Musketeer: all for one, one for all in data processing systems

Ionel Gog Malte Schwarzkopf Natacha Crooks[†] Matthew P. Grosvenor
Allen Clement[†∗] Steven Hand[∗]
University of Cambridge [†] *Max Planck Institute for Software Systems*
∗ now at Google, Inc.

Abstract

Many systems for the parallel processing of big data are available today. Yet, few users can tell by intuition which system, or combination of systems, is "best" for a given workflow. Porting workflows between systems is tedious. Hence, users become "locked in", despite faster or more efficient systems being available. This is a direct consequence of the tight coupling between user-facing front-ends that express workflows (e.g., Hive, SparkSQL, Lindi, GraphLINQ) and the back-end execution engines that run them (e.g., MapReduce, Spark, PowerGraph, Naiad).

We argue that the ways that workflows are defined should be decoupled from the manner in which they are executed. To explore this idea, we have built Musketeer, a workflow manager which can *dynamically* map front-end workflow descriptions to a broad range of back-end execution engines.

Our prototype maps workflows expressed in four high-level query languages to seven different popular data processing systems. Musketeer speeds up realistic workflows by up to 9× by targeting different execution engines, without requiring any manual effort. Its automatically generated back-end code comes within 5%–30% of the performance of hand-optimized implementations.

1. Introduction

Choosing the "right" parallel data processing system is difficult. It requires significant expert knowledge about the programming paradigm, design goals and implementation of the many available systems. Even with this knowledge, any move between systems requires time-consuming re-implementation of workflows. Furthermore, head-to-head comparisons are difficult because systems often make different assumptions and target different use cases. Users therefore stick to their known, favorite system even if other, "better" systems offer superior performance or efficiency gains.

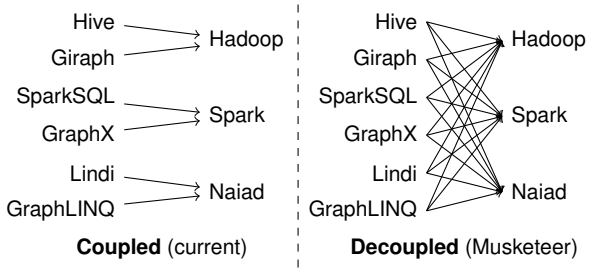


Figure 1: Decoupling front-end frameworks and back-end execution engines *(right)* increases flexibility.

We evaluated a range of contemporary data processing systems – Hadoop, Spark, Naiad, PowerGraph, Metis and GraphChi – under controlled and comparable conditions. We found that *(i)* their performance varies widely depending on the high-level workflow; *(ii)* no single system always outperforms all others; and *(iii)* almost every system performs best under some circumstances (§2).

It thus makes little sense to force the user to target a single system at workflow implementation time. Instead, we argue that *users should, in principle, be able to execute their high-level workflow on any data processing system* (§3). Being able to do this has three main benefits:

1. Users write their workflow once, in a way they choose, but can easily execute it on alternative systems;
2. Multiple sub-components of a workflow can be executed on different back-end systems; and
3. Existing workflows can easily be ported to new systems.

In this paper, we present our Musketeer proof-of-concept workflow manager to show that this is feasible, and that the resulting implementations are competitive with hand-written baseline implementations for specific systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2741948.2741968

Note: In the electronic version of this paper, most figures link to descriptions of the experiments and our data sets (http://goo.gl/BMdT0o).

To decouple workflows from their execution, we rely on the fact that users prefer to express their workflows using high-level *frameworks*, which abstract the low-level details of *distributed execution engines* (Figure 1). For example, the Hive [40] and Pig [35] frameworks present users with a SQL-like querying interface over the Hadoop MapReduce execution engine; SparkSQL and GraphX [15] offer SQL primitives and vertex-centric interfaces over Spark [43]; and Lindi and GraphLINQ [31] offer the same over Naiad [34].

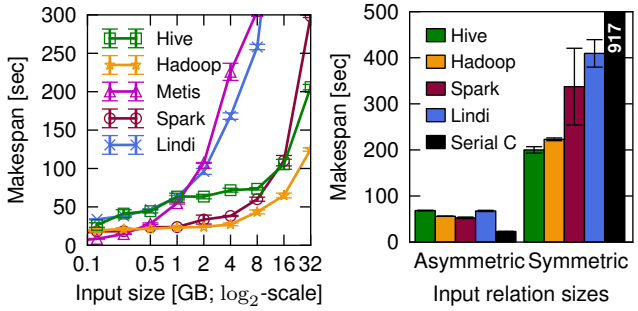
Musketeer breaks the tight coupling between frameworks and execution engines (§3). It achieves this by *(i)* mapping workflow specifications for front-end frameworks to a common intermediate representation; *(ii)* determining a good decomposition of the workflow into jobs; and *(iii)* auto-generating efficient code for the chosen back-end systems.

Musketeer currently supports four front-end frameworks (Hive, Lindi, a custom SQL-like DSL with iteration, and a graph-oriented "Gather-Apply-Scatter" DSL). It can map workflows to seven back-end execution systems: Hadoop, Spark, Naiad, PowerGraph, GraphChi, Metis and simple, serial C code. Users can explicitly target back-end execution engines, or leave it to Musketeer to automatically choose a good mapping using a simple heuristic (§5).

In a range of experiments with real-world workflows, Musketeer offers compelling advantages (§6):

1. **Better system mapping:** Musketeer enables existing workflows implemented for Hive on Hadoop MapReduce to be executed on alternative systems, and achieves a 2× speedup on a TPC-H query workflow as a result.
2. **Optimization of executed code:** by choosing the most suitable Naiad execution primitive independent of the front-end used to implement the workflow, Musketeer speeds up a TPC-H query workflow by up to 9×.
3. **Intelligent system combination:** Musketeer can combine different execution engines within a workflow, and doing so outperforms fixed, single system mappings for a cross-community PageRank workflow.
4. **Competitive generated code:** Musketeer's generated code has no more than a 5–30% overhead over hand-optimized baseline implementations.
5. **Good automatic system choice:** an automated heuristic for choosing back-ends in Musketeer derives reasonably good mappings without manual user action.

These results and our experience of using Musketeer in practice (§7) indicate that decoupling front-end and back-end systems can bring real benefits. Nonetheless, we believe our work represents only the first step in a promising direction. In Section 8, we describe current limitations of our system, and suggest some concrete future work before discussing related research (§9) and concluding (§10).



(a) `PROJECT` on a relation. **(b)** `JOIN` two data sets.

Figure 2: Different systems perform best for simple queries. Lower is better; error bars show min/max of three runs.

2. Motivation

There are many diverse "big data" processing systems and more keep appearing.[2] This makes it difficult to determine which system is best for a given workflow, data set and cluster setup. We illustrate this using a set of simple benchmarks. In all cases, we run over a shared HDFS installation that stores input and output data, and we either implement jobs directly against a particular execution engine, or use a front-end framework with its corresponding native back-end. We measure the *makespan* – i.e., the entire time to execute a workflow, including the computation itself and any data loading, pre-processing and output materialization required.

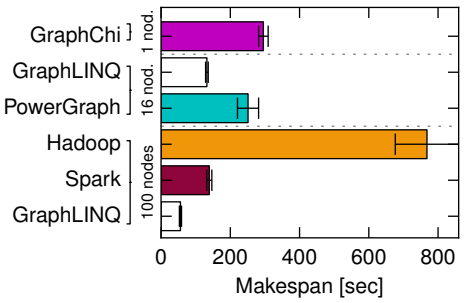
We find that no single system systematically outperforms all others. Roughly speaking, system performance depends on: (*i*) the size of the input data, as single machine frameworks outperform distributed frameworks for small inputs; (*ii*) the structure of the data, since skew and selectivity impact I/O performance and work distribution; (*iii*) engineering decisions, with e.g., the cost of loading inputs varying significantly across systems; and (*iv*) the computation type, since specialized systems often operate more efficiently.

2.1 Query processing micro-benchmarks

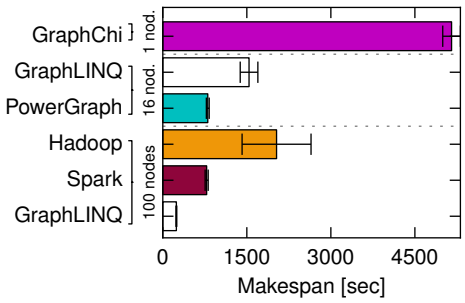
Query-based data analytics workflows often consist of relational operators. In the following, we consider the behavior of two operators in an isolated micro-benchmark. Here we use a local cluster of seven nodes as an example of a small-scale data analytics deployment. Later experiments show that our results generalize to more complex workflows and to larger clusters.

Input size. We first look at a simple string processing workload in which we extract one column from a space-separated, two column ASCII input. This corresponds to a `PROJECT` query in SQL terms, but is also reminiscent of a common pattern in log analysis batch jobs: lines are read from storage, split into tokens, and a few are written back. We consider input sizes ranging from 128MB up to

[2] For a brief summary of systems' properties, see Table 3.



