

# The Myria Big Data Management and Analytics System and Cloud Service

Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin\*, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers†, Jennifer Ortiz, Dan Suciu, Andrew Whitaker‡, Shengliang Xu§  
University of Washington

## ABSTRACT

In this paper, we present an overview of the Myria stack for big data management and analytics that we developed in the database group at the University of Washington and that we have been operating as a cloud service aimed at domain scientists around the UW campus. We highlight Myria’s key design choices and innovations and report on our experience with using Myria for various data science use-cases.

## 1. INTRODUCTION

The need to analyze large collections of data continues to grow in industry, government, and sciences. Data analysis has even become recognized as the fourth paradigm of science [38]. Users need tools to analyze this data efficiently and easily. Many systems exist to support big data management and analytics (e.g. [61, 69, 71, 46, 43, 3, 26]), but much room remains for improvement in terms of performance and usability. Beginning in 2012, the database group at the University of Washington, in collaboration with the UW eScience Institute, has built our own big data management stack called Myria [52, 35]. We have been operating Myria as a cloud service initially in our own physical cluster and more recently in the public Amazon EC2 [1] cloud. A demonstration version of the service is available on our project website [52].

The Myria project has two main goals. The first is to build an engine sufficiently mature, fast, and easy-to-use to be adopted by domain scientists. The second goal is to build a platform for testing novel database research ideas motivated by our users’ needs. Rather than extend other big data systems such as Hadoop [69], Spark [71], or SciDB [61], we decided to build our own because we wanted the freedom to decide on all components of the system’s design. We posit

that we made the right design choices to obtain an efficient and easy-to-use platform on which to build new ideas and serve users.

At a high-level, Myria is a stack for federated data management and analytics with focus on both efficiency and usability. Myria has its own shared-nothing and elastically scalable query execution engine called MyriaX. It can also generate query plans for other backend engines, including Spark [71], SciDB [61], and PostgreSQL [59], and the plans may cross engine boundaries. Users can express their analysis with a combination of MyriaL, a relational query language with imperative extensions, and Python. Myria can be operated as a service with cloud-specific features such as elasticity and service-level agreements. Each Myria cluster deployment includes a web-based interface for interactive querying in MyriaL and a Jupyter [42] notebook server for complex Python analysis.

We designed Myria to be a federated system at its core because we find that scientists have access to many big data systems such as Spark [71], SciDB [61], and others. Each system best supports a specific type of workload. SciDB, for example, is specialized for array processing while MyriaX executes iterative queries efficiently. At the same time, these systems often share the same cluster or public cloud. With Myria, we seek to harness the power of these different backend engines while freeing the user from manipulating them separately. In Section 2, we present RACO, the Relational Algebra Compiler, which is Myria’s query optimizer and federated executor.

Our primary target with Myria is relational data; We find that the relational model, with several imperative extensions, has served us well in expressing complex data analytics. We designed MyriaL, an imperative-declarative hybrid language, to serve this purpose. Each MyriaL statement is declarative but statements can be wrapped with imperative constructs such as variable assignments and iterations. However, we also find that scientists have large collections of Python scripts for specialized algorithms. In order to use these legacy libraries and also express complex, scientific computations, we expose Python APIs for direct use of Myria and include support for expressing user-defined functions and aggregates in Python. We also provide support for a blob data type, which enables query plans with Python UDFs/UDAs to directly manipulate NumPy arrays or other specialized data types without expensive conversions. Users can write their data management and analysis using a combination of MyriaL and Python as we describe in Section 3.

\*Work done while at UW. Current affiliations: Google † University of Iowa ‡ Amazon § Pure Storage

Myria’s query execution engine, MyriaX, builds on a traditional shared-nothing, parallel database system architecture to inherit the high performance associated with pipelined query execution, physical tuning, and the minimal query start-up costs of a standing engine. We extend this architecture with important modern features including efficient support for iterative processing, elasticity, ability to read data from a variety of sources including HDFS [8] and cloud storage such as Amazon S3 [4], deployment on top of novel resource managers such as YARN [70], and a cloud service orientation to minimize barriers to adoption. We present the MyriaX engine and the details of Myria’s federated query execution plans in Section 4.

An important aspect of Myria is its service-orientation. In order to lower barriers to adoption, we developed Myria to be a cloud service from the inception of the project. We have used Myria to investigate important issues around cloud operation including how to sell data management and analytics systems as cloud services and how to leverage their elasticity for cost-effective operation as we describe in Section 5.

Finally, we have used Myria to support different groups at the University of Washington from a variety of domain sciences including oceanography, astronomy, natural language processing, and neuroscience. We describe these use-cases in Section 6 and lessons learned in Section 7.

In this paper, we present the architecture of the Myria stack, describe the known design choices and techniques that Myria adopted, and highlight Myria’s innovative components. The details of several Myria technical contributions have been published in separate papers, which we cite throughout the text. The contribution of this paper is to show how all the components come together, fill in missing pieces not published elsewhere, and report the lessons learned from building the system and supporting users from domain sciences.

## 2. RELATIONAL ALGEBRA COMPILER (RACO)

Although the Myria stack includes a shared-nothing query execution engine (MyriaX, see Section 4), the overall system was designed as a federated data analytics engine for analyzing data held by multiple backend systems, including those with non-relational data models. As data systems continue to become specialized, organizations are increasingly likely to maintain a number of such systems in a common ecosystem. Analysts and application programmers then must either adapt their code for multiple backends or settle for a lowest common denominator system. Our goal is to provide a suite of common services, including query and optimization, over these *polystore* environments that can deliver the performance of specialized systems with the convenience of general purpose systems.

The *Relational Algebra Compiler* (RACO) is Myria’s query optimizer and federated query executor. RACO adopts an extended relational algebra as the core model of computation, but supports compilation of algebra expressions into computations over selected array, graph, and key-value engines. The hypothesis underlying the design is that while performance characteristics vary widely across these systems, relational algebra is sufficient to capture the semantics of their query interfaces. For example, iterative relational algebra is sufficient to express complex graph [19,

31] and machine learning algorithms [53]. Our approach is to use rewrite rules to transform relational algebra expressions into the specific API calls, operators, or query primitives supported by the selected backend system, and use rule-based optimization to generate a federated query plan that takes advantage of the specialized features of multiple backend systems. For example, a matrix multiply expressed as a join followed by a group by can be rewritten into a call to a specialized routine in a sparse linear algebra system.

In Section 2.1, we describe the extended algebra we use as computational model. We then describe the optimization and execution process in Section 2.2.

### 2.1 Extended Relational Algebra

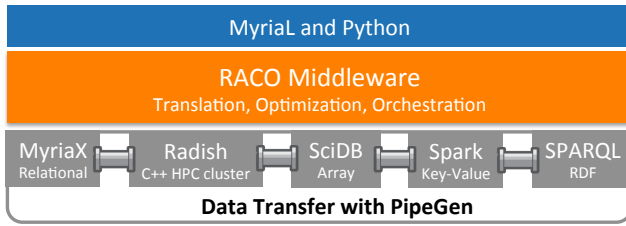
The RACO computational model is the relational algebra extended with *iteration* to enable multi-pass algorithms, a *flatmap* operator to explode non-1NF values into multiple tuples, and a *stateful apply* operator to express window functions and array-oriented neighborhood operations. We first describe these extensions, then describe how the optimization process proceeds.

