Apache Nemo: A Framework for Optimizing Distributed Data Processing

WON WOOK SONG, YOUNGSEOK YANG, and JEONGYOON EO, Seoul National University
JANGHO SEO, Naver Corporation
JOO YEON KIM, Samsung Electronics
SANHA LEE, Naver Corporation
GYEWON LEE, TAEGEON UM, HAEYOON CHO, and BYUNG-GON CHUN,
Seoul National University

Optimizing scheduling and communication of distributed data processing for resource and data characteristics is crucial for achieving high performance. Existing approaches to such optimizations largely fall into two categories. First, distributed runtimes provide low-level policy interfaces to apply the optimizations, but do not ensure the maintenance of correct application semantics and thus often require significant effort to use. Second, policy interfaces that extend a high-level application programming model ensure correctness, but do not provide sufficient fine control.

We describe Apache Nemo, an optimization framework for distributed dataflow processing that provides fine control for high performance and also ensures correctness for ease of use. We combine several techniques to achieve this, including an intermediate representation of dataflow, compiler optimization passes, and runtime extensions. Our evaluation results show that Nemo enables composable and reusable optimizations that bring performance improvements on par with existing specialized runtimes tailored for a specific deployment scenario. Apache Nemo is open-sourced at https://nemo.apache.org as an Apache incubator project.

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Software and its engineering → Distributed systems organizing principles**; • **Information systems → Data management systems;**

Additional Key Words and Phrases: Data processing, distributed systems

This manuscript is an extension of the conference version appearing in the 2019 USENIX Annual Technical Conference (USENIX ATC'19) [53]. We present a more detailed description of the Nemo architecture, design, and implementation with concrete examples to better illustrate the contents. We supplement new subsections to provide the details of the different components of the execution runtime that enrich the explanation for the underlying system execution. We also provide explanations on how Apache Nemo supports external libraries to compile the applications and to use runtime extensions, such as Apache Crail [41] along with its evaluation.
This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00221, Development of a Unified High Performance Stack for Diverse Big Data Analytics) and BK21 FOUR Intelligence Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea (NRF) (4199990214639).
Authors' addresses: W. W. Song, Y. Yang, J. Eo, G. Lee, T. Um, H. Cho, and B.-G. Chun (corresponding author), Computer Science and Engineering Department, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, 08826, Rep. of Korea; emails: {wsong0512, johnyangk, jeongyoon.eo, gyewonlee, taegeonum, hy.cho, bgchun}@snu.ac.kr; J. Y. Kim, Samsung Electronics, 56 Seongchon-gil, Seocho-gu, Seoul, 06765, Rep. of Korea; email: jykim88@gmail.com; J. Seo and S. Lee, Naver Corporation, 6 Buljeong-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13561, Rep. of Korea; email: {jangho.seo, sanhalee}@navercorp.com.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Association for Computing Machinery.
0734-2071/2021/10-ART5 $15.00
https://doi.org/10.1145/3468144

ACM Reference format:
Won Wook Song, Youngseok Yang, Jeongyoon Eo, Jangho Seo, Joo Yeon Kim, Sanha Lee, Gyewon Lee, Taegeon Um, Haeyoon Cho, and Byung-Gon Chun. 2021. Apache Nemo: A Framework for Optimizing Distributed Data Processing. *ACM Trans. Comput. Syst.* 38, 3-4, Article 5 (October 2021), 31 pages.
https://doi.org/10.1145/3468144

1 INTRODUCTION

It is becoming increasingly important to optimize scheduling and communication for different characteristics of resources and data in distributed data processing. Examples of such characteristics widely discussed in recent literature are geographically distributed resources [14, 30, 47, 48], cheap transient resources [34, 35, 39, 52, 54], disk-based large data shuffle [32, 33, 57], and skewed data [17, 20, 21, 31]. Researchers have shown that the existing scheduling and communication methods, unaware of these characteristics, often suffer from substantial performance degradation.

Distributed runtimes such as Dryad [15], Tez [37], and the Spark runtime [44] provide low-level interfaces to plug in computation scheduler and data channel policies to optimize for such diverse deployment scenarios. These policy interfaces have direct access to control messages and data elements and can apply optimizations such as placing computations on specific types of resources and performing in-memory data shuffle. Unfortunately, runtime policy developers must exercise care to ensure that the policies they build and apply maintain correct application semantics. The main reason is that runtime interfaces are designed to be general and allow for arbitrary modifications to scheduling and communication methods.

However, policy interfaces integrated with a high-level application programming model offer indirect control over runtime execution. For example, Optimus [17] integrates with the DryadLINQ programming model to enable specifying alternative DryadLINQ subqueries. This ensures correct application semantics as long as the specified subqueries compute the same results, and thus reduces the effort required to build different optimization policies. However, such application-level interfaces do not provide sufficient fine control over distributed scheduling and communication, such as selecting the type of resources to schedule and execute specific computations on, because application programming models are designed to hide the distributed execution from application developers.

To overcome the limitations of existing interfaces, we believe it is critical to introduce a new policy interface that provides both fine control for high performance and also ensures correct application semantics for ease of use. In this work, we take a middle ground between the existing runtime and application-level interfaces. We design a policy interface that transforms an **intermediate representation (IR)** of applications to express indirect but fine-grained control over distributed scheduling and communication.

There are three main challenges to designing an optimization framework that embodies this middle-ground approach. First, the framework should define the IR transformation methods that provide fine control and also ensure correctness. Second, the framework should enable the development of reusable and composable user-defined optimization policies that transform the IR. Third, the framework should apply the transformations of the IR in the distributed execution of the application.

Figure 1 depicts our Nemo optimization framework that addresses the challenges. Specifically, its IR **directed-acyclic graph (DAG)**, compiler optimization passes, and runtime extensions address the three challenges, respectively. Nemo integrates with high-level application programming model libraries and compatible distributed runtimes.

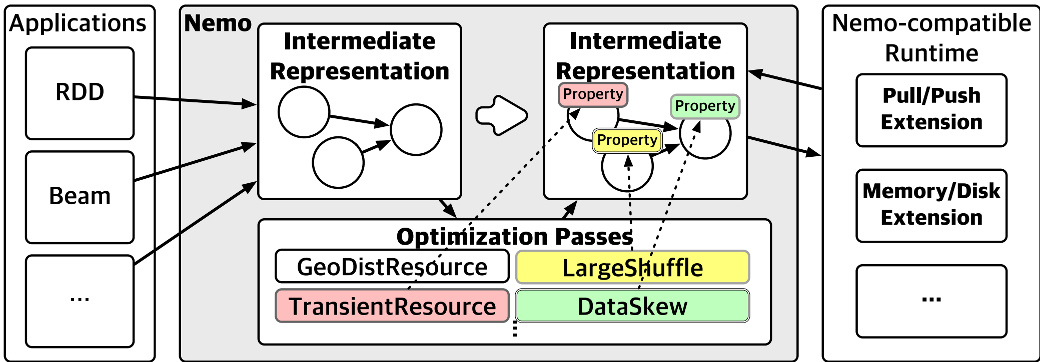

Fig. 1. Nemo optimizes scheduling and communication of distributed data processing.

First, the Nemo IR DAG represents a data-processing application with vertices representing logical operations and edges representing data dependencies. To ensure that the transformed IR DAG produces the same outputs as the original IR DAG, we provide two types of transformation methods: reshaping and annotation. Reshaping methods can insert a set of utility vertices whose semantics are known to Nemo, such as a vertex that samples data. Annotation methods set execution properties of each vertex and edge to configure fine-grained scheduling and communication, such as speculative cloning and data persistence strategies. Nemo ensures correctness using the information about the communication patterns (e.g., shuffle) of edges and the information about the configured utility vertices and execution properties.

Second, the Nemo optimization pass abstraction enables expressing such optimizations as a function that takes as input an IR DAG and calls its transformation methods. Because a pass is a simple function, different combinations of passes can be composed and applied across different applications. We show that optimization techniques previously employed in specialized runtimes, such as Iridium [30] and Pado [54], can be expressed as optimization passes with concise lines of code.

Third, the Nemo runtime extensions integrate with the underlying runtime to apply the IR DAG transformations. Runtimes typically provide a runtime DAG abstraction to run computations on a cluster of machines [15, 37, 44]. Our scheduler extension applies various scheduling policies when scheduling the IR vertices of an IR DAG through a runtime DAG. It also rewrites the runtime DAG during job execution to apply runtime optimizations. Our data channel extension applies the optimized data communication within the runtime DAG.

We have implemented Nemo and also a distributed runtime that is compatible with Nemo. At present, Nemo provides full support for Beam [40] applications and a subset of Spark RDD [56] applications. Our runtime integrates with REEF [49] to run on Hadoop YARN [43] and Mesos [13] clusters. We have evaluated Nemo in a cluster of Amazon EC2 instances using different optimization passes, datasets, and resource environments. Evaluation results show that each optimization pass brings performance improvements on par with existing specialized runtimes, and combinations of passes further improve performance for scenarios with a combination of different resource and data characteristics. Apache Nemo is open-sourced at https://nemo.apache.org as an Apache incubator project.

2 BACKGROUND

We first describe the fundamentals of distributed data processing. Then, we discuss in detail the existing runtime policy interfaces and application-level policy interfaces using concrete code examples.

2.1 Distributed Data Processing

Most distributed data processing systems adopt a dataflow-based execution model, which represents data-processing applications as DAGs. In Spark [44], **RDDs (resilient distributed datasets)** [56] represent datasets that result from source read operations and the different computational logics, defined as Spark transformations, performed on previous RDDs. On Beam [40], PCollections represent datasets that are read from data sources or computed by PTransforms, which are operations that are similar to Spark transformations. Each of such dataset abstractions are connected by data dependencies and computations between the datasets.