(a) Orkut (3.0M vertices, 117M edges).



(b) Twitter (43M vertices, 1.4B edges).

Figure 3: Varying makespan for PageRank on social network graphs; lower is better; error bars: $\pm\sigma$ of 10 runs.

32GB. Figure 2a compares the makespan of this workflow on five different systems. Two of these are programmer-friendly SQL-like front-ends (Hive, Lindi), while the others require the user to program against a lower-level API (Hadoop, Metis and Spark). For small inputs ($\leq$ 0.5GB), the Metis single-machine MapReduce system performs best.[3] This matters, as small inputs are common in practice: 40–80% of Cloudera customers' MapReduce jobs and 70% of jobs in a Facebook trace have $\leq$ 1GB of input [8].

I/O efficiency. Once the data size grows, Hive, Spark and Hadoop all surpass the single-machine Metis, not least since they can stream data from and to HDFS in parallel. However, since there is no data re-use in this workflow, Spark performs worse than Hadoop: it loads all data into a distributed in-memory RDD [43] before performing the projection. The Lindi front-end implementation for Naiad performs surprisingly poorly; we tracked this down to an implementation decision in the Naiad back-end, which uses only a single input reader thread per machine, rather than having multi-threaded parallel reads. Since the `PROJECT` benchmark is primarily limited by I/O bandwidth, this decision proves detrimental.

Data structure. Second, we consider a `JOIN` workflow. This is highly dependent on the structure of the input data: it may generate less, more, or an equal amount of output compared to its input. We therefore measure two different cases: (*i*) an input-skewed, *asymmetric* join of the 4.8M vertices of a social network graph (LiveJournal) and its 69M edges, and (*ii*) a *symmetric* join of two uniformly randomly generated 39M row data sets. Figure 2b shows the makespan of different systems (plus a simple implementation in serial C code) for this workflow. The unchallenging asymmetric join (producing 1.28M rows/1.9GB) works best when executed in single-threaded C code on a single machine, as the computation is too small to amortize the overheads of distributed solutions. The far larger symmetric join (1.5B rows/29GB), however, works best on Hadoop. Other systems suffer from inefficient I/O (e.g., Lindi using a single-threaded writer), or have overhead due to constructing in-memory state and scheduling tasks sub-optimally (Spark).

[3] The bottleneck here is not the computation, but reading the data from HDFS. With the data already local, Metis performs best up to 2 GB.

2.2 Iterative graph processing

Many common workflows involve iterative computations on graphs (e.g., social networks). In the following, we compare different systems running PageRank on such graphs. We also vary the size of the EC2 cluster (`m1.xlarge` instances) in order to determine systems' efficiency at different scales.

Performance. Several specialized graph processing systems based on a vertex-centric or gather-and-scatter (GAS) approach have been built. These computation paradigms are limited, but can deliver significantly better performance for graph workloads. In Figure 3, we show the makespan of a five-iteration PageRank workflow on the small Orkut and the large Twitter graph. It is evident that graph-oriented paradigms have significant advantages for this computation: a GraphLINQ implementation running on Naiad outperforms all other systems.[4] PowerGraph also performs very well, since its vertex-centric sharding reduces the communication overhead that dominates PageRank.

Resource efficiency. However, the fastest system is not always the most *efficient*. While PageRank in GraphLINQ using 100 Naiad nodes has the lowest runtime in Figure 3b, PowerGraph performs better than GraphLINQ when using only 16 nodes (due to its improved sharding).[5] Moreover, when the graph is small (e.g., in Figure 3a), GraphChi performs only 50% worse than Spark on 100 nodes, and only slightly worse than PowerGraph on 16 nodes, despite using only one machine.

2.3 Summary

Our experiments show that the "best" system for a given workflow varies considerably. The right choice – i.e., the fastest or most efficient system – depends on the workflow, the input data size and the scale of parallelism available.

[4] Only the GraphLINQ front-end for Naiad is shown here; Lindi is not optimized for graph computations and performs poorly.

[5] Running PowerGraph on 32 or 64 nodes showed no benefit over 16 nodes.

This information may not be available at workflow implementation time, which motivates our approach of *decoupling* workflow expression from the execution engine used.

3. All for one, one for all data processing

We believe that a decoupled data processing architecture (Figure 4) gives users additional flexibility. In this approach, we break the execution of a data processing workflow into three layers. First, a user specifies her workflow using a *front-end framework*. Next, this workflow specification is translated into an *intermediate representation*. Third, jobs are generated from this representation and executed on one or more *back-end execution engines*.

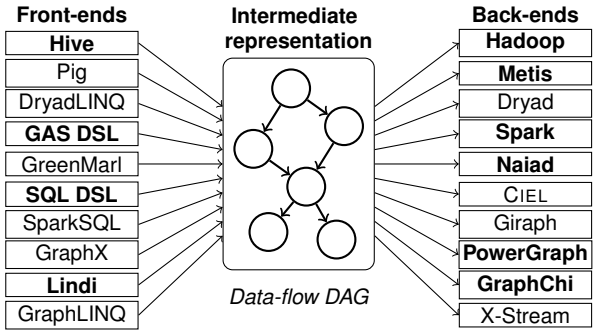


Figure 4: To decouple processing, we translate front-end workflow descriptions to a common intermediate representation from which we generate jobs for back-end execution engines. Our Musketeer prototype supports systems in **bold**.

We give an overview of the three layers below; §4 will describe their realization in our Musketeer prototype.

Front-ends. User-facing high-level abstractions for workflow expression ("frameworks") act as front-ends to the system. Many such frameworks exist: SQL-like querying languages and vertex-centric graph abstractions are especially popular. We assume that users write their workflows for such frameworks. The front-end workflow specifications must then be *translated* to a common form; we do this by either parsing the user input directly, or by building an API-compatible shim for the front-end.

Intermediate representation. Ideally, all available front-end frameworks and back-end execution engines would agree on a single common intermediate representation (IR). The IR must simultaneously (*i*) be sufficiently expressive to support a broad range of workflows, and (*ii*) maintain enough information to optimize back-end job code to a level competitive with a competent hand-coded implementation.

Our intermediate representation is a dynamic directed acyclic graph (DAG) of data-flow operators, with edges corresponding to input-output dependencies. This abstraction is general: it supports specific operator types (cf. Dryad's vertices [19]) and general user-defined functions (UDFs); it can handle iteration by successive expansion of the DAG (as in CIEL [32] and Pydron [30]); and it can be extended with new operators in order to enable end-to-end optimizations.

We can also perform query optimizations on the data-flow DAG (e.g., to reduce intermediate data volume where possible), as is already commonly done between front-end and back-end in other systems [21, 41].

Back-ends. Finally, the system must generate code for specific distributed data processing systems ("execution engines") at the back-end. A naïve approach would simply generate a job for each operator, but this fails to exploit opportunities for optimization within the execution engines (e.g., sharing data scans). Instead, we typically want to run as few independent jobs as possible.

However, some execution engines have limited expressivity and therefore require the data-flow DAG to be partitioned into multiple jobs. Many valid partitioning options exist, depending on the workflow and the execution engines available. In §5, we show that exploring this space is an instance of an NP-hard problem (k-way graph partitioning), and introduce a heuristic to solve it efficiently for large DAGs.

Given a suitable partitioning, we *generate* jobs for the chosen execution engines and *dispatch* them for execution.

Extensibility. Our approach is extensible: new front-end frameworks can be added by providing translation logic from framework constructs to the intermediate representation. Similarly, further back-end execution engines can be supported as they emerge by adding appropriate code templates and code generation logic.

Limitations. Decoupling increases flexibility, but it may obfuscate some end-to-end optimization opportunities from expert users. Our scheme is best suited for non-specialist users writing analytics workflows for high-level front-end frameworks. This is common in industry: up to 80% of jobs running in production clusters come from front-end frameworks such as Pig [35], Hive [40], Shark [41] or DryadLINQ [42], according to a recent study [8].

4. Musketeer implementation

Musketeer is our proof-of-concept implementation of the decoupled "all for one, one for all" approach that we advocate. It translates a workflow defined in a front-end framework into an intermediate representation, applies optimizations and generates code for suitable back-end execution engines. In this section, we describe Musketeer in detail. Figure 5 illustrates the different stages a Musketeer workflow proceeds through from specification to execution.

4.1 Workflow expression

Distributed execution engines simplify data processing by shielding users from the intricacies of writing parallel, fault-tolerant code. However, they still require users to express their computation in terms of low-level primitives, such as `map` and `reduce` functions [10] or message-passing ver-

```
SELECT id, street, town FROM properties AS locs;
locs JOIN prices ON locs.id = prices.id
 AS id_price;
SELECT street, town, MAX(price) FROM id_price
 GROUP BY street AND town AS street_price;
```

Listing 1: Hive code for *max-property-price* workflow.

tices [34]. Hence, higher-level "frameworks" that expose more convenient abstractions are commonly built on top.