**Iteration.** Modern data analytics including graph analytics and machine learning require iterative processing. RACO supports two types of iterative processing constructs: The first one is a general Do-While loop, which executes the content of each loop and synchronizes at iteration boundary to check the termination condition, which is a subquery that returns a relation with one tuple and one boolean attribute. The second one is a Do-UntilConvergence loop, which enables asynchronous processing with several runtime optimizations as we describe further in Section 4.2.2.

**Stateful Apply.** Window functions play a prominent role in a number of applications such as running sums, ranking functions, and sliding window operations on ordered datasets. They are supported in most modern relational systems and are part of the SQL standard, but many are easier to express in array-oriented systems. RACO expresses window functions with a generalization called *Stateful Apply*. This operator acts as a user-defined aggregate but produces a value for each input tuple instead of one tuple per group. Like a user-defined aggregate, stateful apply requires three function arguments: an initialization function, a step function, and an emitter function. *RunningMean* is a simple example that executes by passing a  $(count, sum)$  state along tuples, updating the state at each tuple, and emitting the current state  $sum/count$  at each tuple:

```
apply RunningMean(value) {
  [0 AS _count, 0 AS _sum]; -- init
  [_count + 1, _sum + value]; -- step
  _sum / _count; -- emit
};
```

**Flatmap.** Many use cases require a *Flatmap* operator, including frequent itemsets, entity resolution, locality sensitive hashing, and image analytics. The flatmap operator is used in some big data systems to express a non-relational “reverse aggregation” operation: exploding a single value into multiple values. This operation is critical in practical contexts, including generating ranges of integers, tokenizing a document, splitting strings into  $n$ -grams, and splitting images into image fragments.



**Figure 1:** Myria’s polystore architecture with currently supported backends.

## 2.2 Polystore Optimization

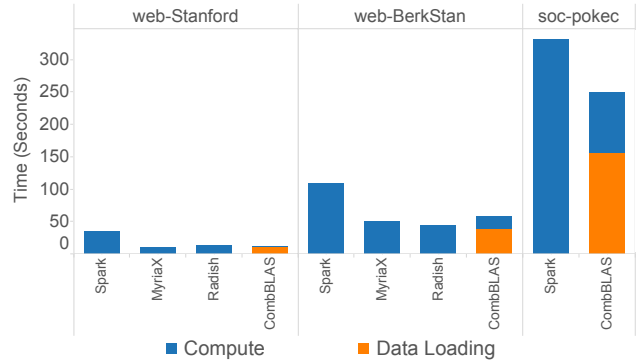
RACO uses rule-based optimization. Optimization proceeds by translating expressions in the logical Myria algebra into expressions in a physical algebra in one or more of the supported backends via a series of rewrite rules. We currently have hundreds of rules in the system for transforming logical and physical plans. These rules are fired in a fixed order until no more rules apply. Rule orders are specified by each backend independently, though rules are frequently shared across backends. Though our original design called for cost-based optimization, we found that in most of our applications, optimization problems typically result from missing rules rather than inadvertent interactions between competing rules.

The rule engine can produce federated execution plans involving computation and data movement across multiple backends, as shown in Figure 1. To register a new backend, the developer needs to provide an AST describing the API or query language supported by the backend, a set of rules mapping the logical algebra into this AST, and a set of administrative functions (e.g., querying the catalog, issuing a query, extracting results).

RACO has a locality-aware algebra used to generate plans for various parallel backends, including its own MyriaX engine (see Section 4). The algebra uses Volcano-style parallelism [33], where data is partitioned and exchange operators such as Shuffle, Broadcast, and Collect are used to communicate between partitions. RACO has optimization rules to eliminate redundant communication operators and take advantage of partitioning information in the catalogs of backend systems.

RACO’s current, default federated optimization strategy is simple: the optimizer assigns each leaf of the plan to the platform on which the dataset resides. Then the optimizer assigns the internal operators bottom-up. If the children of a binary operator are assigned to different platforms, then the optimizer inserts a data movement operator (see Section 4.3) and the process continues.

As an example motivation for a more sophisticated federated optimizer, Figure 2 evaluates sparse matrix multiplication on three of Myria’s backends: Spark, MyriaX, Radish [51], and on the CombBLAS sparse linear algebra library [20]. We compute the square of three progressively larger sparse matrices derived from real-world graphs: web-Stanford, web-BerkStan, and soc-pokec [62]. The four systems have similar execution strategies except for data loading: CombBLAS requires the data to be loaded into memory before starting computation, whereas the other three systems pipeline data loading into computation, which makes it difficult to separate data load time from the compute time. Therefore, we add data load time to compute time for Comb-



**Figure 2:** Time taken for matrix multiplication on three Myria backends and CombBLAS. For MyriaX, Radish and Spark, the graph plots “Pipelined IO + compute” since data loading and execution in these systems are pipelined. For CombBLAS, the time is the sum of data loading and computation since it requires data to be loaded into memory before starting computation. The datasets are real-world web graphs [62].

BLAS for a fair comparison. As the figure shows, MyriaX runs fastest on the web-Stanford dataset, Radish runs fastest on web-BerkStan, but both MyriaX and Radish run out of memory when processing soc-pokec, which CombBLAS executes fastest. The best performing system thus depends on the data scale, motivating in part a federated optimizer that chooses between multiple execution backends.

RACO uses the relational data model for translation and optimization. We support non-relational systems by defining relational semantics for their operations and adding rules to translate them properly. The basis of our “relational at the core” approach lies in the hypothesis that the four pervasive data models—relations, arrays, graphs, and key-values—are fundamentally isomorphic, in the sense that a computation in any one can be expressed against any of the others. Therefore, it suffices to map arrays, graphs, and key-value based systems into relational algebra in order to connect these systems together through RACO.

As a concrete example, one way to express a matrix multiply operation is as a join followed by a group by. The RACO module for the SciDB [61] backend includes a rule that recognizes this pattern and compiles it into a matrix multiply call. Similarly, we include rules to translate graph primitives (e.g., a graph traversal expressed with a while loop can be recognized and compiled into a property path query in SPARQL). The success of this approach is surprising: we do not expect every API call of each backend to be expressible in relational algebra, but the limitations of this approach have not been the bottleneck in adoption. Since we began the project, we have seen other federated systems adopt a relational approach to multi-model processing, including Metanautix’s Quest (since acquired by Microsoft and no longer available), Musketeer [31], Presto [9], and SQL++ [56] (as used in the FORWARD system [30]).

### 3. QUERY INTERFACE

In Myria, users express their data management and analysis using our declarative-imperative language called MyriaL. Myria further provides support for Python. It has both a rich Python API and support for Python user-defined functions and aggregates. We included extensive support for Python because it is today’s language of choice for data scientists. We describe both aspects of Myria’s query interface in this section.