In these applications, the initial datasets are generated from source operations (e.g., file read from HDFS), and the following datasets are the results of the computations from these initial datasets. Each dataset is split into multiple data partitions, each of which is a portion of the dataset that can be processed by the computational operations (e.g., a text split into individual paragraphs in a word-count example). Depending on the characteristics of the computations and the number of partitions for each of the datasets, the data dependencies can be represented as a one-to-one, a shuffle, or a broadcast communication pattern. A one-to-one dependency represents an element-wise computation that can be done on a single partition and result in its corresponding partition (e.g., transforming an element to another, like splitting a paragraph string into a set of individual words of the paragraph), whereas a shuffle dependency represents a computation that produces its results from multiple input partitions (e.g., grouping elements of a specific key together, like summing the occurrences of each word from multiple sets of different words). A broadcast dependency represents input partitions that are distributed across multiple nodes for it to be available for the following computations on them.

Since computations within a *job* that involve one-to-one dependencies are element-wise operations, they can be grouped together into *stages*, which consist of computations that can be sequentially run on individual elements. Each *stage* can then be split into multiple parallel *tasks*, each of which performs the computation on its allocated partitions of the dataset on an executor.

Once a job fails to execute due to different reasons, such as task failures or lack of memory storages, it is required for tasks to be re-launched to recompute for the lost intermediate data. To reduce the recomputations, data processing systems provide caching and checkpointing mechanisms to prevent the intermediate data from being lost or to store them on reliable storage media for stable potential usages. Moreover, to handle slow tasks that are hung or stuck in a job, systems often also support speculative execution that executes backup tasks to prevent bottlenecks caused by the straggler tasks. In the next subsection, we describe how existing application and runtime interfaces describe the distributed data processing.

2.2 Optimization Policy Interfaces

In this subsection, we describe the interfaces of Dryad [15] and Optimus [17] in detail, to discuss the existing runtime policy interfaces and application-level policy interfaces. First, the Dryad policy interface allows for arbitrary modifications to its DAG representation, illustrating the execution graph of the applications. In a Dryad DAG, a vertex represents a unit of work performed on a machine and an edge represents a data transfer from a vertex to another. For example, a map-reduce application can be represented in Dryad as a number of map vertices fully connected with a number of reduce vertices with a shuffle dependency. The Dryad runtime coordinates the scheduling and communication of the vertices on a cluster of machines.

Figure 2 shows the pseudocode of two example Dryad policies [27]. Here, `ConnectionManager` is a callback-based abstraction that listens to events from the configured upstream vertices. First, `TreeAggregate` builds an aggregation tree with a goal to use network bandwidth resources more

```
class TreeAggregate implements ConnectionManager {
 void onUpstreamVertexEvent(event) { // Event is a map vertex
  mapVertexGroups = analyzeLocationsAndSizes(event) // Analyze map outputs
  aggregateVertices = newVertices(mapVertexGroups) // Create aggregate vtxs
  connect(mapVertexGroups, aggregateVertices) // Connect map to aggregate
 }
}

class Repartition implements ConnectionManager {
 void onUpstreamVertexEvent(event) { // Event is result of a map vertex
  desiredPartitions = analyzeDataStatistics(event) // Analyze samples
  modifyPartitionVertices(desiredPartitions) // Set ideal # of partitions
  modifyReduceVertices(desiredPartitions) // Also to the reduce vertices
 }
}
```

Fig. 2. Pseudocode of Dryad policies. The Dryad policy interface provides fine control over distributed scheduling and communication, but does not ensure correctness.

efficiently. Suppose `TreeAggregate` listens to the map vertices in a map-reduce application to obtain the information on the locations and sizes of map vertex outputs. Using the information, `TreeAggregate` groups map vertices, creates intermediate aggregation vertices, and then connects each map vertex group to an aggregation vertex. Second, `Repartition` dynamically distributes data with a goal to handle data skew. Suppose the map-reduce application additionally has bucketizer vertices that consume sample output data from the map vertices and partition vertices that partition the original map vertex outputs prior to transferring the data to the reduce vertices. Then, `Repartition` can be used to monitor the bucketizer vertices and modify the partition and reduce vertices with the goal to evenly distribute the map outputs. As shown by these examples, runtime policies can configure various scheduling and communication methods.

However, the flexibility of runtime interfaces comes at a cost: the policy developer must exercise care to ensure application correctness when developing, reusing, and composing different policies [15, 17, 37, 44]. First, the interface may lead to a bug in `TreeAggregate` to miss connecting one of the map vertices to an intermediate aggregation vertex, making the optimized DAG produce incorrect partial results. Second, `Repartition` can break application semantics when applied on a random vertex in a different DAG that does not use bucketizer and partition vertices. Third, applying both `TreeAggregate` and `Repartition` on the same DAG can lead to conflicting executions that produce incorrect results. Manually building a combined policy can require a significant effort for complex policies, such as the `DrDynamicAggregateManager` in Dryad, which consists of 1.3K lines of C++ code [27]. As a consequence, runtime policies have been mostly hard coded in runtimes and data-processing application compilers such as the DryadLINQ compiler [17, 55] and the Hive compiler [45]. The authors of Optimus also report that their system-level optimization policies are hard-coded in the DryadLINQ compiler, maintaining the DAG property and operator semantics for the pre-defined operators in DryadLINQ [17].

In contrast to the runtime interfaces, Optimus provides an application-level policy interface that ensures correctness by restricting the interface to substituting DryadLINQ subqueries. Figure 3 shows the pseudocode for optimizing a matrix multiplication application described in the original Optimus paper [17]. The code defines two alternative subqueries for multiplying two matrices and a policy for selecting a subquery to use for the execution. Note that as long as the two subqueries produce the same results, changing the policy code does not alter the semantics of the application.

```
// Application code creating two different subqueries
mulA = defineMatMulSubqueryA(matrixX, matrixY)
mulB = defineMatMulSubqueryB(matrixX, matrixY)

// Policy code for selecting the subquery to use for execution
stats = collectDataStatistics(matrixX, matrixY)
rewriter.registerAlternatives(stats, mulA, mulB)
```

Fig. 3. Pseudocode of an Optimus policy. The application-level Optimus policy interface ensures correctness, but provides coarse-grained control of substituting subqueries.

However, as this example shows, such application-level policy interfaces lack fine-grained control over scheduling and communication. Such fine-grained control includes selecting the types of resources to schedule specific computations on, and whether to store intermediate data on disks or in memory, which enables efficient distribution and usages of computing resources and faster data processing. The main reason is that application programming models are designed to hide the distributed execution from application developers.

3 SYSTEM DESIGN

The goal of the Nemo optimization framework is to support fine control over distributed execution of data-processing applications and at the same time maintain correct application semantics. Concretely, given a *DAG* representation of a data-processing application with deterministic operations and a user-defined policy P where $DAG' = P(DAG)$, Nemo aims to provide the following properties.

- **Correctness**: Given the same inputs the optimized DAG' should produce the same outputs as the original *DAG*, even when P is applied in the midst of the *DAG* execution. This ensures that the optimizations maintain correct application semantics.
- **Reusability**: The same P should be applicable to different *DAG*s. This enables for reusing the same policy across different data-processing applications, although the effects may differ between applications.
- **Composability**: If P and P' do not override optimizations specified by the other policy, then enable composing different policies like $P'' = (P \circ P')$. If the policies do have a conflict, then it automatically detects it for analysis. This enables distinct policies that each optimizes for a different resource or data characteristic to be incorporated into a single policy.

We show how Nemo combines an **intermediate representation (IR)** DAG, compiler optimization passes, and runtime extensions to ensure these properties. First, the IR DAG provides reshaping and annotation methods for specifying the optimizations (Section 3.1). Second, optimization passes define the functions that operate on the IR DAG methods (Section 3.2). Third, runtime extensions apply the optimizations in the underlying runtime and the execution (Section 3.3).

3.1 Intermediate Representation

The Nemo IR DAG aims to provide the desired *DAG* representation of an application. The main challenge in designing the IR DAG is defining the methods for transforming it. For Nemo to ensure the desired properties, we make explicit both the intention and the effect of the optimization for each method invocation. For example, instead of providing a single method to insert arbitrary computations, we provide multiple higher-level methods to achieve diverse goals, including, but not limited to, increasing parallelism, speculative cloning, and sampling. We describe the IR DAG reshaping and annotation methods that embody this approach, and in particular how those

methods enable ensuring correctness. We then discuss the types of applications and runtimes supported by our IR DAG design.

3.1.1 Constructing an IR DAG. To construct an IR DAG, Nemo first accepts data-processing applications written in different dataflow programming interfaces, such as Spark RDDs [56] and Beam [40] applications. These applications typically consist of computational functions defined by the programming interface. By importing our runners in these applications instead of their native runners (i.e., Spark Runtime, Google Cloud Dataflow), it calls for our visitor interface, which traverses the data-processing application and extracts the computational functions and dataflow logics and wraps them inside the Nemo IR DAG abstraction without modifying the underlying logics.

The resulting Nemo IR DAG represents a data-processing application with vertices representing logical operations and edges representing data dependencies. The vertices are connected with edges with the information on communication patterns (one-to-one, shuffle, broadcast). When executed, an IR vertex is translated into parallel tasks that run on multiple nodes. An IR edge can be translated into key-partitioned data blocks that are produced by tasks. The IR vertices and edges wrap the computational functions that are defined with the different programming interfaces, so they can be annotated and modified with our optimization passes without modifying the underlying application semantics.

Concretely, Nemo wraps the computational functions and the application logic inside the `SourceVertex` or the `OperatorVertex` interface, depending on whether the operation reads from a source or intermediate data. While `SourceVertex` contains the list of `Readables` that it can read from, `OperatorVertex` contains a `Transform` that performs `prepare`, for preparing the operation, `onData`, for processing the data, and `close`, for cleaning up, in the specified order.

Beam [40] transforms and RDD [56] transformations and actions are wrapped by the interface to construct the IR DAG. Specifically, as all Beam transforms can be expressed with the six core Beam transforms (e.g., `ParDo`, `GroupByKey`, `CoGroupByKey`, `Combine`, `Flatten`, `Partition`), we implement the interface for the core transforms. For Spark RDDs, we implement the interface to call each of the methods corresponding to transforms and actions, supported by the Spark RDD abstraction. While Nemo currently embodies the implementations for Beam and Spark, it may be easily extended to other application semantics as long as it can be wrapped by the aforementioned interfaces.