Musketeer supports two types of front-end frameworks: (*i*) SQL-like query languages, and (*ii*) vertex-centric graph processing abstractions.

4.1.1 SQL-like data analytics queries

Query languages based on SQL are used to express relational queries on data of tabular structure. Listing 1 shows an example analytics workflow that computes the most expensive property on each street for a real-estate data set.

Musketeer currently supports three SQL-like data analytics front-ends: Hive, Lindi and BEER, our own domain-specific workflow language with support for iteration. Translation from these front-ends to the intermediate representation proceeds by mapping the relational operations to operators in the IR DAG; most relational primitives have directly corresponding Musketeer IR operators.

4.1.2 Graph computations

Domain-specific front-end frameworks for expressing graph computations are popular: Pregel [26] and Giraph abstract over MapReduce, the GreenMarl DSL [18] can emit code for multi-threaded and distributed runtimes, and GraphLINQ offers graph-specific APIs over Naiad vertices and timestamps [28]. These front-ends include user-defined code that is concurrently instantiated for every vertex, and adjacent vertices communicate using messages in repeated rounds. This vertex-centric programming pattern is generalized by the **G**ather, **A**pply and **S**catter (GAS) model in PowerGraph [14]. In this paradigm, data are first gathered from neighboring nodes, then vertex state is updated and, finally, the new state is disseminated (scattered) to the neighbors.

Musketeer currently supports graph computations via a domain-specific front-end framework built around combining the GAS model with our BEER DSL. Users run graph computations by defining the three GAS steps, with each step represented by relational operators or UDFs. In Listing 2, we show the implementation of PageRank in Musketeer's GAS front-end framework.

While HiveQL and our BEER DSL have directly corresponding operators in the IR, the GAS DSL requires both syntactic translation and transformation from the vertex-centric paradigm to the data-flow DAG. Musketeer uses idiom recognition to achieve this, which we describe in §4.3.1.

```
GATHER = {
 SUM (vertex_value)
}
APPLY = {
 MUL [vertex_value, 0.85]
 SUM [vertex_value, 0.15]
}
SCATTER = {
 DIV [vertex_value, vertex_degree]
}
ITERATION_STOP = (iteration < 20)
ITERATION = {
 SUM [iteration, 1])
}
```

Listing 2: Gather-Apply-Scatter DSL code for PageRank.

4.1.3 Other workloads

In addition to SQL-like query languages and GAS-style graph computations, Musketeer can also support other types of front-ends. If their abstractions map to Musketeer IR operators, they can be translated directly. Abstractions for which no IR operator exists can be mapped to a user-defined function (UDF), or to a "native" back-end via a "black box" operator. We discuss extension to other front-ends in §8.

4.2 Intermediate representation

Musketeer uses a directed acyclic graph (DAG) of data-flow operators as its intermediate representation. We chose this abstraction because it is expressive [19, 32, 43] and amenable to analysis and optimization [21, 23].

Musketeer's set of operators is extensible: not all front-ends use all operators, and not all back-ends must support all operators. Our initial set of operators is loosely based on relational algebra and covers the most common operations in industry workflows [8]. It includes `SELECT`, `PROJECT`, `UNION`, `INTERSECT`, `JOIN` and `DIFFERENCE`, plus aggregators (`AGG`, `GROUP BY`), column-level algebraic operations (`SUM`, `SUB`, `DIV`, `MUL`), and extremes (`MAX`, `MIN`). This set of operators is, in our experience, already sufficient to model many widely-used processing paradigms. For example, MapReduce workflows can be directly modeled as a `MAP`, `GROUP BY` and `AGG` step, and many complex graph workflows can be mapped to a specific `JOIN`, `MAP`, `GROUP BY` pattern, as shown by GraphX [15] and Pregelix [3].

However, workflows may also involve iterative computations. To allow data-dependent iteration, Musketeer must be able to dynamically extend the IR DAG based on operators' output. We use a `WHILE` operator to do this: it successively extends the DAG every time another iteration is required. As shown by Murray [33, §3.3.3], a DAG with this facility is sufficient to achieve Turing completeness as it can express all while-programs (though not necessarily efficiently).

Optimizing the IR. Many front-end frameworks already optimize workflows before execution. For example, Pig [35],

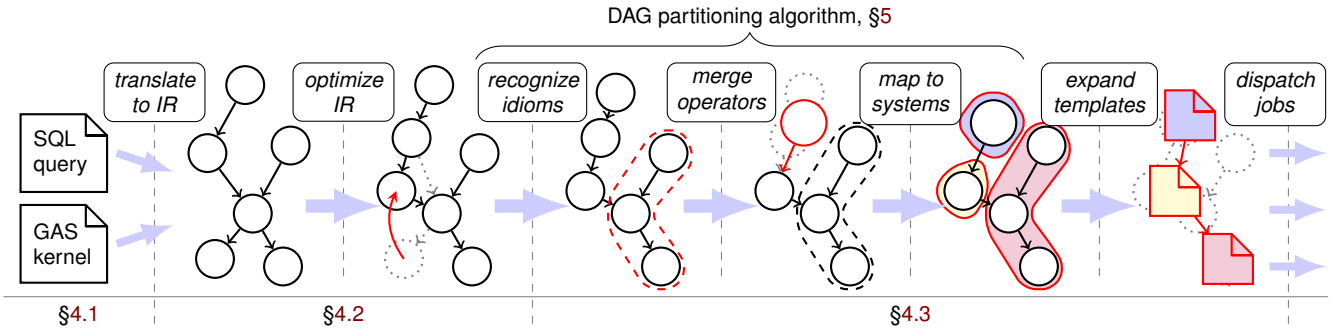


Figure 5: Phases of a Musketeer workflow execution. Dotted, gray operators show previous state; changes are in red.

Hive [40], Shark [41] and SparkSQL optimize relational queries via rewriting rules and FlumeJava [6], Optimus [21] and RoPE [1] apply optimizations to DAGs. Yet, each such optimization must be implemented independently for each front-end framework.

One of the advantages of decoupling front-ends from back-ends is the ability to apply optimizations at the intermediate level, as observed e.g., in the LLVM modular compiler framework [25]. Musketeer can likewise provide benefits to all supported systems (and future ones) by applying optimizations to the intermediate representation.

We currently perform a small set of standard query rewriting optimizations on the IR. Most of these re-order operators – e.g., bringing selective ones closer to the start of the workflow and pushing generative operators to the end.

4.3 Code generation

After Musketeer has translated the workflow to the intermediate representation, it must generate code for execution from the IR. For the time being, we assume that the user explicitly specifies which back-end execution engines to use; in §5, we show how Musketeer can decide automatically.

Musketeer has code templates for specific combinations of operators and back-ends. Conceptually, it instantiates and concatenates these templates to produce executable jobs. In practice, however, optimizations are required to make the performance of the generated code competitive with hand-written baselines. Musketeer uses traditional database optimizations (e.g., *sharing data scans* and *operator merging*), combined with compiler techniques (e.g., *idiom recognition* and *type inference*) to improve upon the naïve approach. In the following, we explain these optimizations with respect to the *max-property-price* Hive workflow example (Listing 1) and the GAS PageRank example (Listing 2).

4.3.1 Idiom recognition

Some back-end execution engines are specialized for a specific type of computation. For example, GraphChi has a vertex-centric computation model and PowerGraph uses the GAS decomposition. Neither system can express computations that do not fit its model. Musketeer must therefore recognize specific computational idioms in the IR to decide if a back-end is suitable for a workflow. *Idiom recognition* is a technique used in parallelizing compilers to detect computational idioms that allow transformations to be applied [36]. We use a similar approach to detect high-level paradigms in Musketeer's IR DAG.

Our prototype detects vertex-oriented graph-processing algorithms in the IR, even if they were originally expressed in a relational front-end (e.g., in Hive instead of the GAS DSL). The idiom is a reverse variant of the way GraphX abstracts graph computation as data-flow operators [15, §3]. Musketeer looks for a combination of the `WHILE` and `JOIN` operators with a `GROUP BY` operator in a particular structure: the body of the `WHILE` loop must contain a `JOIN` operator with two inputs that represent vertices and edges. This `JOIN` operator must be followed by a `GROUP BY` operator that groups data by the vertex column.

This structure maps to the graph computation paradigms as follows: the `JOIN` on the vertex column represents sending messages to neighbors (vertex-centric model), or the "scatter" phase (GAS decomposition); the `GROUP BY` is equivalent to receiving messages, or the "gather" step (GAS); and any other operators in the `WHILE` body are part of the superstep (vertex-centric) or the "apply" step (GAS), updating the state of each vertex.

Other idioms can also be detected: for example, depending on whether the `AGG` operator performs an associative or a non-associative (e.g., subtraction, division) aggregation, different operator implementations are appropriate in different back-ends. We plan to support this in future work.