#### 3.1 MyriaL Query Language

MyriaL is a hybrid imperative-declarative query language designed for programming *relational* algorithms ranging from simple queries to matrix operations to iterative machine learning tasks. MyriaL was designed to balance expressiveness with optimizability. MyriaL is perhaps most similar to stored procedure languages such as PL/SQL, except that no DDL statements are allowed, the only control flow statements allowed are while loops, and all variables hold relations. Even scalars are treated as relations internally. These limitations allow each MyriaL program to be represented as a forest of expressions, simplifying rewriting for optimization.

A MyriaL program consists of a sequence of assignment statements interleaved with one or more loop blocks such as `do ... while`. Loops may not be nested. The left-hand side of each assignment statement is a relation variable and the right-hand side is a relational algebra expression written in one of three syntaxes: SQL, comprehensions (expressions of the form `[from x where y emit z]`), or function calls. This flexibility has been useful; we have found users expressing different tasks in different syntaxes within the same program, as in Figure 3.

We originally envisioned a Datalog-based language but found ourselves writing queries on the whiteboard in an imperative style with loops, and so we decided to implement that language instead. In particular, tasks that involve a fixed number of iterations or iterating until a convergence condition were awkward (although not impossible) to express in Datalog assuming typical extensions, but such tasks were straightforward with a while loop. Figure 3 shows an iterative example of calculating connected components in MyriaL. Line 1 loads a graph dataset in an edge table representation. Line 2 derives the unique nodes. Line 3 initializes the component IDs. Lines 4-11 iteratively assign to each node the minimum component ID among all its neighbors. Line 12 computes the size of each component. Line 13 persists the counts. Through interactions with users, we have written a variety of machine learning and data mining tasks in MyriaL using our iterative construct, including *k*-means, frequent itemset mining, logistic regression, CART, PageRank, betweenness centrality, Markov clustering, naïve Bayes, LDA, and others.

Myria supports both user-defined functions (UDFs) and user-defined aggregates (UDAs), which users write either in MyriaL as shown in the Stateful Apply example in Section 2.1 or Python as we describe further in the next section. Users can also define SQL UDFs to be pushed down to the relational storage layer used by the MyriaX query execution engine. We describe MyriaX in Section 4.

```
1 E = scan(Graph); -- Graph(x, y) is an edge table
2 V = select distinct x from E;
3 CC = [from V emit x as node_id, x as comp_id];
4 do
5   newCC = CC + [from E, CC where E.x = CC.node_id
6                 emit E.y, CC.comp_id];
7   newCC = [from newCC emit
8             newCC.node_id, min(newCC.comp_id as comp_id)];
9   delta = diff(CC, newCC);
10  CC = newCC;
11 while [from delta emit count(*) > 0];
12 components = [from CC emit CC.comp_id, count(CC.node_id)];
13 store(components, ConnectedComponents);
```

Figure 3: Connected components in MyriaL.

#### 3.2 Python Integration

Because data scientists today favor Python as their primary programming language, Myria offers extensive support for Python in two ways: Support for Python UDFs/UDAs and a Python API.

**Python UDFs and UDAs:** In Myria, users can register Python functions that they can later use in MyriaL queries as illustrated in Figure 4. Python functions are currently only supported with the MyriaX query execution engine. MyriaX includes a blob data type so that Python UDFs and UDAs can operate on Python objects (e.g., NumPy arrays) directly. Analogous to Spark [63] and Hadoop Streaming [34], to execute user-specified Python code, MyriaX launches Python subprocesses and communicates with them using pipes to send the user’s code and the data to be processed. MyriaX serializes Python code using PiCloud’s cloudpickle [23] library to ship it to Python worker processes and serializes data using the Python cPickle [25] serializer because it is reasonably fast and supports nearly any Python data structure. MyriaX executes Python UDAs in one of two ways: It either passes one tuple at a time to the function and receives the updated state or it accumulates and passes all tuples in one invocation. We find that the latter approach can improve performance by  $1.5\times$  to  $5\times$ , especially when the state is sizable such as with a large NumPy array. Since Python UDFs and UDAs can utilize any existing Python libraries, we are able to implement many use-cases without rewriting complex calculations in MyriaL: an effort which would otherwise be error prone and expensive.

**Python API.** In addition to supporting queries in MyriaL, we also expose a fluent API for direct use in Python and Jupyter notebooks. This API is similar to the one exposed by PySpark [10], and enables query composition through successive invocations of functions on relation objects. An example of this is illustrated on lines 7-12 of Figure 4. The fluent API exposes functions for standard relational operations (e.g., projections, joins, and aggregations), loading and ingesting data (both local and remote), and user-defined operations. The API accepts native Python functions passed as arguments, a.k.a. lambdas, for its various operations (e.g., selection predicates). At runtime, the Python lambda translator (PLT) converts each embedded Python lambda into an equivalent expression in the RACO extended relational algebra. The PLT then inserts the resulting RACO expressions into a query plan constructed from the chained API invocations. Finally, the PLT submits the plan to RACO for optimization and execution.

```

1 @myria_function(name='denoise',
2                 output_type=BINARY)
3 def apply_denoise(image, mask):
4     sigma = estimate_sigma(image)
5     return nlmeans.nlmeans(image, sigma, mask)
6
7 print MyriaRelation("images")
8     .where(lambda t: t.height > 1080)
9     .where(lambda t: t.date > datetime.now())
10    .denoise()
11    .store("denoised-images")
12    .to_dataframe()

```

**Figure 4: Examples of a Python user-defined function and the Myria fluent API. This example may be executed both as an ordinary Python program or in a Jupyter notebook.**

The PLT is only able to directly translate a subset of the possible Python expressions into RACO equivalents. For example, the selection predicate on line 8 of Figure 4 has a RACO equivalent, while predicate on line 9 does not since RACO exposes no construct that represents the current time. However, for expressions with no RACO equivalent, indirect translation is possible through the use of an implicit Python UDF. Accordingly, when a lambda that cannot be directly transformed into a RACO expression is encountered, the PLT instead registers the lambda body as a Python UDF and generates a RACO-compatible invocation of it. This allows Myria to utilize higher-performing RACO-translated expressions where possible, and fall back to more robust Python UDFs for unsupported cases.

Our Python integration also allows for more complex logic to be encapsulated in extension methods [17]. Myria extension methods transparently handle the process of UDF registration and invocation during query plan construction. For example, lines 1-5 of Figure 4 show an annotated method that may be invoked through the API. We show an example of its invocation on line 10. This approach allows users to more easily build and share complex libraries built on Myria.

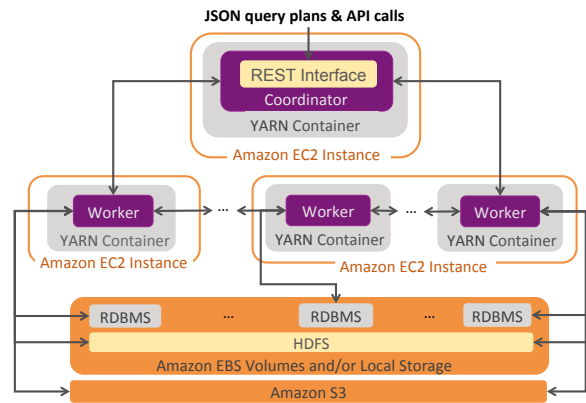
### 3.3 Updates

In Myria, all datasets are immutable. A query can read datasets from internal storage or an external source, as we describe in Section 4, and produce one or more immutable relations as output. A second execution of the same query overwrites any results. We find this simple design adequate and have not encountered a use-case where a data scientist needs to update individual tuples. For base relations that grow over time, we append new data to an existing relation via a union operator.