3.1.2 Transforming an IR DAG. Once the IR DAG is constructed, users can perform optimizations on the DAG while preserving the application logic itself. We expose the configurations to control the ways to schedule and transfer data during the execution on a cluster of multiple machines, but not how the data is actually processed. Users can choose to transform and customize the IR DAG themselves or to use the optimizations that are developed and provided by the Nemo developers.

Table 1 shows examples of reshaping and annotation methods that Nemo provides for transforming the IR DAG. The reshaping methods specify a utility vertex to insert into the IR DAG, and Nemo inserts new edges to connect the specified vertex with the existing vertices in the IR DAG. Utility vertices are restricted so they do not alter application semantics or the computational logics. This way, we can guarantee application correctness of the reshaping methods. Table 1 specifies four utility vertices. `Relay` and `Reshuffle` simply apply an identity function to forward data from an upstream vertex to a downstream vertex, connecting with the downstream vertex with the one-to-one and the shuffle dependency, respectively. `Sampling` vertex applies the same function as an existing vertex and consumes the same data that the existing vertex consumes. During the execution, Nemo schedules only a subset of `Sampling` tasks according to the given sampling rate.

Table 1. Example IR DAG Transformation Methods for Optimizing Scheduling and Communication

Category	Method		Description
IR DAG Reshaping: irdag.insert()	$Relay(f: x \rightarrow x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \rightarrow v), oneToOne(v \rightarrow e.dst)\}$
	$Reshuffle(f: x \rightarrow x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \rightarrow v), shuffle(v \rightarrow e.dst)\}$
	$Sampling(f: x \rightarrow sv.f(x)), sv, rate$	:	$V \cup \{v\}, E \cup \{e.comm(e.src \rightarrow v) \mid e \in E \wedge e.dst = sv\}$
	$MessageGenerator(f: x \rightarrow udf(x)), udf, e$	:	$V \cup \{v\}, E \cup \{oneToOne(e.src \rightarrow v)\}$
	(V/E = original vertex/edge set, v = inserted vertex, f = function of v, $e.comm$ = oneToOne/shuffle/broadcast)		
IR Vertex Annotation: v.set()	$Parallelism/Int$	:	sets the number of tasks for executing v
	$SpeculativeCloning/Thresholds$	:	sets the thresholds for determining and cloning straggler tasks
	$ResourceSite/Map(Index, Site)$	:	sets the geographical sites of the resources to place tasks on
	$ResourcePriority/Enum(Type)$	:	sets the priority of the resources to place tasks on
	$ResourceAntiAffinity/Set(Int)$	:	specifies the group of tasks that should run on different executors
	$ResourceLocality/Boolean$	:	sets whether to schedule the tasks where each of its input data reside
	$ScheduleGroup/Int$	:	sets the order of execution for the scheduler
IR Edge Annotation: e.set()	$DataFlow/Enum(Pull \vert Push)$	:	$e.dst$ is scheduled after $e.src$ finishes, or scheduled concurrently
	$DataStore/Enum(Memory \vert Disk)$	:	$e.src$ tasks store output data for e in memory, or disk
	$NumPartitions/Int$	:	sets the number of partitions that $e.src$ tasks create for e
	$PartitionSets/List(Set(Index))$	:	sets the partitions that each $e.dst$ task fetches for e
	$Persistence/Enum(Keep \vert Discard)$	:	sets whether to keep or discard data after $e.dst$ processes e
	$CacheID/UUID$	:	ID to identify the cached data
	$Compression/Enum(Method)$	:	method to use for compression

Reshaping methods take as input a utility vertex and additional arguments. Annotation methods take as input a key/value execution property.

`MessageGenerator` vertex applies a user-defined function on intermediate data. When a `MessageGenerator` vertex executes and completes, Nemo collects the results of the user-defined function to generate a message containing the information on execution metrics. Nemo then halts the execution of the job and uses the message to trigger a corresponding runtime optimization pass, which we describe in Section 3.2. The IR DAG also supports safe deletion for any of the inserted utility vertices.

The annotation methods configure the scheduling and communication of the vertices and edges by annotating them with specified execution properties. Table 1 specifies nine execution properties. For scheduling, we have execution properties for deciding how, where, and when to

schedule tasks. `Parallelism` and `SpeculativeCloning` configure how many tasks to schedule. `ResourceSite` and `ResourcePriority` specify where to schedule the tasks. `ResourceAntiAffinity` specifies the group of tasks that should be scattered around different executors, when specific tasks are allocated with abnormally large sizes of partitions, for instance. `ResourceLocality` configures whether to consider the locality of the tasks while scheduling to reduce the communication cost. `ScheduleGroup` specifies the order of execution for the vertices, so they are executed in the specified sequence. `DataFlow` determines whether or not to schedule source and destination tasks concurrently. In the case for data communication, we enable the configurations for the medium to store intermediate data with `DataStore`, the persistence method with `Persistence`, and the data partitioning strategy with `NumPartitions` and `PartitionSets`. `CacheID` indicates the ID of the cache data to identify and find the cached data. `Compression` configures the compression method to use for reducing the network overhead during the communication.

Combinations of different execution properties can express optimizations that require significant efforts to implement with low-level runtime policy interfaces. For example, we can configure upfront task cloning with a persistent in-memory data shuffle that pushes data eagerly from transient resources to reserved resources, through simply annotating `SpeculativeCloning` with a `threshold`, `ResourcePriority` with appropriate `transient` or `reserved` resources, `Persistence` to `discard`, `DataStore` to `memory`, and `DataFlow` property to `push` on the two vertices and the shuffle edge that connects them. The IR DAG also supports looking up the execution properties annotated on vertices and edges. Producing the same outcomes through modifying the runtime components would require significant effort and care, as the modifications require deep understanding of how runtime components interact with each other, as well as ad hoc modifications on each of the components for the different designated requirements of the optimization.

3.1.3 Ensuring Correctness. The predefined reshaping methods ensure correctness, because Nemo connects the newly inserted utility vertex with existing vertices without changing the computational logics. As shown in Table 1, only the outputs of the `Relay` and `Reshuffle` vertices are consumed by existing vertices, and these outputs are equivalent to the data that the existing vertices originally consume, producing identical results with or without the inserted vertices. The other utility vertices, however, do not reach data sinks and thus do not affect the final results that the IR DAG produces. When a utility vertex is specified to be deleted, Nemo appropriately reverts the changes and safely returns to its previous state of the DAG before the insertion. Staying within the constraints in the reshaping methods, Nemo can guarantee that the computational semantics are kept valid and correct.

The annotation methods ensure correctness through allowing Nemo to simply examine and configure the execution properties, which affect how data is processed, but not on the input and the output of the computation itself. For each vertex in the IR DAG, Nemo checks the execution properties of its neighboring edges and vertices, with the communication patterns of the edges, and configures the execution properties appropriately. This ensures correctness, as the annotated execution properties do not use and modify computation semantics [12, 16, 58] inside each vertex or have direct access to the control messages and data elements in the runtime. Instead, execution properties simply control the way the application is scheduled and how the data is transferred during the execution runtime, without incurring any effect on the correctness of the application.

Nevertheless, there are several dependencies between the execution properties and requirements for individual execution properties that need to be followed for the execution to be made possible with the configurations. Although the dependency requirements may seem obvious for the experts who understand the runtime execution, it would be prone to mistakes if used negligently without such knowledge. To prevent such mistakes, Nemo is designed with multiple rule-based *integrity checkers* that can be extended to additional conditions regarding the execution

properties and communication patterns. For each vertex in the IR DAG, Nemo integrity checker examines the execution properties of the vertex and its neighboring edges and vertices, along with the communication patterns of the edges.

For example, Nemo checks that `PartitionSets` and `NumPartitions` are only set on shuffle edges. Nemo also checks that the sets included in the `PartitionSets` are disjoint and together contain all offsets for the `NumPartitions` to read each partition exactly once. Sometimes, execution properties have requirements on the given conditions. The `Parallelism` property is required for each vertex to set the number of parallel tasks to schedule for each of the computations, and it has to be configured with the same number within the sets of vertices that are connected with one-to-one edges. Also, `ScheduleGroup` property should be set with appropriate values, so the execution can be scheduled while preserving the topological order, and not the other way around. However, Nemo intentionally leaves some execution properties unchecked when they do not need to be checked. For instance, the `Persistence` property has no dependency, as discarded intermediate data can always be recomputed from the source data when needed and does not incur correctness issues. Likewise, the `DataLocality` property also has no dependency, as data can always be retrieved from remote executors if they are not locally available.

The integrity checker runs after each optimizations to check that each of such conditions and rules are met after each modifications on the IR DAG. Each of the integrity checks is designed to check for its designated conditions, such as those mentioned in the previous paragraph, while topologically traversing the IR DAG. In addition, users are also given the liberty to utilize other methods to perform actions outside the boundary of our provided APIs shown in Table 1 for designing new optimizations or introducing new execution properties. However, in such cases, it is the role of the optimization developers to integrate the additional rules to check for in the integrity checker to guarantee the application correctness. Since we keep the responsibilities of ensuring correctness of the new optimizations on the developers, IR DAGs could break due to unforeseen consequences of the optimizations. In such cases, one could fix the optimizations after detecting the problem, or one could simply revert the IR DAG to the state before the problematic optimization for the application to correctly execute.

Moreover, our transformation methods ensure correctness even when dynamically invoked in the midst of the IR DAG execution. As the IR DAG is decoupled from the underlying runtime, Nemo ensures correctness by controlling when to apply the transformations on the IR DAG during runtime. Specifically, we define that a vertex is being executed when its tasks are being executed, and an edge is being executed when its source or destination vertex is being executed. First, if the transformed vertices and edges have not yet been executed, then we apply the changes immediately so the changes are applied on the actual execution. Second, if they are in the middle of being executed, then we delay applying the changes until the executions are finished to ensure correctness. Third, if they have already finished execution, then we apply the changes immediately so the changes are applied and used when they are re-executed due to reasons such as system faults.