4.3.2 Merging operators

General-purpose systems such as Spark and Naiad can express complex workflows as a single job. However, more restricted systems only support particular idioms (e.g., PowerGraph, GraphChi) or require multiple jobs to express certain operations (e.g., MapReduce). Each back-end job generated for an execution engine comes with some per-job overhead. Musketeer therefore *merges* operators in order to reduce the number of jobs executed.

```
1 locs =
2   properties.map(c => (c.uid, c.street, c.town))
3 id_price = locs
4   .map(l => (l.uid, (l.street, l.town)))
5   .join(prices)
6   .map((key, (l_rel, r_rel)) => (key, l_rel, r_rel))
7 street_price = id_price
8   .map(ip => ((ip.street, ip.town), ip.price)
9   .reduceByKey((left, right) => Max(left, right))
```

Listing 3: Naïve Spark code for *max-property-price*. Four `map`s are required as data structures must be transformed.

```
1 locs =
2   properties.map(c => (c.uid, (c.street, c.town)))
3 id_price = locs
4   .join(prices)
5   .map((key, (l_rel, r_rel)) =>
6         ((l_rel.street, l_rel.town), r_rel.price))
7 street_price = id_price
8   .reduceByKey((left, right) => Max(left, right))
```

Listing 4: Optimized Spark code for *max-property-price*. Scan sharing and type inference reduce the `map`s to two.

Consider an example: MapReduce-based execution engines only support one group-by-key operation per job [35]. Hence, even a simple workflow like *max-property-price* requires at least two jobs: (*i*) Lines 1–3 in Listing 1 (§4.1.1) result in a job that selects columns from the `properties` relation and joins the result with the `prices` relation using `id` as the key; and (*ii*) lines 4–5 group by a *different* key than the prior join, requiring a second job. By contrast, Listing 3 shows simple generated code for the *max-property-price* workflow in Spark, where only one job is required.

To model these limitations and avoid extra jobs when possible, Musketeer has a set of per-back-end mergeability rules. These indicate whether operators can be merged into one job either (*i*) bidirectionally, (*ii*) unidirectionally, or (*iii*) not at all. If execution engines only support certain idioms, only operator merges corresponding to these idioms are feasible. The operator merge rules are used by the DAG partitioning algorithm (§5) to decide upon job boundaries. Operator merging is necessary for good performance: in §6.5, we show that it reduces workflow makespan by 2–5×.

4.3.3 Sharing data scans

Operator merging allows Musketeer to execute several operators as a single job, and thus eliminates job creation overheads where possible. However, this is not enough to obtain competitive results compared to hand-coded baselines. For example, the first `SELECT` and the `JOIN` operator from the *max-property-price* in Spark get translated into two `map` transformations and a `join` (Listing 3, lines 3–6). The first `map` selects only the columns required by the workflow, while the second `map` establishes a key → ⟨tuple⟩ mapping over which the `join` is going to be conducted.

Even though Spark holds the intermediate RDDs in memory, scanning over the data twice yields a significant performance penalty. Musketeer avoids redundant scans by combining them where supported by the back-end. For example, in the optimized generated Spark code (Listing 4) for the *max-property-price* workflow, the anonymous lambdas from the first two map transformations (Listing 3, lines 4 and 6) are combined into a single one (Listing 4, lines 5–6). As a result, the generated code only scans the data once, selecting the required columns and preparing the relation for the `join` transformation in one go.

4.3.4 Look-ahead and type inference

Many execution engines (e.g., Spark and Naiad) expose a rich API for manipulating different data types. For example, the `SELECT ... GROUP BY` clause in the *max-property-price* workflow (Listing 1, lines 4–5) can be implemented directly in Spark using a `reduceByKey` transformation. However, such API calls often require a specific representation of the input data. In the example, Spark's `reduceByKey` requires the data to be represented as a set of ⟨key, value⟩ tuples. Unfortunately, the preceding `join` transformation outputs the data in a different format (*viz.* ⟨key, ⟨left_relation, right_relation⟩⟩). Hence, the naïve generated code for Spark ends up generating *two* `map` transformations, one to flatten the output of the `join` (Listing 3, line 6), and another to key the relation by a ⟨`town`, `street`⟩ tuple (line 8).

To mitigate this, Musketeer looks ahead and uses type inference to determine the input format of the operators that ingest the current operator's output. With this optimization, the two `map` transformations can be expressed as a single transformation (Listing 4, lines 5–6). In combination with shared scans, look-ahead and type inference enable Musketeer to guarantee that no unnecessary data scans will take place in most cases.

5. DAG partitioning and automatic mapping

To generate back-end jobs, Musketeer partitions the IR DAG into sub-regions, each representing a job. As we explained in §4.3.2, some execution engines constrain the operators that can be combined in a single job.

Our method of partitioning the IR DAG must therefore generalize over different back-end constraints and extend to future systems' properties. Hence, we consider *all* possible partitionings; when this is too expensive, we apply an efficient heuristic based on dynamic programming (§5.1).

Provided that back-ends' relative performance can be predicted with reasonable accuracy, Musketeer can also automatically decide which back-ends to use. The goodness of different options is quantified using a simple cost function that considers information specific to both workflows and back-ends (§5.2).

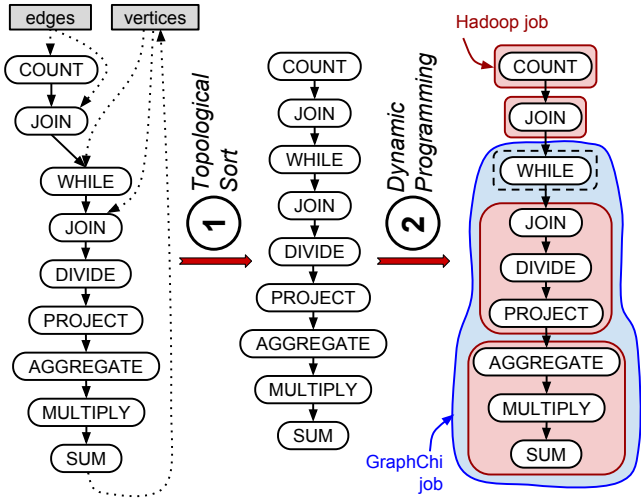


Figure 6: The dynamic partitioning heuristic takes an IR DAG, *(1)* transforms it to a linear order, and *(2)* computes job boundaries via dynamic programming. On the right, we show several possible partitions and system mappings.

5.1 DAG partitioning

There are many ways of breaking the IR DAG into partitions (≡ back-end jobs). Musketeer uses a simple *cost function* to compare different partitioning options. The cost of any partition containing non-mergeable operators is infinite; otherwise it is finite and depends on the back-ends (see §5.2).

With a known optimal number of jobs, k, partitioning the DAG is an instance of the *k-way graph partitioning* problem [22]. Unfortunately, k-way graph partitioning is NP-hard [13]: the best solution is guaranteed to be found only by exploring all k-way partitions. Moreover, the optimal number of jobs into which to partition the DAG is unknown. Hence, Musketeer must solve k-way graph partitioning for all $k \leq N$, where N is the number of operators in the DAG.

Where possible, Musketeer uses an exhaustive search to find the cheapest partitioning (in practice, up to about 18 operators). It switches to a dynamic programming heuristic for larger, more complex workflows.

5.1.1 Exhaustive search

The exhaustive search explores all possible graph partitionings. It first considers the cost of running each operator in isolation. Next, it looks at all merge opportunities, and finally, it recursively generates all valid (finite-cost) partitions. The algorithm is guaranteed to find the optimal solution with respect to the cost function. However, it requires exponential time in the number of operators.

5.1.2 Dynamic heuristic

Some industry workflows consist of large DAGs containing up to hundreds of operators [6, §6.2]. To support these workflows, we use a dynamic heuristic. Its execution time scales linearly with the number of operators, and it obtains good solutions in practice. The dynamic heuristic explores only a subset of the possible partitions by focusing on a single linear ordering of operators. In Figure 6, we illustrate the algorithm using the IR DAG for PageRank (Listing 2).

First, Musketeer topologically sorts the DAG to produce a linear ordering. This ordering maintains operator precedence – i.e., an operator does not appear in the linear ordering before any of its ancestors. Second, Musketeer finds the optimal partitioning of the linear ordering using dynamic programming. The dynamic programming algorithm uses the cost function, $c_s(o_1, o_2, \ldots, o_j)$, which estimates the cost of running the operators $o_1, o_2, \ldots, o_j$ in a single job. It then computes the matrix $C[n][m]$, which stores the minimum cost of running the first n operators in exactly m jobs:

$$C[n][m] = \min_{k<n} \left(C[k][m-1] + \min_{s} \left(c_s(o_{k+1} \ldots o_n) \right) \right)$$

In other words, we determine the best combination that runs a k-element prefix of operators in $m-1$ jobs and the remainder in a single job. This approach finds a good solution because it considers all partitions of the linear ordering. The cost function guides it to merge as many operators as possible within each individual job.

The dynamic heuristic can miss out on opportunities to merge operators due to the linear ordering breaking operator adjacencies. We discuss this further and show an example in §8; in practice, we found the dynamic heuristic to work well.