## 4. QUERY EXECUTION

In this section, we present Myria’s query execution engine MyriaX. We also give details of efficient data movement during federated query plan execution.

MyriaX is a parallel, shared-nothing relational query execution engine. Its design uses both state-of-the-art approaches and new techniques. Figure 5 shows the overall architecture of the MyriaX parallel query execution engine together with cluster deployment details. As the figure shows, MyriaX comprises a coordinator and multiple workers. The coordinator and each of the workers are separate MyriaX processes. Each executes in its own YARN container.



**Figure 5: MyriaX architecture with deployment details using YARN on the Amazon cloud. The coordinator can schedule multiple workers on the same physical or virtual machine.**

### 4.1 MyriaX Data Storage

MyriaX workers can read data from a variety of sources including their local file system, Amazon S3, and HDFS. The read of a single file can be automatically parallelized in the case of Amazon S3, or Myria can also assign workers to read from specific files in parallel if a dataset is partitioned across files. The system provides support for reading and parsing CSV, TSV, and a simple binary format. MyriaX also supports data ingest in several domain-specific formats such as TIPSy and NChilada [55] for astronomy simulation data. Users can also write new FileScan operators to support new file formats.

When workers store query results, however, they use relational DBMS instances for efficient, node-local storage with indexing. Subsequent queries can then read the data distributed across these instances. Myria can be configured to use any relational DBMS, including column-store systems [39]. In our default deployments, we choose PostgreSQL [59] because it is open source and we also found it to be consistently efficient.

The above design ensures both ease-of-use and high performance. By reading data from commonly used data storage systems such as HDFS and Amazon S3, Myria makes it easy for users to point the system at their data. By storing intermediate results in RDBMS instances, Myria leverages their performance and indexing capabilities.

Our approach also facilitates physical tuning by enabling users to specify how they want their data to be partitioned across the cluster and what indexes to add at the local DBMS level. As of now, users are responsible for physical tuning. For example, in the MyriaL script below, the store command hash-partitions the data on a single attribute, x:

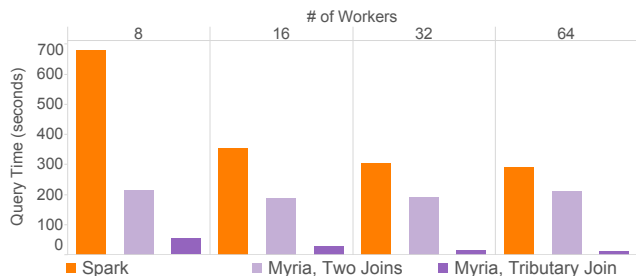
```

T = load("https://uwdb.s3.amazonaws.com/.../points.csv",
         csv(schema(x:float,y:float)));
store(T, points, [x]);

```

### 4.2 MyriaX Query Execution

MyriaX takes as input query plans generated by optimizers such as RACO (Section 2). Query plans take the form of graphs of operators. Graphs can have cycles when iterative processing, as described in 4.2.2, is involved. Operators are grouped into query fragments. Each query fragment runs in a separate thread on a worker. The set of MyriaX operators includes both relational operators (e.g., joins and



**Figure 6: Triangle query on Myria and Spark, using Spark SQL. “Two Joins” refers to using two binary joins with data shuffling. “Tributary Join” refers to using HyperCube Shuffle and Tributary Join together.**

aggregates) and specialized operators that we present in this section. Query plans are fully pipelined using Volcano-style exchange operators [33] such as Shuffle and Collect. These operators mark query fragment boundaries. Queries that do not require any data shuffling run inside a single fragment at each worker. Query plans are thus pull-based within query fragments and push-based across fragments as in traditional parallel database systems. Each operator consumes and produces batches of tuples in the form of objects of a specialized TupleBatch type. The data inside these objects is organized following the PAX format [13]: Each batch is a horizontal data partition. Inside the batch, data is stored using an in-memory columnar representation.

#### 4.2.1 HyperCube Parallel Joins

An important contribution in Myria is our support for queries that join multiple large relations. Unlike traditional star-schema queries where a large fact table must be joined with one or more small dimension tables, new data analytics workloads often require joining two or more large tables with cycles. For example, to compute triangles in a large graph represented by an edge relation, one needs to join the edge relation with itself two times as opposed to joining it with some smaller, dimension tables. For this type of query, the traditional approach, which shuffles intermediate results or replicates input tables, is expensive.

Myria has efficient algorithms, based upon solid theoretical foundations for such challenging queries. More precisely, we leverage the Shares [12] or HyperCube [16] data distribution algorithm and a new single-node multiway join operator called Leapfrog Triejoin [66]. We implement the HyperCube algorithm in a new operator that we call *HyperCube Shuffle* and a multiway join operator based on Leapfrog Triejoin that we call *Tributary Join*. Our contribution in Myria is twofold: We implement and empirically evaluate HyperCube Shuffle and Tributary Join, and we further develop important optimizations to build efficient hypercubes for arbitrary numbers of servers and select the variable order in Tributary Join. More details about this approach and these optimizations are in our paper [22].

The query that generates all the triangles from a graph represented with an Edge( $x, y$ ) relation exemplifies the performance of our approach:

```
select A.y, B.y, C.y
from Edge as A, Edge as B, Edge as C
where A.y = B.x and B.y = C.x and C.y = A.x;
```

```
E = scan(Graph); -- Graph(x, y) is an edge table
V = select distinct E.node_id from E;
do
  CC := [node_id, MIN(comp_id)] <-
    [from V emit V.node_id, V.node_id as comp_id] +
    [from E, CC where E.x = CC.node_id emit E.y, CC.comp_id];
until convergence;
store(CC, ConnectedComponents);
```

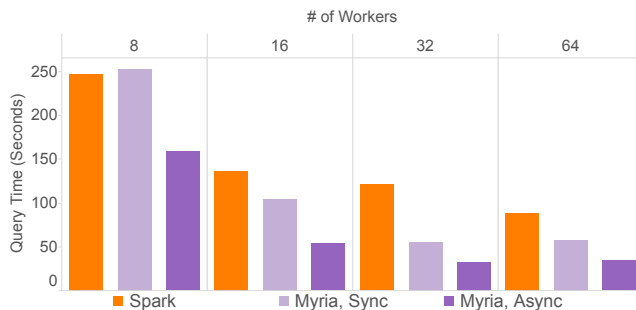
**Figure 7: Connected components in MyriaL using optimized iterations.**

Figure 6 shows query run times for the above triangle query on Myria and Spark using Spark SQL. The graph is a subset of the Twitter dataset [44] containing 4.5 million edges and 166 thousand vertices. The result has 89 million triangles, and they are materialized in memory then dropped. We run all experiments in a 16-node shared-nothing cluster interconnected by 10 Gbps Ethernet. Each machine has four Intel Xeon CPU E5-2430L 2.00GHz processors with 6 cores, 64GB DDR3 RAM and four 7200rpm hard drives. We deploy both systems on top of YARN [70] and vary the cluster size from 8 to 64 YARN containers, each with a memory limit of 14 Gigabytes. To make both systems start with the same number of partitions, we modify the size of HDFS blocks to control the number of blocks of the input dataset for Spark. Each point represents the average time of five trials. The evaluation shows that Myria’s Tributary Join and HyperCube Shuffle combination dramatically outperforms the traditional plan with two binary joins, meanwhile Myria also outperforms Spark even with a traditional plan. While it is possible to develop specialized solutions to efficiently answer particular queries, such as triangle counting in a graph, our approach is generally applicable.