3.1.4 Supported Applications and Runtimes. As mentioned earlier, the current design of the IR DAG supports data-processing applications that can be represented as a DAG of data-parallel and deterministic operators that process bounded data. Many real-world applications, including Beam and RDD batch applications and also higher-level domain-specific applications such as machine learning and SQL applications, meet this assumption. The current IR DAG would need to be extended to support other types of applications, such as those that have cyclic dependencies and process unbounded data [28].

The IR DAG assumes an underlying distributed runtime that supports configuring and applying utility vertices and execution properties. Existing runtimes can be enhanced to provide full

support for the IR DAG optimizations through introducing additional features. For example, new data channels in addition to the existing ones (FIFO, File, TCP Pipe) can be introduced in Dryad [15] to provide support for various combinations of the `DataStore`, `DataFlow`, and `Persistence` execution properties. Similarly, a feature to dynamically add computations to a running application can be introduced in Tez [37] and the Spark runtime [44] to apply utility vertices inserted at runtime.

3.2 Optimization Passes

Nemo optimization passes aim to provide the desired user-defined policy abstraction P. A pass is a function that receives an input IR DAG and produces a transformed IR DAG. We first describe the default settings that Nemo provides. We then describe how to develop and compose passes and how Nemo applies the given passes on the IR DAG using our examples.

3.2.1 The Default Pass. If the user simply wants to execute the application without applying any particular optimization designed for a specific use case, then they can choose to run the application with the default policy that configures execution properties with the default settings for the runtime execution. The default optimization passes can also be used in parts of the optimization that the user does not wish to particularly optimize. In the default parallelism optimization pass, Nemo first determines the number of partitions of the initial datasets (stages) based on the default number of partitions from the input source (e.g., HDFS). Spark either uses the default parallelism value set by the user configurations, or the largest number of partitions from the upstream RDDs including the source RDD, and uses it throughout the application. Likewise, Nemo also takes the largest partition number from the upstream datasets, but also provides the option to reduce the number of partitions by a specified factor upon each shuffle operation (e.g., by half), as shuffle operations often reduce the amount of the dataset with combine and reduce operations. One-to-one data dependencies are processed across the computations in memory, where the data is discarded after each usage. The data within shuffle and broadcast dependencies are stored and transferred on local disks, where Nemo persists the data on the disk under the default settings. We use the LZ4 algorithm for data compression by default and use the default Java serializer/deserializer for the serialization/deserialization.

Upon scheduling, Nemo traverses the IR DAG in a topological order and distinguishes the stages that can be executed in parallel from those that have dependencies on the outputs of other stages. By doing this, Nemo keeps track of the list of tasks that can be scheduled at a particular point of execution and those that can be executed after the already scheduled tasks. While scheduling, Nemo takes the resource locality into consideration, meaning that it schedules tasks on the executors where each of its input data reside, as long as the available resources allow them to be.

3.2.2 Developing and Composing Passes. We now describe the rationale and the algorithm for several example custom passes to demonstrate how to develop and compose new passes for specific goals. Users can write two types of passes: compile-time and runtime passes. Compile-time passes take an IR DAG as its input and are run prior to the actual job execution. Runtime passes additionally receive a message produced by a `MessageGenerator` vertex that triggers the runtime pass during job execution.

Each pass is composed of algorithms that utilize the IR DAG transformation methods based on the conditions defined in the pass. One can observe the execution properties of the DAG via the `get` method and `set` specific execution properties based on the conditions derived from the observations. Moreover, if required, then one can finalize an execution property to ensure that no more overriding changes are made on the execution properties set.

Geo-distributed data analytics: We aim to cope with the low and variable capacity of WAN links when processing data that are geographically distributed [14, 30, 47, 48]. To reduce network

bottlenecks, we formulate the problem of placing computations to geographically distributed sites as a **linear program (LP)**, similar to specialized scheduler extensions like Iridium [30]. Here, we use bandwidth information and data size estimations. We also use an off-the-shelf linear solver library, since Nemo allows using external libraries when writing a pass. While the entire optimization consists of a total of 177 LoC (lines of code) in Java (including imports), the pseudocode of this algorithm is as follows:

```
CompileTimePass GeoDistPass(irdag):
 solution = solveLP(bwInfo(), sizeEstimates(irdag)) // solve linear problem
 for v in irdag.vertices:
  v.set(newResourceSite(solution.get(v)))  // set new ratio
```

Harnessing transient resources: We aim to reduce recomputation costs when using transient resources that are cheap but frequently evicted [34, 35, 39, 52, 54]. Based on the communication patterns, we identify operations that incur large recomputation costs and place them on reserved resources. We place the other operations on transient resources. We also quickly move intermediate data produced on transient to reserved resources. This applies key scheduling and communication optimizations employed in specialized runtimes like Pado [54]. While the entire optimization consists of a total of 85 LoC in Java, the pseudocode of this algorithm is as follows:

```
CompileTimePass TransientResourcePass(irdag):
 for v in irdag.vertices.topologicallySorted(): // for all vertices
  if (allOneToOneFromReserved(v.inEdges)
   || !isOneToOne(v.inEdges)): // from reserved or has complex dependency
   v.set(ResourcePriority.Reserved)  // set as reserved
  else:  // from transient or has simple dependency
   v.set(ResourcePriority.Transient)  // set as transient
  for e in v.inEdges:
   if fromTransientToReserved(e.src, v):
    e.set(DataFlow.Push)  // push data from transient to reserved
```

Large-scale data shuffle: We aim to reduce random disk read overheads that can grow quadratically with data size when shuffling data, similar to specialized shuffle systems such as Sailfish [32] and Riffle [57]. We insert a `Relay` vertex to specify shuffling data in memory as soon as produced and writing the data as-is to a local disk. We also ensure that the in-memory data are discarded once transferred to avoid running into out of memory errors. Following computations sequentially read the data from the local disk, after the shuffle completes. While the entire optimization consists of a total of 55 LoC in Java, the pseudocode of this algorithm is as follows:

```
CompileTimePass LargeShufflePass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()): // for all shuffle
  rv = newRelayVertex()
  irdag.insert(rv, e) // insert relay vertex that performs group by key
  rv.inEdge.set(DataFlow.Push) // push sorted data
  rv.inEdge.set(DataStore.Memory) // on memory
  rv.inEdge.set(Persistence.Discard) // discard after pushing
  rv.outEdge.set(DataFlow.Pull) // pull the grouped data
  rv.outEdge.set(DataStore.Disk) // write to disk afterwards
```

Mitigating data skew: We aim to assign the same amount of data across parallel computations to prevent stragglers. We first set the number of partitions for the data to be shuffled. We then insert

a MessageGenerator vertex with a function for obtaining the set of data partition sizes. We also ensure that the shuffle receiver is executed after the the shuffle sender and the MessageGenerator vertex complete, at which point we will have obtained the statistics and optimized the execution of the shuffle receiver. While the entire optimization consists of a total of 76 LoC in Java, the pseudocode of this algorithm is as follows:

```
CompileTimePass SkewCTPass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()): // for all shuffle
  e.set(newNumPartitions(e)) // set the number of partitions
  e.set(DataFlow.Pull) // set as pull
  irdag.insert(newOptVertex(), sizeFunction(), e) // insert msg generator
```

At runtime, when the MessageGenerator vertex completes and makes the set of size numbers available, we partition the set into subsets such that the sum of the numbers in the subsets are as equal as possible. We then assign each subset to a distinct shuffle receiver task. While the entire optimization consists of a total of 125 LoC in Java, the pseudocode of this algorithm is as follows:

```
RunTimePass SkewRTPass(irdag, message): // message with skew info
 subsets = partition(message) // repartition based on info
 message.edge.set(newPartitionSets(subsets))
```

We have shown a few example passes to demonstrate how a pass can be designed and built. While the methods that we provide enable users to safely modify and optimize the job execution, users may utilize other Java methods at their own risk and perform further optimizations that reconstruct DAG structures, such as loop optimizations and operator fusion. Such optimizations could also be extended by the users to optimize applications to iterate until convergence, to dynamically adjust resources, such as adding or removing executors from the pool of available containers for the data processing workload, depending on the specific use case of the application.

Finally, users can organize and package multiple relevant passes to build an optimization policy that performs an optimization for a specific environment, like the following example that simultaneously handles large shuffle and data skew. When registering a runtime pass, it requires specifying a compile-time pass that inserts MessageGenerator vertices, which produce the message containing the runtime information and trigger the runtime passes that leverage the information.

```
policyBuilder.register(LargeShufflePass) // register CTPass
policyBuilder.register(SkewRTPass, SkewCTPass) // register RTPass
policy = policyBuilder.build() // build policy composed of the passes
```

3.2.3 Applying Passes. Given an IR DAG and a policy composed of passes, users are given the choices of selecting one of the optimization policies provided by Nemo to apply for their use case, to develop their own optimization policy themselves or to simply run the application using the default policy. Whichever way they choose, Nemo first applies the compile-time passes on the IR DAG in the same order as they were registered in the policy. After all compile-time passes are applied in their order, the optimized IR DAG is executed. As the execution progresses, each MessageGenerator vertex completes execution and produces a message. For each message, Nemo runs the corresponding runtime pass to transform the IR DAG. Nemo serially runs the runtime passes for the different messages.

