their queries over multiple vertically-integrated systems such as column stores, NewSQL engines, and array stores. As a result, users can leverage the advantages of each of them. However, users explicitly specify the underlying platforms (called islands) on which their queries must run on. This implies that users need to know how to divide their queries into subqueries and which underlying platform is best suited for each of them.

Other groups have been working on a general platform for big data analytics [5, 7–9, 27, 29]. For example, AsterixDB [5] offers an open data model, native data storage and indexing, declarative querying over multiple datasets, and a rule-based optimizer. SimSQL [10] compiles SQL queries into Java code that can run on top of Hadoop. Moreover, users can use UDFs to materialize views with simulated data, which enables a range of applications requiring stochastic analytics. PACTs [7] extends the MapReduce programming model with second-order functions on top of Nephele, a processing platform that RHEEM can also use as underlying platform. However, none of the above systems provides the multi-platform data processing and storage we propose with RHEEM.

8. ROAD TO FREEDOM

> "*I have walked that long road to freedom. I have tried not to falter; I have made missteps along the way. But I have discovered the secret that after climbing a great hill, one only finds that there are many more hills to climb... and I dare not linger, for my long walk is not yet ended.*"
>
> **– Nelson Mandela –**

Oftentimes, users are confronted with the hard decision to choose the right processing platform given the requirements of their analytic application. In addition, their data, born out of various processes, ends up in different storage platforms. To make things worse, the same application may have tasks requiring different platforms to be performed efficiently. Thus, there is a real urgency to free both users and data from (i) being tied to a specific platform, either for processing or storage, and (ii) going through the pain of moving from one platform to another, depending on the requirements of their applications and the characteristics of their data. While the *road to freedom* is full of challenges, RHEEM data processing and storage abstractions hold promise to achieve this freedom. As a case in point, a data cleaning application [19] is our first success towards this goal. IEJoin [20] also showcases the extensibility of RHEEM. While we have laid down the basic ideas on how to build RHEEM, many challenges remain to be addressed.

(1) Extensibility. *How to adapt to extensions and improvements in a data processing platform without requiring the developers to go into the source code? What is the right language to provide hints to the optimizer?* We envision an optimization process based on a flexible data model, such as RDF. Developers will specify mappings between operators as well as encode rule- and cost-based models in RDF triples. The optimizer will use this RDF representation as a first-class citizen in its optimization process.

(2) Multi-platform optimization. *How to divide a task into atoms, assign the best platform to each atom, and combine results?* We envision a solution based on data processing profiles and inter-platform cost models. A data processing profile denotes the type of data processing a platform can support, *e.g.,* batch-processing profile for Hadoop. An inter-platform cost model will capture different multi-platform aspects, such as the cost of transferring and transforming data from one processing platform to another.

(3) Unified storage abstraction. *How to provide a unified abstraction for data storage and access for multiple storage platforms? How to decide where and how to store data?* We envision a three-layer abstraction as discussed in Section 6. This abstraction will enable storage platforms with specialized data buffers to embrace frequently accessed data in their native format.

In summary, the above challenges can be categorized into five main research themes: (i) processing and storage abstractions, (ii) platform-independent task specification, (iii) multi-platform optimization, (iv) multi-platform execution, and (v) data storage and data movement optimizations.

9. REFERENCES

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

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

[3] Spark SQL. `http://spark.apache.org/sql/`.

[4] Powering Big Data at Pinterest. Interview with Krishna Gade. `http://goo.gl/UMGSvy`, April 2015.

[5] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14), 2014.

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

[7] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, 2010.

[8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: a flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[9] F. Bugiotti, D. Bursztyn, A. Deutsch, I. Ileana, and I. Manolescu. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *CIDR*, 2015.

[10] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD*, 2013.

[11] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. In *VLDB 2015 (demo)*, 2015.

[12] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: All for One, One for All in Data Processing Systems. In *EuroSys*, 2015.

[13] 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.

[14] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11), 2011.

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

[16] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.

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

[18] A. Jindal, J. Quiané-Ruiz, and S. Madden. CARTILAGE: adding flexibility to the hadoop skeleton. In *SIGMOD*, 2013.

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

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

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

[22] 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.

[23] M. Stonebraker and U. Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, 2005.

[24] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution over a Map-reduce Framework. *PVLDB*, 2(2), 2009.

[25] K. Tzoumas, J.-C. Freytag, V. Markl, F. Hueske, M. Peters, M. Ringwald, and A. Krettek. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. In *ICDE*, 2013.

[26] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *OSDI*, 2008.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*, 2010.

[29] J. Zhou et al. SCOPE: Parallel Databases Meet MapReduce. *VLDB J.*, 21(5), 2012.