#### 4.2.2 Iterative Processing

Modern data analytics requires iterations (e.g., graph analytics, machine learning, specialized scientific computations). For this reason, we have developed a new technique for iterative processing in Myria. Our approach, described in detail in a separate paper [67], takes as input an important sub-class of declarative recursive queries with aggregation and compiles them into efficient parallel query plans that can be executed either synchronously or asynchronously and with different processing priorities for intermediate and input tuples. With this approach, MyriaX is one of few engines to provide simultaneously data management capabilities with *declarative* iterative queries, and the above types of optimizations. Most other systems lack at least one of these three features as we describe in detail in our paper [67].

Our key design choice is to introduce a new physical operator that we call *IDBController*. Each IDBController accumulates and aggregates the state of one recursively computed relation and manages query execution synchrony. A query plan can have multiple IDBControllers. The rest of the query plan comprises only select, project, and join operators, which incrementally generate new facts to be aggregated by the IDBControllers. An important contribution of our work is to show that runtime optimizations, which involve selecting between synchronous and asynchronous execution as well as choosing good evaluation priority, are critical to achieving high performance. More details about this can also be found in our paper [67].



**Figure 8: Connected components on Myria and Spark (GraphX).**

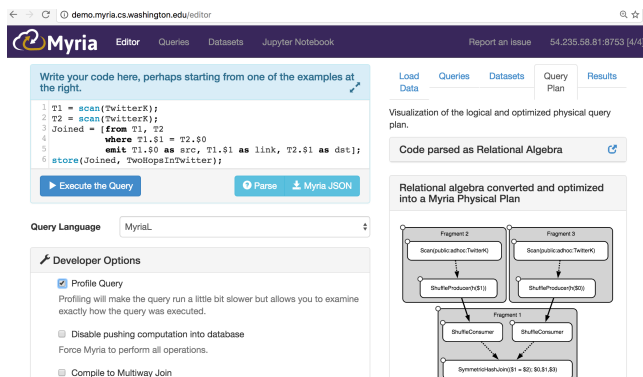
Initially, we developed the approach based on Datalog. To make adoption easier, however, we decided to enable users to specify queries declaratively using the more SQL-like MyriaL syntax. Instead of extending MyriaL’s existing Do-While loop, however, we developed a new Do-UntilConvergence syntax to make explicit the types of recursive queries with aggregation that our optimized execution supports. Figure 7 shows the connected components example using the new syntax. Other iterative queries can use the more general syntax as in Figure 3 but are executed using a simple, synchronous approach.

Figure 8 shows query run times of connected components on Myria and Spark. We use the same experiment settings as in Section 4.2.1, with a subgraph containing 221 million edges and 5 million vertices. For Spark, we use GraphX’s connected components implementation. The result shows that Myria’s asynchronous evaluation approach boosts query performance compared with synchronous execution for this application. Myria’s synchronous evaluation performance is similar to Spark/GraphX’s.

### 4.3 Federated Execution and Data Movement

A central issue with federated data analytics is data movement between constituent query engines during query execution. If a query spans an engine boundary, intermediate query results must move across systems. One approach is to export the data into a CSV file and store it in HDFS, since most big data systems are able to import using this format. This approach, however, yields high data transfer costs. Another approach is to write new operators for each pair of systems that must be interconnected. This approach, however, does not scale to large numbers of systems. Even if these new transfer operators leverage an existing framework (e.g., Apache Flume[2]), integration still requires manual modifications to the involved DBMSs.

To better support federated data analytics between MyriaX and other systems, we developed a tool called PipeGen, described in detail in a separate paper [36], that *automatically* enables optimized data transfer between arbitrary pairs of database systems. PipeGen leverages the existing ability of a DBMS to import and export delimiter-separated data to and from the file system, and replaces that functionality with a highly-optimized version that transmits Apache Arrow [15] ArrowBufs over a network socket, in parallel when possible. Our experiments show that PipeGen consistently delivers speedups of up to 3.8× between DBMSs such as MyriaX, Spark, and Hadoop when compared to transfer routed through the file system.



**Figure 9: Web interface to Myria service.**

To execute a federated query plan, a user or the RACO optimizer issues independent queries to each of several backend query execution engines extended with PipeGen. To transfer data between systems, each query includes a call to export data to or import data from a specially designated filename. The PipeGen-added code then manages the cross-system connection and data movement.

## 5. CLOUD OPERATION

One key decision that we made early on in the project is to offer Myria as a service rather than having users manage their own Myria clusters. As a result, we developed a powerful web interface for interacting with the service. Figure 9 shows a screenshot. The interface enables users to write MyriaL scripts directly in the browser, view datasets, logs of past queries, and query evaluation details. Evaluation details are produced by our Perfopticon [50] tool, an interactive query profiling tool that provides visualizations of query plans, overall query execution, data flow among servers, and execution traces. Myria’s web interface also provides access to a Jupyter notebook, enabling users to write Python scripts, which we find to be the preferred mode of interaction for our users with the Myria service. Figure 4 shows an example analysis with Myria from a Jupyter notebook. The example also leverages Python user-defined functions.

For a long time, we operated a 76-worker service in our local cluster. We recently decommissioned the cluster and now run a small public Myria deployment on Amazon EC2 [52]. For users who wish to analyze larger datasets, we provide a simple script to allow deployment of larger, customizable Myria clusters in the Amazon cloud.

### 5.1 Resource Management and Elasticity

MyriaX runs on top of REEF [60], which is a library that facilitates the development of applications, including long-running services, on top of resource managers such as YARN [70]. Figure 5 illustrates MyriaX’s cluster deployment. When a MyriaX cluster launches, the MyriaX coordinator runs in the REEF driver and workers run in REEF evaluators. Multiple MyriaX workers can execute on the same physical or virtual machine. Workers are persistent and wait to receive query plans from the coordinator. They can execute one or more queries simultaneously. This architecture enables MyriaX to share a cluster with other big data systems, provide negligible start-up overheads for short queries, and at the same time support isolation for long-running queries by spinning up new workers when needed.

The current version of the system requires that new workers be added by manually executing a command; however, support for automatically spinning up new workers is part of our design.

Given a cluster configuration, the MyriaX coordinator can schedule a query on any subset of available workers. The only current constraint is that each worker holding a partition of a base relation must execute a query fragment to scan that relation. Subsequent operations in the query plan can be scheduled on any subset of available workers by specifying parameters in the query plan. If not specified, all the available workers are used if possible (fewer workers are used for certain operators such as group-by aggregates).