5.2 Automatic system mapping

A simple extension of the DAG partitioning algorithm allows Musketeer to automatically choose back-end execution engine mappings. To achieve this, we use Musketeer's cost function and run the DAG partitioning algorithm for *all* back-ends. We then pick the best k-way partitioning.

The cost function scores the performance of a particular combination of operators, input data and execution engine. The score is based on three high-level components:

1. **Data volume.** Each operator has bounds on its output size based on its behavior (e.g., whether it is generative or selective). These bounds are applied to the run-time input data size to predict intermediate and output sizes.
2. **Operator performance.** In a one-off calibration, Musketeer measures each operator in each back-end and records the rate at which it processes data.
3. **Workflow history.** Musketeer collects information about each job it runs (e.g., runtime and input/output sizes), and uses this information to refine the scores for subsequent runs of the same workflow.

The operator performance calibration only requires modest one-off profiling for a deployed cluster. It supplies Musketeer with the four rates listed in Table 1. `PULL` and `PUSH` quantify read and write HDFS throughput at the start and end of a job. We measure them using a "no-op" operator. `LOAD`,

Parameter	Description
PULL	Rate of data ingest from HDFS.
LOAD	Rate of loading or transforming data.
PROCESS	Rate of processing operator on in-memory data.
PUSH	Rate of writing output to HDFS.

Table 1: Rate parameters used by Musketeer's cost function.

System	Modification
Hadoop	Tuned configuration to best practices.
Spark	Tuned configuration to best practices.
GraphChi	Added HDFS connector for I/O.
Naiad	Added support for parallel I/O and HDFS.

Table 2: Modifications made to systems deployed.

by contrast, corresponds to back-end-specific data loading or transformation steps (e.g., partitioning the input in PowerGraph). Finally, PROCESS approximates the rate at which the operator's computation proceeds. In some systems, we measure this directly, while in others, we subtract the estimated duration of the ingest (from PULL) and output (from PUSH) stages from the overall runtime to obtain PROCESS.

This information lets us estimate the benefit of shared scans: we pay the cost of PULL, LOAD and PUSH just once (rather than once per-operator) and combine those with the costs of PROCESS for all the operators.

These rate parameters enable generic cost estimates, but we can achieve more accurate scoring by using workflow-specific historical information. When a workflow first executes, no such information is available. Musketeer thus applies conservative data size bounds and only merges selective operators and generative operators with small output bounds. As a result, more jobs may be generated on the first execution – e.g., due to JOIN operators, which have unknown data size bounds. On subsequent executions, Musketeer tightens the bounds using historical information, which may unlock additional merge opportunities.

6. Evaluation

Musketeer's goal is to improve flexibility and performance of data processing workflows by dynamically mapping from front-end frameworks to back-end execution engines. In this section, we show that Musketeer meets these goals:

1. **Legacy workflow speedup:** Musketeer reduces legacy workflows' makespan by up to 2× by mapping them to a different back-end execution system (§6.2).
2. **Flexible combinations of back-ends:** by exploring combinations of multiple execution systems for a workflow, Musketeer finds combinations that perform well (§6.3).
3. **Increased portability:** compared to time-consuming, hand-tuned implementations for specific back-ends, Musketeer's automatically generated code has low overhead, yet offers superior portability (§6.4, §6.5).
4. **Promising automatic system mapping:** our automated mapping prototype makes good choices based on simple parameters characterizing execution engines (§6.6, §6.7).

We implemented seven real-world workflows to evaluate Musketeer: three batch workflows, three iterative workflows and a hybrid one. The batch workflows are (*i*) TPC-H query 17, (*ii*) *top-shopper*, which identifies an online shop's top spenders, and (*iii*) the Netflix movie recommendation algorithm. The iterative ones are (*i*) PageRank, (*ii*) single-source shortest path (SSSP), and (*iii*) *k*-means clustering; the hybrid workflow is PageRank with a batch pre-processing stage.

6.1 Setup

Most of our experiments run on a 100-node cluster of m1.xlarge instances on Amazon EC2. However, we also run some experiments on a local, dedicated seven-machine cluster with low variance in performance. We deployed all systems supported by Musketeer[6] on these clusters. We use a shared HDFS as the storage layer; this makes sense as HDFS is already supported by Hadoop, Spark and PowerGraph. In order to establish a level playing field for our experiments, we tuned and modified some systems (see Table 2).

Metrics. As in §2, the **makespan** of a workflow refers to its total execution time, measured from its launch to the final result appearing in HDFS. This includes time to load inputs from HDFS, pre-process or transform them (e.g. in PowerGraph and GraphChi) and write the outputs back. As a result, the numbers we present are not directly comparable to those in some other papers, which measure the actual computation time only.

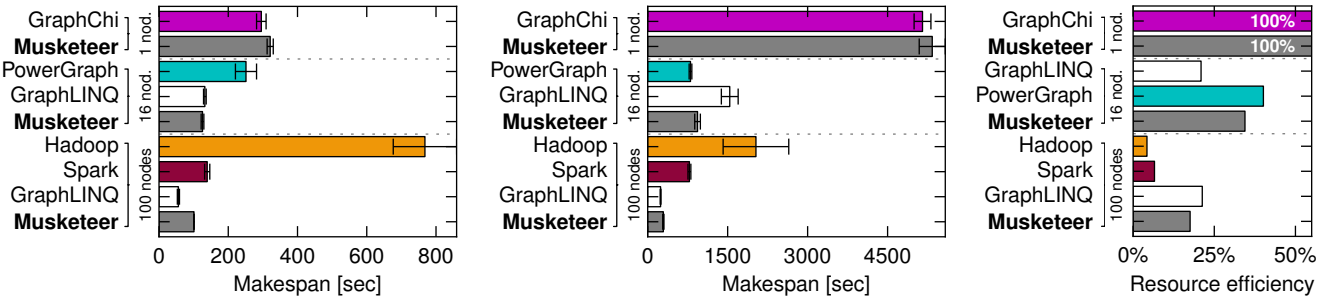
Resource efficiency, on the other hand, is a measure of the efficiency loss incurred due to scaling out over multiple machines. We compute it by normalizing a workflow's fastest single-node execution (assumed to be maximally resource-efficient) to its aggregate execution time over all nodes in a distributed system. For example, a workflow that runs for 30s on all 100 nodes of the EC2 cluster has an aggregate execution time of 3,000s. If the best single-node system completes the same workflow in 2,000s, the resource efficiency of the distributed execution is 66%.

6.2 Dynamic mapping to back-end execution engines

We consider both batch and iterative graph processing workflows to investigate the benefits of Musketeer's ability to dynamically map workflows to back-ends. This mirrors our motivational experiments in §2.

Batch workflows. To illustrate the flexibility offered by Musketeer, we run query 17 from the TPC-H business decision benchmark using the HiveQL and Lindi front-ends. Figure 7 shows the resulting makespan as the data size increases from 7.5 GB (scale factor 10) to 75 GB (factor 100).

[6] Hadoop 2.0.0-mr1-chd4.5.0, Spark 0.9, PowerGraph 2.2, GraphChi 0.2, Naiad 0.2 and Metis commit e5b04e2.



(a) Orkut (3.0M vertices, 117M edges). **(b)** Twitter (42M vertices, 1.4B edges). **(c)** Resource efficiency, Twitter.

Figure 8: Musketeer's performs close to the best-in-class system for five iterations of PageRank on 100, 16 and 1 nodes. In *(a)* and *(b)*, the *x*-axis is makespan (less is better); in *(c)*, it is resource efficiency (more is better). Error bars are $\pm\sigma$ over 5 runs.

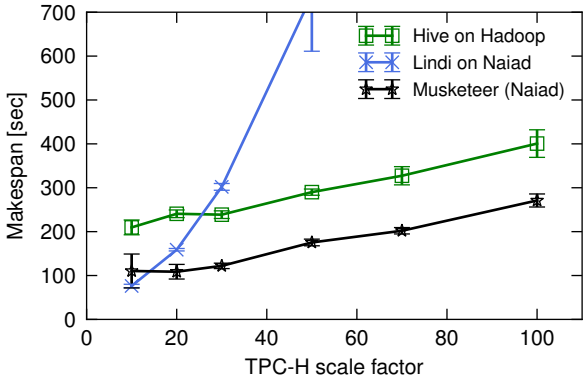


Figure 7: Musketeer reduces the makespan of TPC-H query 17 on EC2 compared to Hive and Lindi on native back-ends. Less is better; error bars show min/max of three runs.

When running the Hive workflow directly using its native Hadoop back-end, the makespan ranges from 200–400s.

Musketeer, however, can map the Hive workflow specification to different back-ends. In this case, mapping it to Naiad reduces the makespan by 2×. This is not surprising: Hive must run the workflow as three Hadoop jobs due to the restrictive MapReduce paradigm, while Naiad can run the entire workflow in one job.