After applying each pass, Nemo checks whether the IR DAG produced by the pass is correct with the integrity checkers, as described in Section 3.1.3, and also whether the pass encounters any conflicts with its previous passes. A conflict occurs when a pass overwrites the value of an execution property set by a previous pass to a different value, or deletes a utility vertex inserted

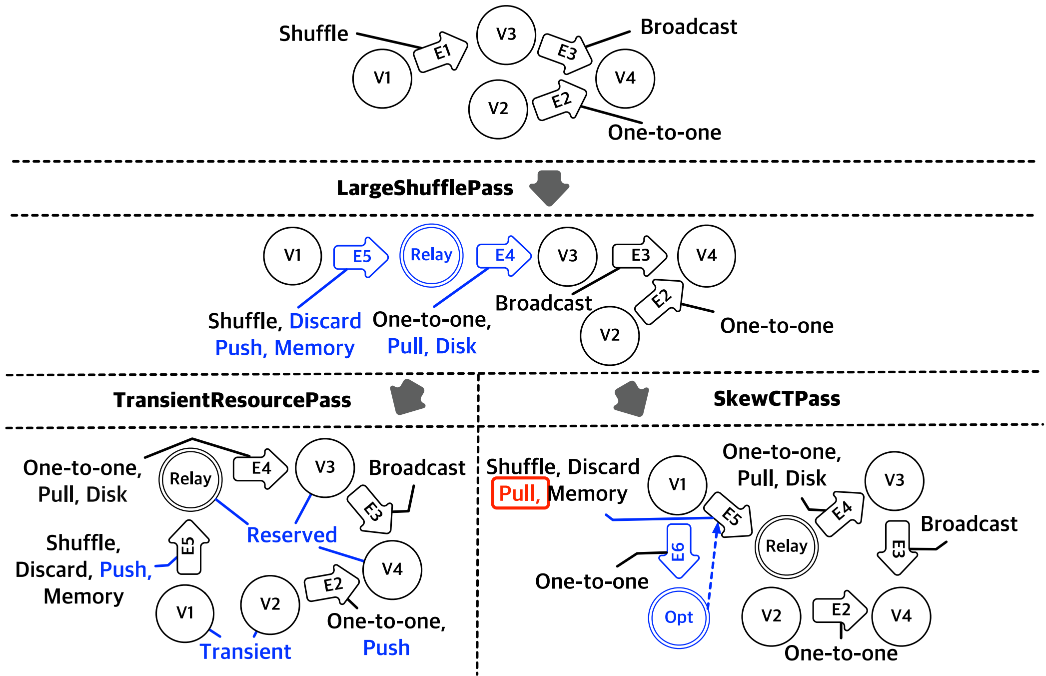

Fig. 4. A policy composed of the LargeShufflePass and the TransientResourcePass and another policy composed of the LargeShufflePass and the SkewCompileTimePass are applied on an input IR DAG.

by a previous pass. Nemo throws an error and refuses to execute in case of a check failure after running a compile-time pass. Upon a check failure of a runtime pass, Nemo just ignores the IR DAG output by the pass and logs the failure, as stopping an already running application can be costly.

Figure 4 shows how Nemo runs two example policies. Both policies first apply the LargeShufflePass, which inserts a Relay vertex between V1 and V3 and annotates E5 and E4. The first policy then applies the TransientResourcePass, which performs annotations without any conflict with the previous pass. The second policy applies the SkewCTPass, which inserts a MessageGenerator vertex and tries to annotate E5 with the pull DataFlow. However, the SkewCTPass encounters a conflict as the push DataFlow has already been set for E5 by the previous LargeShufflePass.

Fundamentally, the conflict in the second policy occurs because the LargeShufflePass tries to shuffle data eagerly in memory, whereas the SkewCTPass tries to use the statistics of the data before the downstream computations start to consume the data. If undetected, then this conflict results in a pull-based in-memory data shuffle, where the outputs of all V1 tasks are stored in memory before the Relay tasks start fetching the data. Although this configuration avoids disk seek overheads and also handles data skew at the same time, it can cause out of memory errors for large input data.

Because Nemo detects such conflicts explicitly, users can quickly detect and address the issue. In this case, we can solve the conflict by designing a new SkewSamplingPass that avoids the conflict with the LargeShufflePass. This new compile-time pass clones the IR DAG using Sampling vertices and first runs the clone to obtain the statistics of sampled data. Our third policy with the LargeShufflePass and the SkewSamplingPass can be applied together on the IR DAG to optimize for both large data shuffle and data skew. However, compared to the SkewCTPass, the

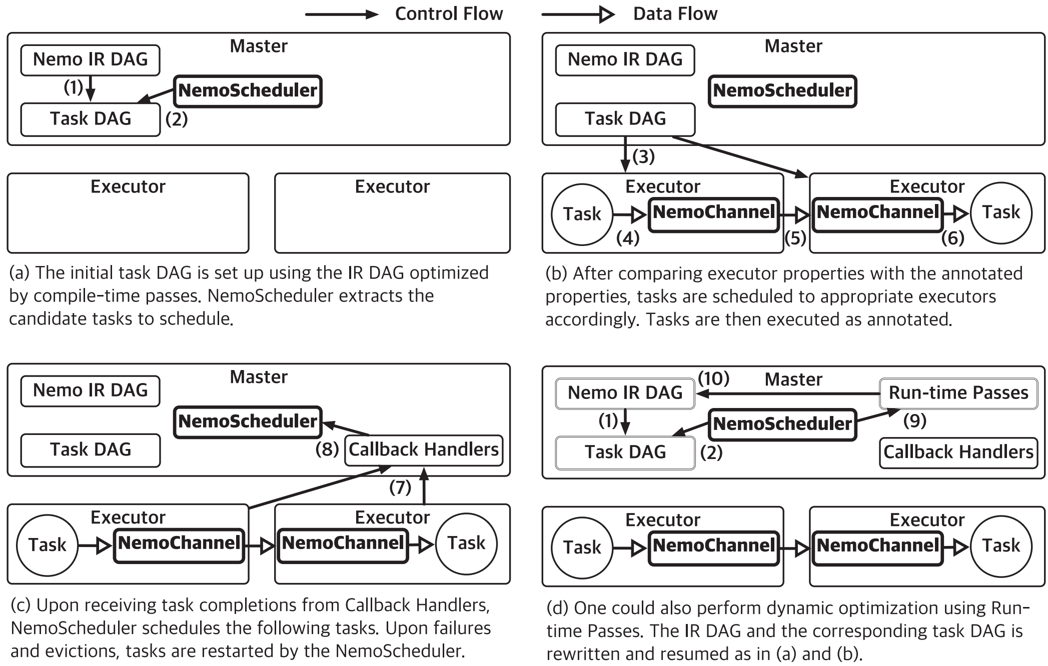

Fig. 5. Nemo runtime extensions (bold) apply optimizations in a distributed runtime.

`SkewSamplingPass` incurs the cost of executing additional vertices and using the statistics of sampled data rather than the entire data.

Next, we describe how these various transformations of the IR DAG are reflected in the distributed execution.

3.3 Nemo Execution Runtime

We use a Nemo-compatible runtime depicted in Figure 5 to describe how the Nemo runtime extensions apply the IR DAG transformations in the distributed runtime. Upon launching a job, the runtime starts a master process and executor processes on user-specified resources. Within the master, the `NemoScheduler` extension operates on the task DAG abstraction that the runtime provides for scheduling tasks to executors. Executors spawn a thread to run each scheduled task and use the `NemoChannel` extension to communicate data between the tasks. In the rest of the section, we describe how such extensions actually execute the optimizations.

3.3.1 Overview. The Nemo execution runtime keeps track of the pool of executors launched based on the user-specified resources. Upon receiving the optimized IR DAGs, the Nemo master merges multiple vertices connected with one-to-one dependencies into stages and divides each stage into parallel tasks to distribute them among the pool of executors. Once tasks are scheduled to an executor, they are executed to consume and process each of their partitions of input data. The statuses of the tasks are tracked by the Nemo master, and the metrics are collected in the `MetricStore`.

The callback handlers in the Nemo master take care of the changing statuses of the tasks. When a task asks the master to put the job on hold to perform dynamic optimization, the master does so and re-schedules the job on the executors based on the updated execution plan. Upon failures of the tasks, Nemo master provides fault tolerance by restarting the failed tasks to recover from the

failures and evictions. Also, to mitigate the effects of straggler tasks, Nemo also provides speculative execution by executing cloned tasks in parallel.

During the execution, the Nemo execution runtime provides support for different libraries and features by providing the modules specified for the different purposes. For example, Nemo provides different modules to use different storage media for the intermediate data, not only limited to memory and disk, but also to remote storage memory and off-heap memory.

3.3.2 Scheduling and Executing Tasks. First, we set up the initial task DAG using the IR DAG optimized by compile-time passes (1). Here, we merge neighboring IR vertices into the same stage as much as possible to minimize the data communication overheads within the set of tasks, while considering communication patterns of the IR edges and the related execution properties, such as the `Resource` properties and the `Parallelism` property. Whenever a task enters a new state, it notifies the master about its updated state, which has callback handlers that provide different mechanisms for each of the state changes. In case of a `MessageGenerator` vertex, we register its specific callback handler to collect the results produced by the corresponding tasks from executors as a message.

Each of the tasks are then scheduled to an executor in the topological order. Upon initiating a job, we select the candidate tasks for scheduling, which are the source tasks and their children tasks connected with the push `DataFlow` (2). For each candidate task, we select candidate executors by comparing the corresponding `Resource` properties of the task with the information on the executors. We then schedule the task to a candidate executor with the least number of running tasks (3).

When a task emits a data element, we write it to the corresponding `DataStore` implementation, creating a data block when all data elements for the channel are written (4). If the corresponding edge is shuffle, then the block is partitioned into `NumPartitions`. When a task reads input data elements, we look for the locations of the input data blocks, blocking the call when looking for blocks that are not yet available. We fetch the input data elements from the local and remote `DataStores` while applying `PartitionSets` for shuffle edges (5–6). Once all of the downstream tasks successfully read a block, we decide to either keep or discard the block based on the `Persistence` property.

3.3.3 Metric Collection. During the job execution, Nemo collects and provides metrics in three different levels: jobs, stages, and tasks. As mentioned, jobs are split into stages, each of which is a set of parallel tasks that each perform a set of computations on an executor. On the job level, it manages metrics such as the total size of input data, job duration, the IR DAG structure, the execution properties, and the job state transition between `ready`, `executing`, `complete`, and `failed` states. Stage metrics simply keep track of the completion times for the stages, while task metrics keep track of the most important fine-grained information of the workload. Task metrics include the task duration, the executor the task is scheduled to, the scheduling overhead, the number of scheduling attempts, as well as the size and time metrics of reading and writing and serialization and deserialization of the task input and output data. Task states include `on_hold` and `should_retry` in addition to the `ready`, `executing`, `complete`, and `failed` states to indicate states related to dynamic optimization and fault tolerance, relatively, which we elaborate in the following subsections. Task metrics are collected in each executor and are sent to the master upon each completion of the tasks. Job and Stage metrics are collected and updated on the master by the callback handlers that are implemented on the master. The metric collection occurs along the execution and incurs overheads on a milliseconds scale. The metrics can be stored as a JSON file or on a relational database according to the user preference.