In addition to MyriaX cluster elasticity, we also study memory elasticity. The idea is to free the user (or system) from deciding on the memory limits associated with MyriaX worker containers. Instead, as multiple queries execute in multiple containers on the same physical machine, a global scheduler dynamically changes their memory allocations. The goal is to eliminate the need to set memory bounds before query execution, avoid out-of-memory failures when possible, and reduce garbage collection overheads. More details are available in a separate paper [68].

## 5.2 Selling Performance

In Myria, we argue that selling resources (i.e., number of instances) is the wrong abstraction for cloud data management services because it requires users to have the expertise to determine the resource configuration that they should use. Such an expertise requirement limits the extent of users leveraging a cloud DBMS service to manage and analyze data cost-effectively.

One approach to addressing this problem is to show the performance of a cloud service on existing benchmarks [29]. Although this technique can help demonstrate the performance of a cloud service, users need to extrapolate this information to their own datasets. Other prior work in this area requires users to input a predefined query workload [21] or job profiles [40, 37] to the system in order to obtain a good configuration for the specific workload.

In our approach, we assume users do not necessarily have a workload when they first come to a cloud service. Instead of asking the user for performance requirements on a specific workload, we focus on telling the user what is possible with their data and let them pick among those options. Our approach only requires users to upload their schema and statistics over their data. The cloud service then generates a *personalized* service-level agreement (PSLA) that offers guaranteed performance levels for queries over the user’s data at a fixed rate. Rates correspond to cluster sizes but the correspondence is hidden from the user. We developed a technique to generate PSLAs [57] and implemented the approach in a new PSLAManager system. We present the details of this approach in a separate paper [57].

A fundamental challenge with performance-centric SLAs lies in cost-effectively guaranteeing the performance that the user purchases. To address this challenge, we also developed a second system, PerfEnforce (described in a recent demonstration paper [58]), which scales a cluster of virtual machines allocated to a user in a way that minimizes cost to the cloud provider while probabilistically meeting the query runtime guarantees offered by a PSLA.

We recently integrated both systems with Myria and our users will soon be able to use both tools when launching Myria clusters. We refer readers to the PSLAManager [57] and PerfEnforce [58] papers for details.

## 6. USE-CASES

**Myria in Oceanography:** Our first users were members of the UW Armbrust lab who extensively used Myria to analyze large-scale environmental flow cytometry data. A flow cytometer advances ocean water through a capillary illuminated by light of different wavelengths; the absorption and refraction patterns can be used to classify the species of individual microbial organisms in the environment. With millions of particles flowing through the system every minute, a multi-week cruise can generate terabytes of data, and the vision is to have hundreds of devices on hundreds of vessels. Since very little is known about the exact population profiles of microbial communities in the open ocean, the analysis is primarily exploratory. Myria was used to clean and calibrate the data, experiment with classification algorithms, and compute abundance and richness metrics over time. The RA-based programming model was (somewhat surprisingly) sufficient for the entire analysis.

More recently, we worked with the MIT Chisholm Lab whose members study ocean microbial life by analyzing terabytes of dense genomic sequences alongside sparse environmental data collected from ocean expeditions. We used the Myria system to explore and analyze data from two expeditions in partnership with the Intel Science and Technology Center for Big Data. These analyses included sliding window  $k$ -mer extraction, matrix multiplications with overloaded  $\oplus$  and  $\otimes$ , and a variety of ad-hoc rollup and sense-making queries.

**Myria in Astronomy:** We successfully used Myria in support of multiple applications in astronomy. The first application was concerned with the analysis of results from large-scale cosmological simulations, which can produce from a few to hundreds of terabytes of data [64]. In the simulations, the universe takes the form of particles in 3D space. The simulation output is the position and state of these particles at different points in time. We used Myria to enable astronomy collaborators from the UW N-Body shop [54] to analyze and compare galactic merger trees (i.e., how galaxies evolved from the mergers of earlier galaxies). To facilitate the analysis, we built a specialized, graphical application on top of Myria [45]. The core analysis involved joining and aggregating data across time steps in an iterative fashion to rebuild galaxy history starting from present day galaxies. The second astronomy application used Myria to analyze catalog data from sky surveys. It implemented Gaussian mixture models to classify point sources listed in the catalogs [48]. This use-case also leveraged iterative processing with two user-defined operators in the inside of the loop, corresponding to the E and M steps of the expectation maximization algorithm. The final application used Myria to analyze astronomy sky survey images themselves [49]. The implemented analysis executed an abridged version of the LSST image processing pipeline [47] with the goal of enabling individual researchers to easily execute and modify that pipeline. This application heavily utilized Myria’s Python UDFs and UDAs.



**Myria in Natural Language Processing (NLP):** The Google syntactic  $n$ -grams dataset [32], released in 2013, is an important resource in NLP research. This dataset contains billions of parsed sentence snippets, each a few words in length. The words in each snippet are annotated with their part-of-speech and their dependency relation to the root of the parsed snippet. Our UW NLP collaborators built a service on top of Myria to query this dataset efficiently. The queries in the service were heavy on joins and leveraged Myria’s physical tuning (partitioning and indexing) capabilities to achieve high performance. This use-case also leveraged Myria’s PostgreSQL storage layer with its string tokenization functions to parse the original data and store it in a structured format.

**Myria in Neuroscience:** Many sub-fields of neuroscience use image data to make inferences about the brain [41]. Our neuroscience collaborators from the UW eScience Institute used Myria to analyze diffusion MRI data of human brains at scale. The data came from the Human Connectome Project [28]. To implement the analysis, we worked with them to port their Python pipelines to Myria. We expressed the overall pipeline structure and data selection, partitioning, and grouping in MyriaL. We implemented the core image processing operations as Python UDFs and UDAs to reuse all core image analytics routines without reimplementing them. We also leveraged Myria’s blob data type to store image data fragments directly as NumPy arrays and avoid data conversions. We showed that Myria’s implementation of this use-case compared in performance to Spark and Dask [49].

## 7. LESSONS LEARNED

We draw several lessons from observing domain scientists working with Myria:

**Usability:** The need for lowering barriers to adoption and maximizing usability cannot be overstated. Users want a Myria service that is available, easy-to-use, and reliable with less effort on their behalf. These needs trump the need for high performance. Overall, we find that a continuously running cloud service does lower barriers to adoption compared with asking users to spin up and operate their own clusters. However, users also require access to examples of how to get started with processing their data (or enough examples with different types of data) and examples of analytics that resonate with their own needs. We were most successful in recruiting users when we could demonstrate some type of analysis that the researchers needed to perform executed in Myria on their data. Once users get started with the system, informative error messages are most critical, including messages related to system errors such as running out-of-memory on a query. If something goes wrong, the system must produce clear instructions for what the user should do in order to address the problem. Finally, effective user isolation is important to ensuring predictable performance and a consistent experience with the service.

We also found that building specialized vertical services on top of Myria, such as our MyMergerTree service for the visualization of galaxy evolutions, can further drive adoption by hiding all interactions with the system behind a graphical interface. The challenge, however, lies in finding ways for expert users, as opposed to systems builders, to develop such services and in facilitating their maintenance as the underlying data management system evolves.