However, a user might also specify the workflow using the Lindi front-end and target Naiad directly. When using Lindi however, the query scales less well than using Hive and Hadoop, despite running in Naiad. This result comes because Lindi's high-level `GROUP BY` operator is non-associative, meaning that data must be collected on a single machine before the operator can be applied. Musketeer supplies an improved `GROUP BY` operator implemented against Naiad's low-level vertex API. Consequently, its generated Naiad code scales far better than the Lindi version (up to 9× at scale 100). The Naiad developers may of course improve Lindi's `GROUP BY` in the future, but this example illustrates that Musketeer's decoupling can improve performance even for a front-end's native execution engine by generating improved code.

Iterative workflows. While batch workflows can be expressed using SQL-like front-end frameworks such as Hive and Lindi, iterative graph processing workflows are typically expressed differently (see §4.1.2). To evaluate Musketeer's benefit for graph computations, we implemented PageRank using our GAS DSL front-end (Listing 2, §4.1.2). We run this workflow on the two social network graphs we evaluated PageRank on in §2 (Orkut and Twitter).

Figure 8 compares the makespan of a five iterations of PageRank using Musketeer-generated jobs to hand-written baselines for general-purpose systems (Hadoop, Spark), an implementation using Naiad's GraphLINQ front-end and special-purpose graph processing systems (PowerGraph, GraphChi). Different systems achieve their best performance at different scales, and we only show the best result for each system. The only exception to this is GraphLINQ on Naiad, which is competitive at both 16 and 100 nodes. At each scale, Musketeer's best mapping is almost as good as the best-in-class baseline. On one node, Musketeer does best when mapping to GraphChi, while a mapping to Naiad (Orkut) or PowerGraph (Twitter) is best at 16 nodes, and a mapping to Naiad is always best at 100 nodes.

Figure 8c shows the resource efficiency for the same configurations for PageRank on the Twitter graph. Musketeer achieves resource efficiencies close to the best stand-alone implementations at all three scales.

This demonstrates that Musketeer's dynamic mapping approach adds flexibility for batch and iterative computations.

6.3 Combining back-end execution engines

In addition to mapping an *entire* workflow to different back-ends, Musketeer can also map *parts* of a workflow to different systems. We find that this ability to explore many different combinations of systems can yield useful (and sometimes surprising) results.

We use the hybrid *cross-community PageRank* workflow to demonstrate this. This workflow yields the relative popu-

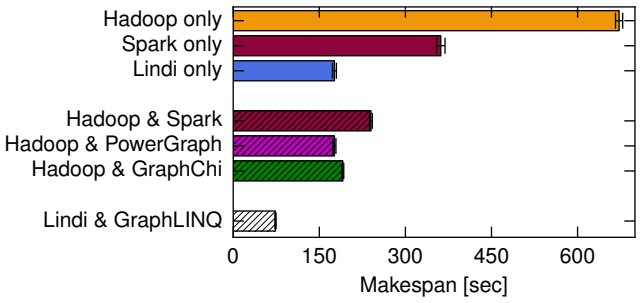


Figure 9: A cross-community PageRank workflow is accelerated by combined back-ends. Local cluster, all jobs apart from the "Lindi & GraphLINQ" combination were generated by Musketeer. Error bars show the min/max of 3 runs.

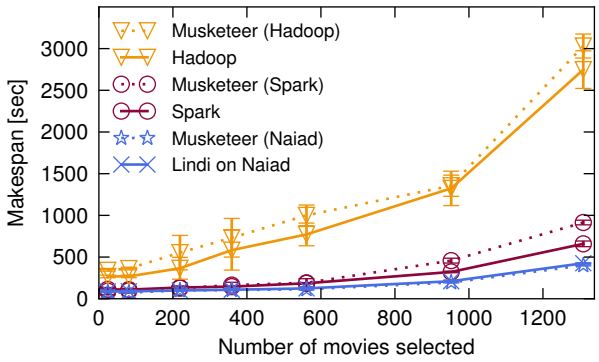


Figure 10: Makespan of the NetFlix movie recommendation workflow on the EC2 cluster. Error bars: $\pm\sigma$ over five runs.

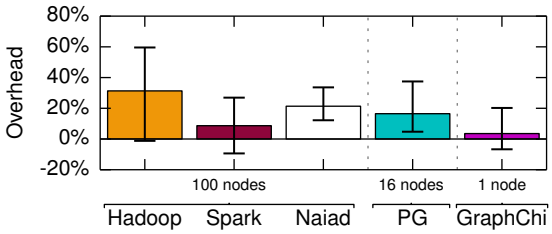


Figure 11: Generated code overhead for PageRank on the Twitter graph (PG ≡ PowerGraph). Error bars show σ over five runs; negative overheads are due to variance on EC2.

larity of the users present in both of two web communities. It involves a batch computation followed by an iterative computation: first, the edge sets of two communities (e.g., all LiveJournal and WordPress users) are intersected, and subsequently, the PageRank of all links present in both communities is computed.

Figure 9 shows the makespan of cross-community PageRank for different combinations of systems, explored using Musketeer.[7] The inputs are the LiveJournal graph (4.8M nodes and 68M edges) and a synthetically generated web community graph (5.8M nodes and 82M edges). Out of the three single-system executions, the workflow runs fastest in Lindi at 153s. However, the makespan is comparable when Musketeer combines Hadoop with a special-purpose graph processing system (e.g., PowerGraph), even though these systems use fewer machines. This happens because general-purpose systems (like Hadoop) work well for the batch phase of the workflow, but do not run the iterative PageRank as fast as specialized systems. However, a combination of Lindi and GraphLINQ, which both use Naiad as their back-end execution engine, does even better. This comes as this combination avoids the extra I/O to move intermediate data across system boundaries. Musketeer currently does not fully automatically generate the low-level Naiad code to combine Lindi and GraphLINQ; we will support this in future work.

Musketeer's ability to flexibly partition a workflow makes it easy to explore different combinations of systems.

6.4 Overhead over hand-tuned, non-portable jobs

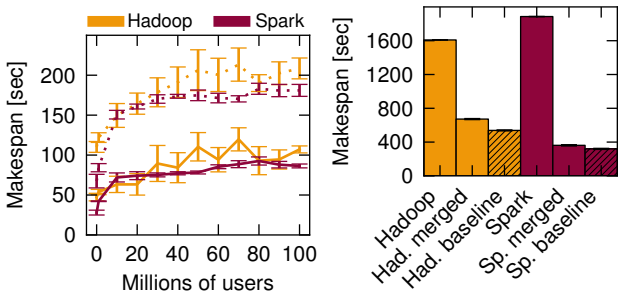
For Musketeer to be attractive to some users, its generated code must add minimal overhead over an optimized hand-written implementation. In the following, we show that the overhead over an optimized baseline does not exceed 30% and is usually around 5–20%.

Batch processing. We measure the NetFlix movie recommendation workflow [2]. This workflow highlights any overheads: it contains a large number of operators (13) and is very data-intensive, with up to 600 GB of intermediate data generated. The workflow takes two inputs: a 100 million-row movie ratings table (2.5GB) and a 17,000-row movie list (0.5MB). The algorithm computes movie recommendations for all users, and finally outputs the top recommended movie for each user. We control the amount of data processed by the algorithm by varying the number of movies used for the prediction. Figure 10 compares Musketeer-generated code for the NetFlix workflow to hand-optimized baselines for the three general-purpose systems that support it (Hadoop, Spark and Lindi on Naiad). We extensively tuned each of the baselines to deliver good performance for the given system, taking advantage of system-specific optimizations available.

For all three systems, the overhead added by Musketeer's generated code is low: it is virtually non-existent for Naiad and remains under 30% for Spark and Hadoop even as the input grows. The remaining overhead for Spark is primarily due to the simplicity of our type-inference algorithm, which can cause the Musketeer-generated code to make an extra pass over the data.

Graph processing. We also measure Musketeer's overhead for the iterative PageRank workflow. Figure 11 shows the overhead of Musketeer-generated jobs over hand-written baselines for the back-ends compatible with the PageRank workflow. The average overhead remains below 30% in all cases. Variability in overhead (and improvements over the

[7] The 100-node EC2 cluster had similar results, albeit at increased variance.



(a) *top-shopper* workflow running on the EC2 cluster.

(b) Hybrid cross-community PageRank on the local cluster.

Figure 12: Operator merging (§4.3.2) helps bring generated code performance close to hand-written baselines.

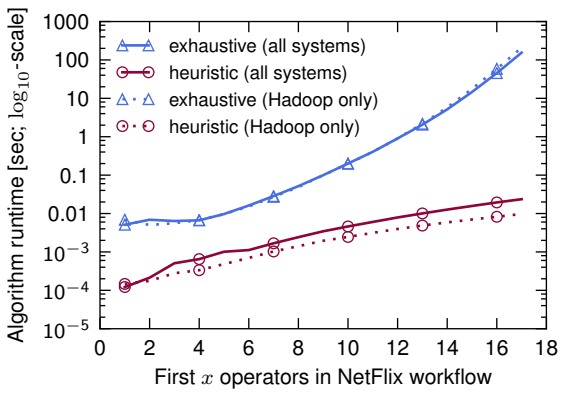


Figure 13: Runtime of Musketeer's DAG partitioning algorithms when considering the first x operators of an extended version of the NetFlix workflow (N.B.: $\log_{10}$-scale y-axis).