3.3.4 Dynamic Optimization. To deal with problems that occur dynamically, Nemo provides runtime passes that enable for the dynamic modifications of the task DAG based on the metrics collected during runtime. Such metrics are usually represented as a message, as such runtime

metrics must be aggregated from the executors to a central master in advance to the dynamic optimization. Such messages are produced by a `MessageGenerator` vertex, and upon receiving such messages in the callback handler (7), the Nemo master postpones scheduling new tasks in the NemoScheduler (8). Once the tasks are put on hold, Nemo invokes the corresponding runtime pass (9), rewrites the task DAG based on the new IR DAG output by the runtime pass at the correct timing described in Section 3.1.3 (10), and resumes scheduling starting from the following tasks. Then, the new tasks, which are adjusted and optimized by the runtime pass, are executed with the optimizations applied.

3.3.5 Fault Tolerance. Faults can occur in the executors for a variety of reasons. Misconfiguration, inefficient queries, excessive concurrency, and insufficient resources can all lead to various problems, mainly to **out-of-memory (OOM)** conditions that lead to executor failures and abrupt discontinuation of workloads.

To deal with such problems during the runtime, if not handled by the optimization passes, then Nemo provides methods to recover from such failures. While Nemo keeps track of the tasks and the workload with the runtime metrics, Nemo schedules new tasks upon learning about failures on task progress and executor status in the callback handler to restart the failed tasks to recover from failures and evictions (7–8). Upon the rescheduling of the failed tasks, Nemo executors compute for the lost data with the newly scheduled tasks and resume the progress from where it has left off. Specifically, it re-schedules the tasks that are required to recompute the lost intermediate data and topologically schedules the rest of the tasks of the workload to perform the rest of the job. As tasks are regenerated and rescheduled progressively, instead of being transferred and relaunched on healthy executors, it incurs trivial rescheduling and context switching costs. In the meanwhile, to prevent wasting computational cycles on repetitive recomputations, Nemo supports caching to enable checkpointing for frequently accessed intermediate data on its storage resources.

To deal with master failures, Nemo supports resource manager restart mechanisms on YARN, which restarts the master based on reloading the states from a pluggable state-store containing the checkpointed master status and metadata.

3.3.6 Speculative Execution. Straggler tasks, inefficient shuffle operations, and excessive garbage collection in specific executors contribute to major slowdowns of job execution. In addition to task faults, such slowdowns have major impact on the system performance, leading to inefficient usages of the computational resources and time, leading to consequences that can turn out as even more fatal than executor faults.

To deal with the straggler tasks, Nemo clones tasks based on the `SpeculativeCloning` property and executes backup tasks. While there exist several methods to distinguish straggler tasks from others, sampling approach is one of the approaches that enables for the estimation of the task completion times of parallel tasks. For instance, if a stage is split into n parallel tasks for the execution, we could sample a small portion of the n parallel tasks to derive the duration of the entire stage. The backup tasks are executed in parallel on other executors, which might not possess the problems that the straggler executors suffer from. Upon the completion of the tasks, the other set of tasks is canceled and the job continues on its execution.

3.3.7 Simulator. On top of the Nemo runtime, Nemo also provides a separate `scheduler` implementation that enables for a `simulation` of a workload, which follows the exact, same procedure as the runtime execution except for the actual computations and I/O operations. Jobs are split into stages and tasks and scheduled as they would originally have been in a cluster of multiple machines. Through the simulation of the distributed computational environments, the duration of the workload can be derived from the predicted task completion times. The task completion times

could be represented as a number portraying the average task duration or as a distribution where the predicted task completion time could be extracted from the distribution. With the simulator, Nemo enables users to predict the workload execution and to fix inefficient queries ahead of the execution without having to wait for the actual execution of the workload.

3.3.8 Integration of Remote Storage Memory. With the advancement of high-performance storage and fast network, data processing has been able to utilize such technologies to provide speed enhancement in executing data processing queries. Although using disaggregated memory in some cases could lead to extra serialization overheads, such as when used too frequently for small data, in other cases, disaggregated memory can bring advantages over the classic local memory. Disaggregated memory makes memory resource management much easier and efficient in large datacenters, bringing down the cost for the management and resource distribution. Utilizing disaggregated memory can bring down the memory contentions in local memory, which may lead to OOM errors and enable for more memory to be utilized for computation rather than as a storage for large intermediate data, while keeping the overheads low. Such characteristics could be leveraged to off-load skewed data with popular keys to disaggregated memory to reduce local memory pressures.

Apache Crail [41] is one of the libraries that facilitates the usage of heterogeneous storages like DRAM and NVMe flash with the support of advanced network technologies like RDMA. On top of Apache Nemo, we have implemented support for Crail, through connecting the Nemo runtime with the Crail adapters and creating a new `DataStoreProperty` called `CrailFileStore`, in addition to the existing properties such as `LocalFileStore` and `MemoryFileStore`. This facilitates the usage of Crail file system by making it as simple as just annotating the IR DAG edge with the appropriate execution property through the provided API, for storing the intermediate data, and enjoy the benefits of the new technologies.

3.3.9 Utilization of Off-heap Memory. While many data processing systems, including Spark [44], Flink [42], and Nemo runtime are implemented to run on the **Java Virtual Machine (JVM)**, memory management plays a crucial part as such systems store large amounts of data in memory. Nevertheless, JVM has a major reliance on Java memory management and **garbage collection (GC)**, which are often sub-optimal and pose a major issue in the system performance. To solve the garbage collection issues, one can use data structures with fewer Java objects, such as using arrays instead of lists, or use specialized data structures such as Koloboke or fastutil, which optimize memory usage for primitive types. Another solution is storing data off-heap as Spark and Flink, which is the region in the JVM that is not handled by the JVM GC algorithms, to handle memory manually.

Apache Nemo provides the support for using off-heap memory by utilizing the `ByteBuffer` object for storing intermediate data in the Nemo runtime and providing a new `DataStoreProperty` called `SerializedMemoryStore`, in addition to the `MemoryStore` property, which stores objects on heap. Using the new execution property, similar to how we have used disaggregated memory for intermediate storage, users can simply annotate the IR DAG edge to indicate where to save the intermediate data corresponding to the IR edge. During the runtime execution, Nemo internally keeps track of the metadata of memory chunks that are allocated off-heap.

4 IMPLEMENTATION

We have implemented Nemo and a distributed runtime that is compatible with Nemo in around 32K lines of Java code. Our Nemo compiler implementation consists of the following three components: frontend, optimizer, and backend.

The frontend translates applications such as Beam and RDD applications into an IR DAG (Section 3.1). At present, our frontend provides translation support for all Beam [40] operators and

a subset of `RDD` [56] operators such as `map`, `reduce`, `collect`, `broadcast`, and `cache`. The main reason for not fully supporting RDDs is that the current iterator implementation used in Nemo is not readily compatible with some of the `RDD` implementations. In the future, we plan to modify our iterator implementation to address this limitation. The optimizer applies optimization passes on the IR DAG (Section 3.2). The backend configures the underlying runtime with the optimizer and the runtime extensions (Section 3.3).

Existing Beam applications can run on Nemo by specifying the Beam `PipelineRunner` implementation as our implementation of the runner, `NemoPipelineRunner`. This can either be done by simply passing on `NemoRunner` as a configuration parameter or directly importing it from the code. The frontend wraps and converts each Beam `PTransform` in an IR vertex and `PCollection` to an IR edge. The frontend also obtains the information regarding the communication patterns during the translation. For example, it specifies shuffle edges on the incoming `PCollections` of the `GroupByKey` `PTransforms`.

Similar to Beam, existing RDD applications can run on Nemo with simple modifications to the lines importing the implementations of `SparkSession` and `SparkContext` to our implementations of the classes. Each RDD becomes an IR edge, and each user-defined function that generates an RDD becomes an IR vertex. Our frontend also aims to respect all of the user-specified parameters on RDDs such as parallelism and data caching by setting the execution properties on the translated IR DAG accordingly.

Our runtime implementation is built on top of Apache REEF [49] and consists of master and executor processes similar to the Nemo-compatible runtime described in Section 3.3. **REEF (Retainable Evaluator Execution Framework)** is a library for easily developing execution runtimes on top of different resource managers. A REEF job consists of a single driver that obtains containers from a resource manager and multiple evaluators that provide runtime environments on containers. To take advantage of the abstractions provided by REEF, the runtime master runs as the REEF driver and the runtime executors run as REEF evaluators. Through the integration with REEF [49], our runtime runs on resource managers such as Apache Hadoop YARN [43], and Apache Mesos [13].

5 EXPERIMENTAL EVALUATION

We evaluate Nemo on the following three dimensions: First, we evaluate how Nemo applies fine control under different resource and data characteristics. Second, we evaluate how different combinations of optimization passes optimize the same application. Third, we evaluate how the same Nemo policy optimizes different applications.

We run data-processing applications with different combinations of following resource and data characteristics: geographically distributed resources, transient resources, large-shuffle data, and skewed data. We run each application five times, and we report the mean values with error bars showing standard deviations.

We use h1.4xlarge Amazon EC2 instances, each of which provides 16 vCPUs, 64 GiB memory, two 2 TB HDDs, and 10 Gbps network. We use different numbers of instances for different experiments. On each instance, one of the two disks is used by a Hadoop Distributed File System [43] cluster that we set up on the instances, and the other is used as a scratch disk for maintaining intermediate data. Input datasets are stored in HDFS and fetched by the systems at the beginning of each job.