**Domain-Specific File Formats:** Every scientific use-case starts with data in a new, domain-specific file format (TIPSY, NChilada [55], FASTQ [24], etc.). The ability to painlessly add parsers for them is paramount. Users typically already have parsers for their data in Python. They need easy ways to integrate these parsers with the system.

**Analytics Pipeline Features:** We find that the relational model and relational algebra are a good foundation even for complex analytics. However, relational algebra must be extended with iterative processing, flatmap, and stateful apply operations as we describe in Section 2.1. We find that all three features are critical in supporting the above use-cases: galaxy evolution requires recursively finding ancestor galaxies, GMMs require iteratively updating the models, genomic sequence data requires flatmaps to extract  $k$ -mers from the sequences, image analytics (in both neuroscience and astronomy) also requires flatmap to partition images into pixel blocks for parallel processing, and stateful apply is frequently used for assigning IDs to records.

**User-Defined Functions and Aggregates:** Relational algebra extended with the above three constructs took us far in terms of expressing complex scientific analytics, but scientists already have large collections of Python scripts with carefully debugged operations, such as in the neuroscience and astronomy image analytics use-cases. They also wish to interleave declarative querying with specialized algorithms written in Python (e.g., specialized spatial clustering primitives or image denoising). Support for Python UDFs and UDAs has been critical in enabling the quick implementation of scientific analysis and increasing user enthusiasm. Myria is written in Java (and not Python) and providing good support for UDFs/UDAs in a language other than the system’s language comes with some challenges: One challenge is moving data to and from the user-defined operations, including incurring data translation overheads. In Myria, one way we reduce this overhead is by supporting blob data types and thereby avoiding the need for translation; however, these blob data types are then opaque to the rest of the system. Other challenges include propagating error messages from Python functions back to users and allocating resources between the functions and the rest of the query plan when the Python UDFs/UDAs execute in separate processes.

**Data Types:** While a relational system is a strong foundation for scientific analytics, scientists have large amounts of text and multimedia data (images and videos). A big data system must have the ability to process different data types simultaneously. In Myria, we find that Python UDFs/UDAs together with blob data types can yield both ease-of-use and acceptable performance. However, this approach leaves room for optimization. Specialized systems such as SciDB, which focuses on multidimensional array processing, are not always a better solution, though. They require the reimplementation of entire analytics pipelines and sacrifice support for other data types that are not their design focus. Scientists need a system that helps them easily and efficiently process many different data types at the same time.

**Autonomic:** All the above use-cases required some type of tuning to achieve high performance: workload-driven data partitioning, indexing, and varying the degree of parallelism in the cluster. Users, however, do not tune for performance. Users execute data analysis scripts. If a script fails, for example due to an out-of-memory error, they will take the necessary actions to get their script to run. If the script

executes, however, they will most often not try to improve performance through tuning. As a result, the system performance that matters the most is the performance obtained out-of-the-box.

**User Retention:** We find that user retention is not trivial. The groups of scientists that we worked with would successfully leverage Myria for specific projects but would not adopt it across all their projects. Part of the reason is that scientists today still do not automatically turn to database systems when they start to work with new datasets or on new problems. Another issue is with the constant service availability. When we operated the service in our physical cluster, we could run it continuously at good scale (76 workers across 20 physical machines including a coordinator). This helped with user retention in the sense that the service was there and available whenever they decided to use it. The local, physical cluster, however, had too much contention between users as the system’s popularity grew. It was also tedious for our team to operate due to constant hard disk failures. In the cloud, we operate only a small cluster on a continuous basis due to costs. While we technically could launch clusters automatically for users and charge their Amazon credentials, we shied away from doing so due to the responsibility associated with manipulating user credentials. Instead, our approach is to maximally simplify and automate the process of launching Myria clusters in the cloud, including offering features such as the PSLA-Manager, as part of our start-up scripts.

## 8. RELATED WORK

Many big data systems have emerged in recent years. In this section, we contrast Myria with some of the key systems representative of different points in the overall design space.

Several systems, such as Spark [71], Flink [5], and Hadoop [69], are designed to be general-purpose big data systems, while several others, such as SciDB [61] and GraphLab [46], are designed to serve specific types of workloads. Myria is a general-purpose engine. Similar to the other general-purpose systems, it focuses on making big data analytics both fast and easy for a broad range of applications. In particular, Myria’s rich support for Python builds on the above systems’ support for that same language. Unlike the above systems, however, Myria has more advanced federation capabilities and MyriaX is foremost a relational engine: it uses relational, node-local data stores internally (i.e., PostgreSQL instances), leverages long-running workers, and pipelined query execution (which several [69, 71] but not all [5] the above big data systems lack). It also differs in other design choices including its support for declarative queries compiled into synchronous and asynchronous iterative query plans, and multiway joins. These design choices contribute to Myria’s performance.

Many systems target relational queries on shared-nothing architectures. HadoopDB [11] combines local RDBMSs for query evaluation with Hadoop for communication. While MyriaX has the ability to push some of the computation down to local RDBMSs before reading data out, it uses these RDBMSs mainly for storage. It provides its own query execution layer and data shuffling primitives. For Massively Parallel Processing (MPP) databases, Greenplum [7] builds on top of local PostgreSQL instances with its own parallel processing layer similar to Myria, while Teradata [65] has a similar architecture built on top of its own local data man-

agement units called Access Module Processors (AMPs). Asterix [14] exploits Log-Structured Merge (LSM) trees for internal storage and indexing with its own runtime execution layer Hyracks [18]. Impala [43] reads data from HDFS and Amazon S3 instead of local RDBMS instances, while Myria can utilize RDBMS features such as indexing for better performance. Compared with all these relational systems, Myria includes novel iterative processing and multiway join algorithms together with cloud-specific features. Myria also supports powerful polystore processing capabilities, where queries can be executed by one or more big data systems sharing the same cluster.

Myria is part of the recently introduced BigDAWG stack [27]. Unlike BigDAWG, Myria hides the data model differences between the federated backends. Gog et al. proposed Musketeer [31], a federated system unifying graph and relational systems with relational algebra as a common programming model. However, no support for array-based or matrix-oriented systems is described.

Similar to Myria, Snowflake [26] is a recent, elastic data warehouse designed for the cloud. The two systems, however, have multiple important differences. Unlike MyriaX, Snowflake uses Amazon S3 as its primary data store; it does not index data; and it spawns new processes to execute new queries. Additionally, it does not provide polystore query processing capabilities. Similar to our PSLA-Manager, Snowflake hides the details of compute instances from users, however it does not give insights about expected performance to help users guide their cluster size selections.

Several other RDBMSes, such as Redshift [3] and Azure SQL Database [6], are available as cloud services. Myria’s innovative cloud service features such as PSLAs could be implemented on top of these other services.

Finally, for additional work related to specific Myria features, we also refer the readers to the various papers detailing these components [22, 36, 49, 57, 58, 68, 67].

## 9. CONCLUSION

The Myria stack combines performance and ease-of-use for big data management and analytics. It provides support for federated analytics and is available as a cloud service. Myria is an open source project. Its GitHub repository can be found through our project website [52].