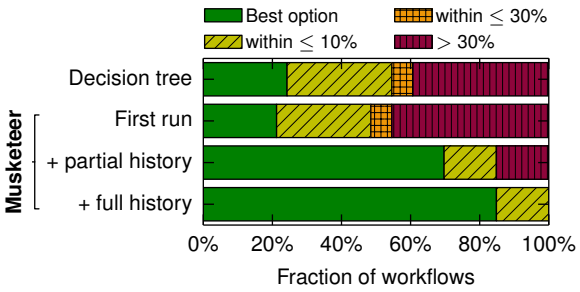


Figure 14: Makespan overhead of Musketeer's automated choice compared to the best option. Workflow history helps, and our cost function outperforms a simple decision tree.

baseline) are due to performance variance on EC2. Further optimizations of the code generation are possible. Most such optimizations benefit *all* code Musketeer generates for a particular back-end.

In conclusion, Musketeer generates code that performs nearly as well as hand-written baselines. Combined with the improved portability and the ability to dynamically explore multiple execution engines, we believe that this makes for a compelling case.

6.5 Impact of operator merging and shared scans

One key technique that Musketeer uses to reduce overhead is operator merging (§4.3.2). We measure its impact on workflow makespan using a simple micro-benchmark: the *top-shopper* workflow. This benchmark finds the largest spenders in a certain geographic region by first filtering a set of purchases by region, then aggregating their values by user ID and finally selecting all users above a threshold. The workflow consists of three operators that can be merged into a single job, and indeed a single scan of the data. Figure 12a shows top-shopper's makespan for varying data size with operator merging turned off and on. In Figure 12b, we show that cross-community PageRank sees the same benefit.

These results illustrate that the impact of operator merging can be significant: we observe a one-off reduction in makespan of ≈25–50s due to avoiding per-job overheads, along with an additional 5–10% linear benefit per 10M users attributable to the use of shared scans.

6.6 DAG partitioning runtime

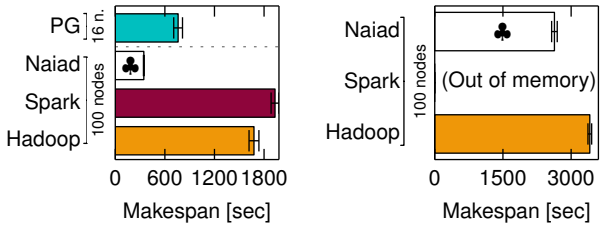
Next, we focus on the DAG partitioning algorithm (§5). We measure the time it takes the exhaustive search and dynamic heuristic algorithms to partition the operator DAG. Ideally, they should not noticeably affect the total runtime of the workflow.

Figure 13 compares the runtimes of the two algorithms as the number of operators in a workflow increases. In the experiment, we run subsets of an extended version of the NetFlix workflow with a total of 18 operators. This workload affords many operator merging opportunities, thus making a good test case for the DAG partitioning algorithms. Up to 13 operators, the exhaustive search runs in under a second, but its runtime grows exponentially beyond 13 operators. While it guarantees that the optimal partitioning subject to the cost function is found, the delay soon becomes impractical. The dynamic programming heuristic, however, runs in under 10ms even at 18 operators and scales gracefully.

6.7 Automated mapping performance

Musketeer can be used to manually map jobs to back-end execution engines, but we believe that the framework choice should be automated. We first investigate the quality of Musketeer's automated mapping decisions (§5.2) using the workflows discussed so far, and then test its performance on two additional workflows.

We tested Musketeer's automated choices using the six workflows described before in 33 different configurations by varying the input data size. For each decision, we compare (*i*) Musketeer's choice on the first run (with no workflow-specific history), (*ii*) its choice with incrementally acquired partial history, and (*iii*) the choice it makes when it has a full history of the per-operator intermediate data sizes. We also



(a) SSSP. **(b)** *k*-means clustering.

Figure 15: Makespan of SSSP and *k*-means on the EC2 cluster (5 iterations). A club (♣) indicates Musketeer's choice.

compare the choices to those that emerge from a decision tree that we developed. The decision tree considers different back-ends' features and known characteristics. We consider a choice that achieves a makespan within 10% of the best option to be "good", and one within 30% as "reasonable". Figure 14 shows the results: without any knowledge, Musketeer chooses good or optimal back-ends in about 50% of the cases. When partial workflow history is available, over 80% of its choices are good ones. If each workflow is initially executed operator-by-operator for profiling, Musketeer always makes good or optimal choices. By contrast, using the decision tree yields many poor choices. This is due to its inflexible decision thresholds and its inability to consider the benefits of operator merging and shared scans.

We also test the automatic mapping on two new workflows: single-source shortest path (SSSP) and *k*-means clustering. SSSP can be expressed in vertex-centric systems, while *k*-means cannot. Figure 15 shows the workflows' makespan for different back-ends and Musketeer's automated choice. The input for SSSP was the Twitter graph extended with costs, and we used 100M random points for *k*-means (100 clusters, two dimensions).[8] Even with our simple proof-of-concept cost function and a small training set, Musketeer in both cases correctly identifies the appropriate back-end (Naiad).

7. Practical experience with Musketeer

System integration complexity. We found the effort it take to integrate a front-end framework or a back-end execution engine with Musketeer to be reasonable. A graduate student typically takes a few days to add full support for another back-end execution engine. Our experiences with front-end frameworks were similar, although the effort required varies depending on their expressivity. Additional time and careful profiling is required to fully optimize the performance of generated code, but such improvements only need to be made once in order to benefit all Musketeer users.

[8] Our *k*-means uses the `CROSS JOIN` operator, which is inefficient. By replacing it, we could reduce the makespan and address Spark's OOM condition. However, we are only interested in the automated mapping here.

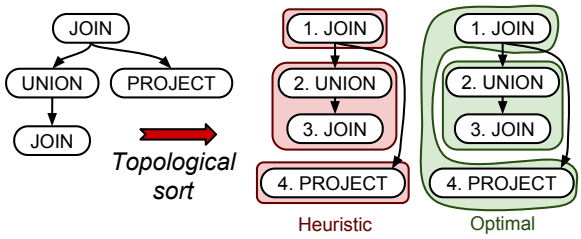


Figure 16: The dynamic heuristic does not return the minimum-cost partitioning for this workflow: it misses the opportunity to merge `JOIN` with `PROJECT`.

Benefit over hand-coded jobs. To anecdotally compare the performance of Musketeer's generated code to a baseline written by an average programmer, we asked eight CS undergraduate students to implement and optimize the simple `JOIN` workflow from §2.1 for a given input data set using Hadoop. The best student implementation took 608s, compared to the Musketeer-generated job at 223s. While not a rigorous evaluation, we take this as an indication that using Musketeer offers benefit for non-expert programmers.

8. Limitations and future work

In the following, we highlight some of the current limitations in Musketeer and how they can be addressed in future work.

Front-ends (§4.1). In the future, we see Musketeer offering better support for user-defined functions in front-ends. Many vertex-centric systems, for example, allow the user to specify arbitrary per-vertex code in Java (Giraph) or C++ (GraphChi). This increases flexibility, but restricts the level of optimization that Musketeer can offer. It either limits its choice to back-ends that can directly execute the user-provided code, or requires use of inefficient foreign-function interfaces. In the future, techniques that perform query synthesis on arbitrary user code [9, 20] might help here.

Idiom recognition (§4.3.1). As with many idiom recognition techniques, our approach is sound, but not complete. Musketeer may occasionally fail to detect graph workloads, consequently generating less efficient code. For example, a *triangle counting* workflow may encounter this problem: the user may represent it as a workflow that joins the edges twice and then filters out the triangles. In the latter case, Musketeer fails to detect the opportunity of running the computation in a graph-oriented execution engine. A "reverse loop unrolling" heuristic that detects when multiple operators take the same input and produce the same output (or a closure thereof) can partly solve this.