5.1 Fine Control

In this experiment, we evaluate how Nemo applies fine control under different resource and data characteristics. For comparison, we run Spark 2.3.0 [44], because it is an open-source,

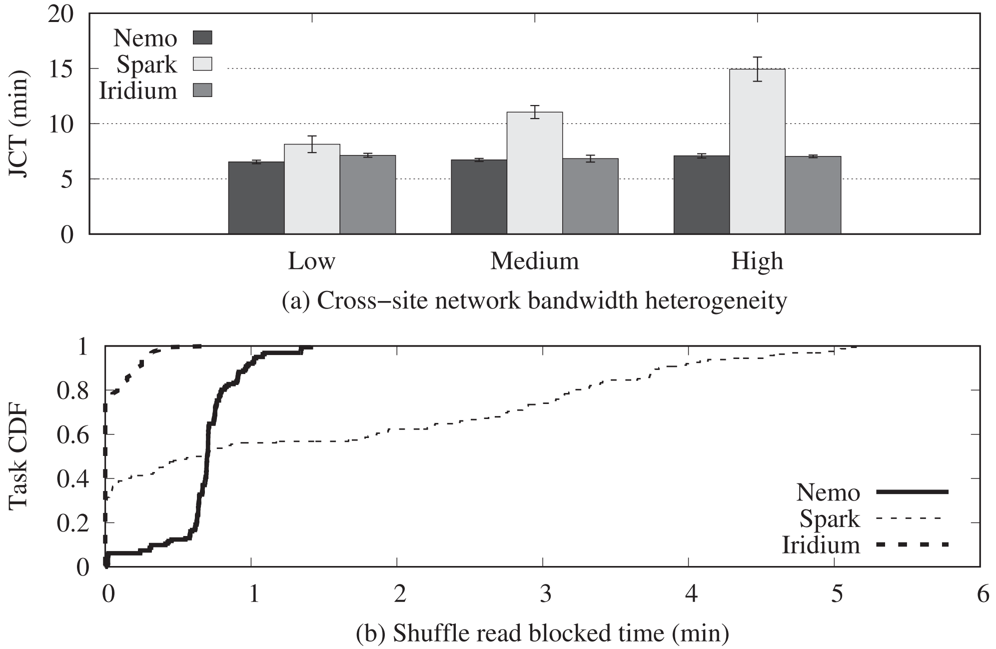

Fig. 6. (a) JCT for different cross-site network bandwidths and (b) the CDF of shuffle read blocked time of tasks under the high cross-site network bandwidth heterogeneity, where Spark shows a long tail compared to the others.

state-of-the-art system. We enable any available basic optimizations, like speculative execution, for our Spark experiments. We also run a specialized state-of-the-art runtime for each deployment scenario. Specifically, we run Iridium [30] for geo-distributed resources, Pado [54] for transient resources, and Hurricane [3] for data skew, which already has done its comparisons versus Spark in their works and has proven that it performs better. We examine the results of Beam applications on Nemo and Pado, Spark RDD applications on Spark and Iridium, and a Hurricane application on Hurricane.

We confirm that the baseline performance is comparable for Beam and basic RDD applications on Nemo. We also confirm that the baseline performance is comparable for Spark and Nemo with the `DefaultPass`, which configures pull-based on-disk data shuffle with locality-aware computation placement similar to Spark. We observe that the overhead of running the compile-time passes on Nemo is roughly 200 ms for each execution, which is small compared to the entire job completion time of each of the applications, as you can see in each of the experiments below. We also measure and report runtime overheads of the `Relay` vertex, `Trigger` vertex, and `SkewRTPass` throughout this section.

Geo-distributed Resources: To set up geo-distributed resources and heterogeneous cross-site network bandwidths, we use Linux Traffic Control [2] to control the network speed between instances, as described in Iridium [30]. Each site is configured with 2 Gbps uplink network speed and a specific downlink network speed between 25 Mbps and 2 Gbps. We experiment with Low, Medium, and High bandwidth heterogeneity with the fastest downlink outperforming the slowest downlink by 10×, 41×, and 82×. With this, we use 20 EC2 instances as resources scattered across 20 sites. The compile-time optimization takes about 173 ms on average. To evaluate data shuffle under heterogeneous network bandwidths, we use a workload that joins two partitions of 373 GB Caida [5] network trace dataset and computes network packet flow statistics.

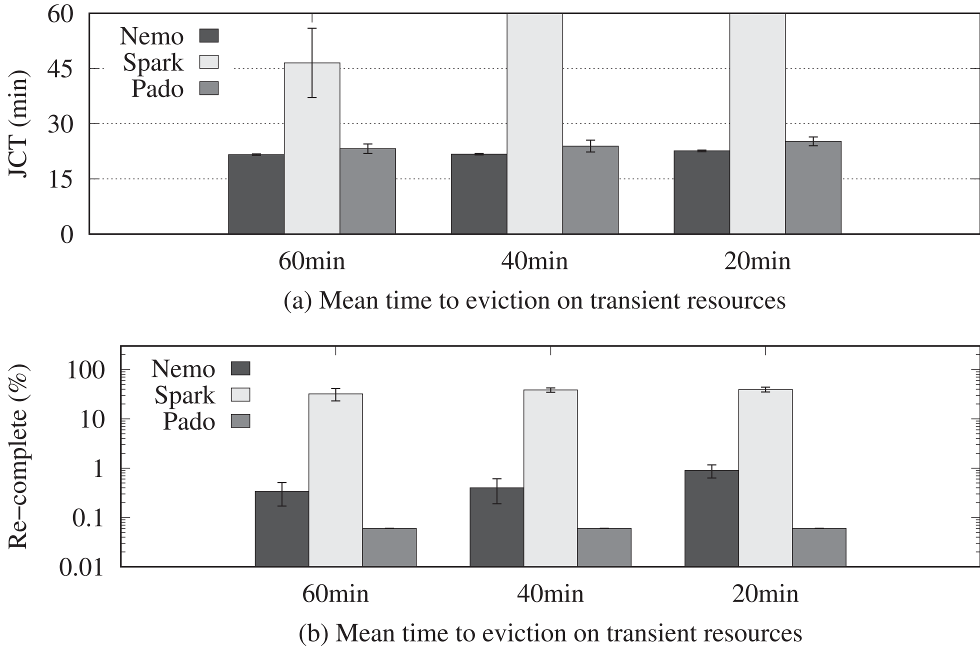

Fig. 7. (a) JCT and (b) ratio of re-completed tasks to original tasks for different mean times to eviction on transient resources.

The **job completion time (JCT)** of Iridium, Spark, and Nemo optimized with the `GeoDistPass` are shown in Figure 6(a). Spark degrades significantly with larger bandwidth heterogeneity, since tasks that fetch data through slow network links become stragglers. In contrast, Iridium and Nemo are stable across different network speeds. Figure 6(b) shows that the **cumulative distributive function (CDF)** of shuffle read time has a long tail for Spark compared to Iridium and Nemo. Iridium and Nemo show comparable performance with similar largest shuffle read blocked times, although Iridium shows overall better shuffle read blocked times using a more sophisticated linear programming model.

Transient Resources: Based on existing works [39, 52, 54], we classify resources that are safe from eviction as reserved resources and those prone to eviction as transient resources. We set up 10 EC2 instances for providing transient resources and 2 instances for reserved resources. When an executor running on transient resources is evicted, we allow the system to immediately relaunch a new executor using the transient resources to replace the evicted executor as described in Pado [54]. To evaluate handling long and complex DAGs with transient resources, we run an **Alternating Least Squares (ALS)** [18] workload, an iterative machine learning recommendation algorithm, on 10 GB Yahoo! Music user ratings data [51] with over 717M ratings of 136K songs given by 1.8M users. We use 50 ranks and 15 iterations for the parameters. The compile-time optimization takes about 236 ms on average. By varying the mean time to eviction for transient resources, we show how systems deal with the different eviction frequencies. The distribution of the time to eviction is approximated as an exponential distribution, similar to TR-Spark [52].

Figure 7(a) shows the JCT of Pado, Spark, and Nemo optimized with the `TransientResourcePass` for different mean times to eviction. With the 40-minute and 20-minute mean time to eviction, Spark is unable to complete the job even after running for an hour, at which point we stop the job. The main reason is heavy recomputation of intermediate data across multiple iterations of the ALS algorithm, which is repeatedly lost in recurring evictions. However, Nemo and Pado

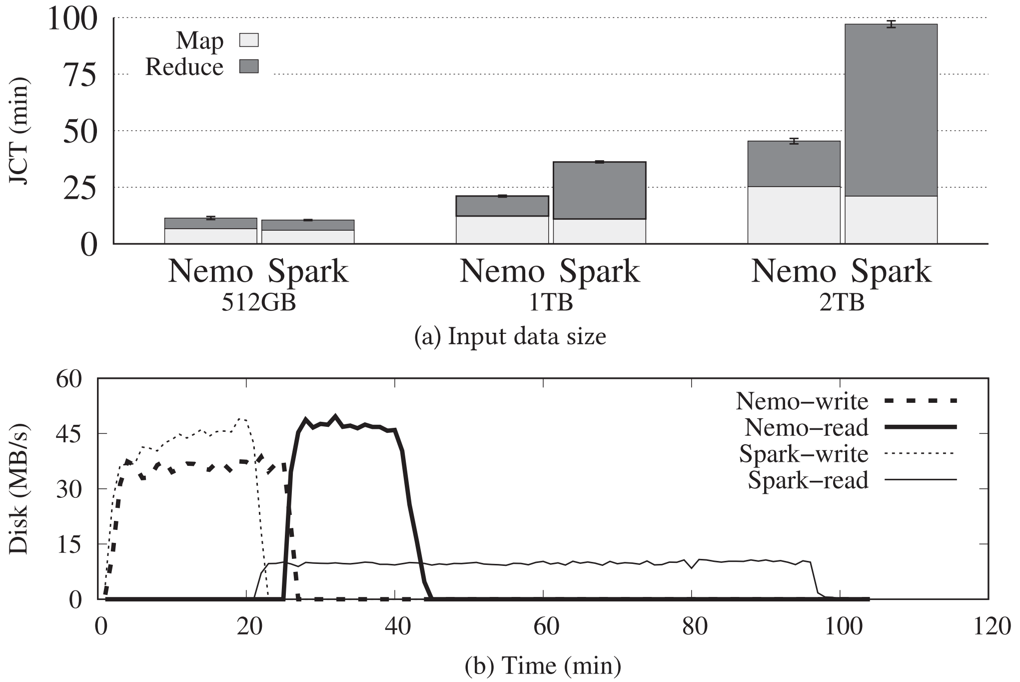

Fig. 8. (a) JCT for different input data sizes and (b) mean throughput of scratch disks for maintaining intermediate data when processing the 2 TB input data.

successfully finish the job in around 20 minutes, as both systems are optimized to retain a set of selected intermediate data on reserved resources. Figure 7(b) shows the ratio of re-completed tasks to original tasks for different mean times to eviction. It shows that Nemo and Pado re-complete significantly fewer tasks compared to Spark, leading to a much shorter JCT. Nemo and Pado show comparable performance although Nemo re-completes more tasks, because the tasks that both systems re-complete are executed quickly and do not cause cascading recomputations of parent tasks.