## 10. ACKNOWLEDGMENTS

The Myria project is supported in part by the National Science Foundation through NSF grants IIS-1247469, AITF-1535565, and IIS-1524535, gifts from EMC, Amazon, the Intel Science and Technology Center for Big Data, Facebook, an award from the Gordon and Betty Moore Foundation and the Alfred P Sloan Foundation, the Washington Research Foundation Fund for Innovation in Data-Intensive Discovery, and the UW eScience Institute. Dylan Hutchison is supported in part by an NSF Graduate Research Fellowship DGE-1256082.

## 11. REFERENCES

- [1] Amazon EC2. <https://aws.amazon.com/ec2/>.
- [2] Amazon Flume. <https://flume.apache.org/>.
- [3] Amazon Redshift. <https://aws.amazon.com/redshift/>.
- [4] Amazon S3. <https://aws.amazon.com/s3/>.
- [5] Apache flink. <http://flink.apache.org/>.

- [6] Azure SQL database. <https://azure.microsoft.com/en-us/services/sql-database/>.
- [7] Greenplum. <http://pivotal.io/big-data/pivotal-greenplum-database>.
- [8] Hadoop Distributed File System. [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html).
- [9] Presto. <https://prestodb.io/>.
- [10] Spark Python programming guide. <http://spark.apache.org/docs/0.9.0/python-programming-guide.html>.
- [11] A. Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proc. of the VLDB Endowment*, 2009.
- [12] F. N. Afrati et al. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.
- [13] A. Ailamaki et al. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 2002.
- [14] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. In *VLDB*, 2014.
- [15] Apache Arrow. <https://arrow.apache.org/>.
- [16] P. Beame et al. Communication steps for parallel query processing. In *PODS*, 2013.
- [17] G. M. Bierman et al. Lost in translation: formalizing proposed extensions to C#. In *OOPSLA*, 2007.
- [18] V. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. 2011.
- [19] Y. Bu et al. The HaLoop approach to large-scale iterative data analysis. *The VLDB Journal*, 2012.
- [20] A. Buluç et al. The combinatorial BLAS: Design, implementation, and applications. *IJHPCA*, 2011.
- [21] U. Cetintemel et al. Query steering for interactive data exploration. In *CIDR*, 2013.
- [22] S. Chu et al. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD 2015*.
- [23] Cloudpickle. <https://github.com/cloudpipe/cloudpickle>.
- [24] P. J. A. Cock et al. The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants. *Nucleic Acids Research*, 2010.
- [25] Python cPickle. <https://docs.python.org/2/library/pickle.html#module-cPickle>.
- [26] B. Dageville et al. The snowflake elastic data warehouse. In *SIGMOD*, 2016.
- [27] J. Duggan et al. The BigDAWG polystore system. *SIGMOD Record*, 2015.
- [28] D. C. V. Essen et al. The WU-Minn human connectome project: an overview. *Neuroimage*, 2013.
- [29] A. Floratou et al. When free is not really free: What does it cost to run a database workload in the cloud? In *TPCTC*, 2012.
- [30] Y. Fu et al. Forward: Data-centric UIs using declarative templates that efficiently wrap third-party JavaScript components. *Proc. of the VLDB Endowment*, 2014.
- [31] I. Gog et al. Musketeer: All for one, one for all in data processing systems. In *EuroSys*, 2015.
- [32] Y. Goldberg et al. A dataset of syntactic-ngrams over time from a very large corpus of English books. In *\*SEM*, 2013.
- [33] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE TKDE*, 1994.
- [34] Apache Hadoop streaming. <https://hadoop.apache.org/docs/r1.2.1/streaming.html>.
- [35] D. Halperin et al. Demonstration of the Myria big data management service. In *SIGMOD*, 2014.
- [36] B. Haynes et al. Pipegen: Data pipe generator for hybrid analytics. In *SOCC*, 2016.
- [37] H. Herodotou et al. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *SOCC*, 2011.
- [38] A. J. G. Hey et al. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
- [39] S. Idreos et al. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 2012.
- [40] V. Jalaparti et al. Bridging the tenant-provider gap in cloud services. In *SOCC*, 2012.
- [41] N. Ji et al. Technologies for imaging neural activity in large volumes. *Nat. Neurosci.*, 2016.
- [42] T. Kluyver et al. Jupyter notebooks - a publishing format for reproducible computational workflows. In *ELPUB 2016*.
- [43] M. Kornacker et al. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, 2015.
- [44] H. Kwak et al. What is Twitter, a social network or a news media? In *WWW*, 2010.
- [45] S. Loebman et al. Big-data management use-case: A cloud service for creating and analyzing galactic merger trees. In *DanaC 2014 in conjunction with SIGMOD*.
- [46] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *Proc. of the VLDB Endowment*, 2012.
- [47] LSST Data Management. <http://dm.lsst.org/>.
- [48] R. Maas et al. Gaussian mixture models use-case: In-memory analysis with Myria. In *Proc. of the 3rd IMDM Workshop at VLDB*, 2015.
- [49] P. Mehta et al. Comparative Evaluation of Big-Data Systems on Scientific Image Analytics Workloads. *ArXiv e-prints*, Dec. 2016.
- [50] D. Moritz et al. Perfopticon: Visual query analysis for distributed databases. *Computer Graphics Forum (Proc. EuroVis)*, 2015.
- [51] B. D. Myers. High-performance parallel systems for data-intensive computing. <http://www.cs.washington.edu/tr/2016/02/UW-CSE-16-02-02.pdf>.
- [52] Myria: Big Data management as a Cloud service. <http://myria.cs.washington.edu/>.
- [53] M. Navas et al. Efficient computation of PCA with SVD in SQL. In *Proceedings of the 2nd Workshop on Data Mining Using Matrices and Tensors*, 2009.
- [54] University of Washington N-Body Shop. <http://www-hpcc.astro.washington.edu/>.
- [55] N-body software tools. <http://hpcc.astro.washington.edu/tools/>.
- [56] K. W. Ong et al. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv:1405.3631*, 2014.
- [57] J. Ortiz et al. Changing the face of database cloud services with personalized service level agreements. In *CIDR*, 2015.
- [58] J. Ortiz et al. PerfEnforce demonstration: Data analytics with performance guarantees. In *SIGMOD*, 2016.
- [59] PostgreSQL: The world's most advanced open source database. <https://www.postgresql.org/>.
- [60] Apache REEF. <http://reef.apache.org/>.
- [61] J. Rogers et al. Overview of SciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.
- [62] Stanford large network dataset collection. <https://snap.stanford.edu/data/>.
- [63] Apache Spark. <http://spark.apache.org/>.
- [64] V. Springel et al. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 2005.
- [65] Teradata. <http://www.teradata.com/>.
- [66] T. L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, 2012.
- [67] J. Wang et al. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. of the VLDB Endowment*, 2015.
- [68] J. Wang et al. Toward elastic memory management for cloud data analytics. In *BeyondMR with SIGMOD*, 2016.
- [69] T. White. *Hadoop: The Definitive Guide*. 2009.
- [70] Apache Hadoop YARN. <https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/>.
- [71] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.