Dynamic DAG partitioning heuristic (§5.1.2). The dynamic programming heuristic returns the optimal *k*-way partitioning of a given linear order. However, it may miss fruitful merging opportunities, since it only explores a single linear ordering of operators. In Figure 16, we show an example of a workflow for which the dynamic heuristic

Data processing system	*Paradigm*	*Environment*	*In-memory*	*Distributed I/O*	*Pre-processing*	*Default sharding*	*Work unit size*	*Fault tolerance*	*Language*
MapReduce [10], **Hadoop**	MapReduce	cluster	–	✓	–	user-def.	large	✓	C++/Java
Spark [43]	transformations	cluster	✓	✓	–	uniform	med.	✓	Scala
Dryad [19]	static data-flow	cluster	–	✓	–	user-def.	large	✓	C#
Naiad [34]	timely data-flow	cluster	✓	(✓)	(✓)	user-def.	small	(✓)	C#
Pregel [26], Giraph **PowerGraph** [14]	vertex-centric	cluster	–, ✓	✓	–	uniform, power-law	med.	(✓)	C++
CIEL [32]	dynamic data-flow	cluster	(✓)	✓	–	user-def.	med.	✓	various
Serial C code	none/serial	machine	–	–	–	–	small	–	C
Phoenix [37], **Metis** [27]	MapReduce	machine	✓	–	–	user-def.	small	–	C++
GraphChi [24]	vertex-centric	machine	✓	–	✓	–	short	–	C++
X-Stream [38]	edge-centric	machine	✓	–	–	–	med.	–	C++

Table 3: A selection of contemporary data processing systems with their features and properties. Systems supported by Musketeer are highlighted in **bold**. (✓) indicates that the system can be extended to support this feature.

does not achieve optimality. In the MapReduce paradigm, it makes sense to run the top `JOIN` in the same job as the `PROJECT`, but the linear ordering based on depth-first exploration breaks this merge opportunity. This limitation does not affect general-purpose back-ends (e.g., Naiad and Spark), which are able to merge any sub-region of operators. However, merging opportunities are occasionally be missed for systems with restricted expressivity, such as Hadoop and Metis. A simple solution generates multiple linear orderings and runs the heuristic for each of them.

9. Related Work

Musketeer is, to our knowledge, the first "big data" workflow manager that decouples front-ends from back-ends and supports multiple execution engines (Table 3). Nonetheless, there is considerable related work:

Workflow managers. Pig [35] and Hive [40] are widely used workflow managers on top of Hadoop that present a SQL-like interface to the user. Shark [41] replaces Hive's physical plan generator to use Spark RDDs and supports fast interactive in-memory queries. SCOPE [4] and Tenzing [7] make the relationship to SQL more explicit, with Tenzing providing an almost complete SQL implementation on top of MapReduce. The semantics of these tools, however, are heavily influenced by the execution engine to which they compile (e.g., Pig relies on `COGROUP` clauses to delineate MapReduce jobs).

Dynamic paradigm choice. FlumeJava [5] defers execution of operations on Java parallel collections until runtime. The implementation of operations is abstracted away from the user and can range from a local iterator to a MapReduce job, depending on data size. QoX [39] combines databases and data processing engines by separating logical operations and physical implementation, but, unlike Musketeer, is limited to ETL workflows.

Automatic system tuning. A number of efforts have looked at automatically tuning the *configuration* of data processing systems. Starfish [17] automatically infers Hadoop configuration variables, while Jockey [12] and Quasar [11] automatically determine the resources to allocate to a workflow in order to meet its deadline or QoS requirements. Musketeer could be extended to perform these tasks as well [16, 29].

10. Conclusion

Musketeer decouples front-end frameworks from back-end execution engines. As a result, users benefit from increased flexibility: workflows can be written once and mapped to many systems, different systems can be combined within a workflow and existing workflows seamlessly ported to new execution engines. Musketeer enables compelling performance gains and its generated code performs almost as well as unportable, hand-optimized baseline implementations. Musketeer is open-source, and available from:

`http://www.cl.cam.ac.uk/netos/musketeer/`

Acknowledgments

We would like to thank Derek G. Murray, Frank McSherry, John Wilkes, Kim Keeton, Robert N. M. Watson, Jon Crowcroft, Tim Harris and our anonymous reviewers for their valuable feedback. Thanks also go to Anne-Marie Kermarrec, our shepherd. Natacha Crooks and Ionel Gog were partly supported by Google Europe Fellowships; parts of this work were supported by the EPSRC INTERNET Project EP/H040536/1, the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the DARPA or the Department of Defense.

References

[1] AGARWAL, S., KANDULA, S., BRUNO, N., WU, M.-C., STOICA, I., AND ZHOU, J. Re-optimizing data-parallel computing. In *Proceedings of NSDI* (2012).

[2] BELL, R. M., KOREN, Y., AND VOLINSKY, C. The BellKor solution to the Netflix prize. Tech. rep., AT&T Bell Labs, 2008.

[3] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND TYSON, C. Pregelix: Big(ger) Graph Analytics on A Dataflow Engine. *Proceedings of the VLDB Endowment 8*, 2 (2015), 161–172.

[4] CHAIKEN, R., JENKINS, B., LARSON, P., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1265–1276.

[5] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: easy, efficient data-parallel pipelines. In *ACM SIGPLAN Notices* (2010), vol. 45, pp. 363–375.

[6] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: Easy, Efficient Data-parallel Pipelines. In *Proceedings of PLDI* (2010), pp. 363–375.

[7] CHATTOPADHYAY, B., LIN, L., LIU, W., MITTAL, S., ARAGONDA, P., LYCHAGINA, V., KWON, Y., AND WONG, M. Tenzing: a SQL implementation on the MapReduce framework. *Proceedings of the VLDB Endowment 4*, 12 (2011), 1318–1327.

[8] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1802–1813.

[9] CHEUNG, A., SOLAR-LEZAMA, A., AND MADDEN, S. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of PLDI* (2013), pp. 3–14.

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: a flexible data processing tool. *Communications of the ACM 53*, 1 (2010), 72–77.

[11] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of ASPLOS* (2014), pp. 127–144.

[12] FERGUSON, A. D., BODIK, P., KANDULA, S., BOUTIN, E., AND FONSECA, R. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of EuroSys* (2012), pp. 99–112.

[13] GAREY, M. R., JOHNSON, D. S., AND STOCKMEYER, L. Some simplified NP-complete graph problems. *Theoretical Computer Science 1*, 3 (1976), 237–267.

[14] GONZALEZ, J., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of OSDI* (2012).

[15] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of OSDI* (2014), pp. 599–613.

[16] HERODOTOU, H., AND BABU, S. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proceedings of the VLDB Endowment 4*, 11 (2011), 1111–1122.

[17] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of CIDR* (2011), pp. 261–272.

[18] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of ASPLOS* (2012), pp. 349–362.

[19] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys* (2007), pp. 59–72.

[20] IU, M.-Y., AND ZWAENEPOEL, W. HadoopToSQL: A MapReduce Query Optimizer. In *Proceedings of EuroSys* (2010), pp. 251–264.

[21] KE, Q., ISARD, M., AND YU, Y. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of EuroSys* (2013), pp. 15–28.

[22] KERNIGHAN, B. W., AND LIN, S. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal 49*, 2 (1970), 291–307.

[23] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of EuroSys* (2013), pp. 169–182.

[24] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of OSDI* (2012), pp. 31–46.

[25] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO* (Mar 2004).

[26] MALEWICZ, G., AUSTERN, M., BIK, A., DEHNERT, J., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of SIGMOD* (2010), pp. 135–146.

[27] MAO, Y., MORRIS, R., AND KAASHOEK, F. Optimizing MapReduce for multicore architectures. Tech. Rep. MIT-CSAIL-TR-2010-020, MIT Computer Science and Artificial Intelligence Laboratory, May 2010.

[28] MCSHERRY, F. GraphLINQ: A graph library for Naiad, May 2014. Big Data at SVC blog, http://goo.gl/Q0F9gw; accessed 03/10/2014.

[29] MORTON, K., BALAZINSKA, M., AND GROSSMAN, D. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of SIGMOD* (2010), pp. 507–518.

[30] MÜLLER, S. C., ALONSO, G., AMARA, A., AND CSILLAGHY, A. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In *Proceedings of OSDI* (2014), pp. 645–659.

[31] MURRAY, D. Building new frameworks on Naiad, Apr. 2014. Big Data at SVC blog, http://goo.gl/qqvAQv; accessed 03/10/2014.

[32] MURRAY, D., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: a universal

execution engine for distributed data-flow computing. In *Proceedings of NSDI* (2011), pp. 113–126.

[33] MURRAY, D. G. *A distributed execution engine supporting data-dependent control flow*. PhD thesis, University of Cambridge, 2011.

[34] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of SOSP* (2013), pp. 439–455.

[35] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of SIGMOD* (2008), pp. 1099–1110.

[36] POTTENGER, B., AND EIGENMANN, R. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proceedings of SC* (1995), pp. 444–448.

[37] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for multi-core and multiprocessor systems. In *Procedings of HPCA* (2007), pp. 13–24.

[38] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of SOSP* (2013), pp. 472–488.

[39] SIMITSIS, A., WILKINSON, K., CASTELLANOS, M., AND DAYAL, U. Optimizing analytic data flows for multiple execution engines. In *Proceedings of SIGMOD* (2012), pp. 829–840.

[40] THUSOO, A., SARMA, J., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive – A Warehousing Solution over a Map-Reduce Framework. *Proceedings of the VLDB Endowment 2*, 2 (2009), 1626–1629.

[41] XIN, R. S., ROSEN, J., ZAHARIA, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Shark: SQL and Rich Analytics at Scale. In *Proceedings of SIGMOD* (2013), pp. 13–24.

[42] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P., AND CURREY, J. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of OSDI* (2008).

[43] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of NSDI* (2012), pp. 15–28.