Large-shuffle Data: We evaluate how Nemo and Spark handle large shuffle operations using 512 GB, 1 TB, and 2 TB data of the Wikimedia pageview statistics [50] from 2014 to 2016, as the datasets provide sufficiently large amount of real-world data. We use a map-reduce application that computes the sum of pageviews for each Wikimedia project. The compile-time optimization takes about 155 ms on average. We choose the ratio of map to reduce tasks to 5:1, similar to the ratios used in Riffle [57] and Sailfish [32], and use 20 EC2 instances to run the workload.

The JCT of Spark and Nemo optimized with the `LargeShufflePass` are shown on Figure 8(a). Both show comparable performance for the 512 GB dataset, but Nemo outperforms Spark with larger datasets. To understand the difference, we measured the mean throughput of the disks used for intermediate data. Figure 8(b) illustrates the mean disk throughput of scratch disks used for intermediate data when running the 2 TB workload. Here, a spike in the write throughput is followed by a spike in the read throughput, which illustrates disk writes during the map stage followed by disk reads during the reduce stage while performing the shuffle operation. For Spark, the disk read throughput during the reduce stage is as low as about 10 MB/s, indicating severe disk seek overheads. In contrast, the throughput is as high as 45 MB/s for Nemo, as the `LargeShufflePass` enables sequential read of intermediate data by the following reduce tasks, which minimizes the disk seek overhead.

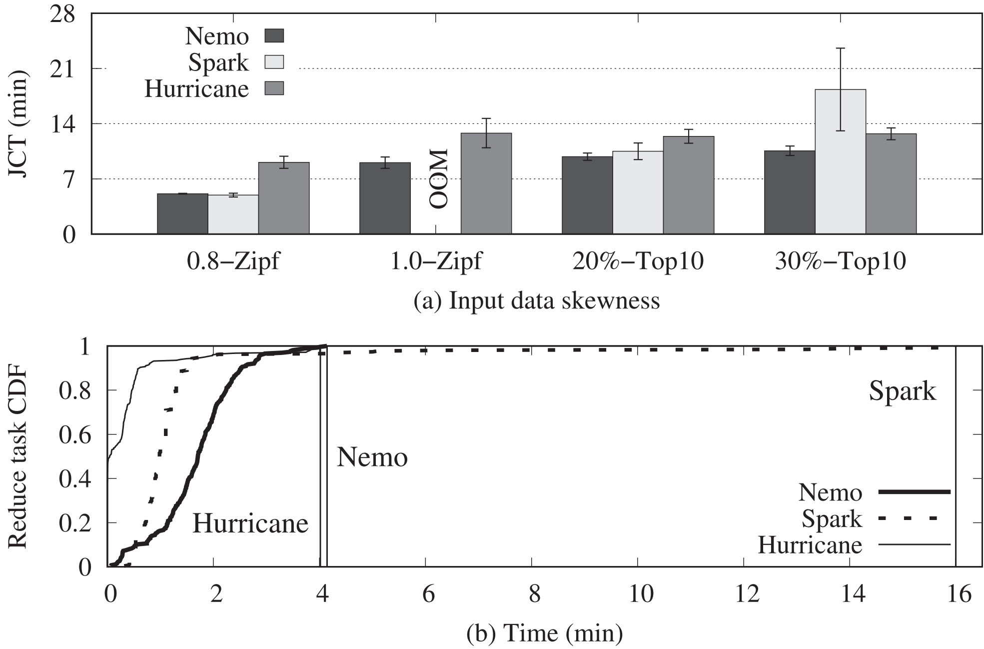

Fig. 9. (a) JCT for different input data skewness and (b) CDF of reduce task completion time when processing the 30%-Top10 skewed data. Each vertical line in the CDF graph denotes the completion time of the slowest reduce task.

To measure the overhead of the `Relay` vertex inserted by the `LargeShufflePass` before the reduce operation, we have also run the 2 TB workload on Nemo without the `LargeShufflePass`. The reduce operation begins 56 seconds earlier without the `LargeShufflePass` and the `Relay` vertex, where 56 seconds represent 2.05% of the JCT of Nemo with the `LargeShufflePass`.

Skewed Data: To experiment with different degrees of data skewness, we generate synthetic 200 GB key-value datasets with two different key distributions: Zipf and Top10. For the Zipf distribution, we use parameters 0.8 and 1.0 with 1 million keys [3]. Datasets with Top10 distribution have heaviest 10 keys that represent 20% and 30% of the total data size. We run a map-reduce application that computes the median of the values per key on 10 EC2 instances. The compile-time optimization takes about 188 ms on average, and the runtime pass takes about 268 ms on average. Because this application is non commutative-associative, for evaluating Hurricane, we use an approximation algorithm similar to Remedian [36] to fully leverage its task cloning optimizations [3]. The Hurricane application also uses 4 MB data chunks and uses its own storage to handle input and output data, similar to the available example application code.

Figure 9(a) shows the JCT of Hurricane, Spark, and Nemo optimized with the `SkewCTPass` and the `SkewRTPass`. Performance of Spark degrades significantly with increasing skewness. Especially, Spark fails to complete the job with the 1.0 Zipf parameter, due to the load imbalance in reduce tasks with skewed keys, which leads to out-of-memory errors. In contrast, both Nemo and Hurricane handle data skew gracefully. In particular, Nemo achieves high performance and at the same time computes medians correctly without using an approximation algorithm.

Figure 9(b) shows the CDF of reduce task completion time when processing the 30%-Top10 dataset. The CDF for Spark shows that reduce tasks with popular keys take a significant amount of time to finish compared to its shorter tasks. In contrast, the slowest task completes much quicker than Spark for Hurricane and Nemo. We can observe that Hurricane processes short-lived tasks

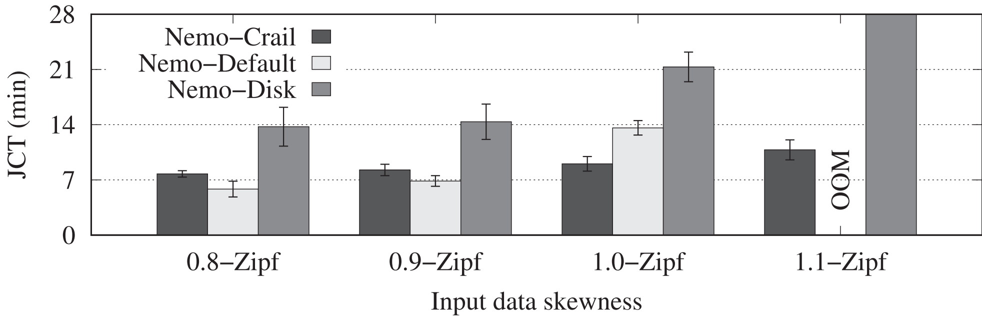

Fig. 10. JCT for different input data skewness using different data storages for the intermediate data.

alongside with longer tasks with its task cloning optimization, showing an illusion that it performs better in the bottom 50% of the CDF, but we can see that longer and shorter tasks have balanced completion times on Nemo with its data repartitioning optimization.

To measure the overhead of the `Trigger` vertex inserted by the `SkewCTPass`, we also run the 30%-Top10 workload on Nemo without the `SkewCTPass` and the `SkewRTPass`. The reduce operation begins 35 seconds earlier without the `Trigger` vertex, where 35 seconds represent 5.52% of the JCT of Nemo configured with the `SkewCTPass` and the `SkewRTPass`.

These results for each deployment scenario show that each optimization pass on Nemo brings performance improvements on par with specialized runtimes tailored for the specific scenario.

Integrating the Crail Filesystem: Another simple way of dealing with skewed data is to utilize fast data storage during data processing. As shown in Figure 9(a) and Figure 10, default implementations of general data processing systems based on local memory fail to complete the jobs from a certain extent of skewness due to OOM errors that occur while using the local memory as the medium to store intermediate data. Nevertheless, if we use Crail [41], a high-performance distributed data store designed for fast sharing of ephemeral data, for storing the intermediate data, then we can reduce the costs from skewed data by overcoming the costs with the fast storage and network performances. For this evaluation, we have simply set the `DataStoreProperty` for storing intermediate data to `CrailStore`, without any other optimizations described above, to compare only effects of the data store.

As shown in Figure 10, while the default usage of local memory brings the best performance among the implementations when the level of skewness is moderate, the performance quickly starts to sharply degrade upon memory pressure caused by the skewness and fails to perform under conditions when the skewness becomes large due to OOM errors. Crail exhibits larger network costs compared to local memory due to its distributed architecture and is slower under moderate skewness, but its performance degrades gracefully even towards intensive skewness, as the skewness has much smaller effect on the local memory pressure. Although using disks could also prevent the system from facing OOM errors, we can see that the disk overheads largely exceed the JCTs of the default and Crail-integrated Nemo. This particular experiment is performed on GCP Compute Engine VM Instances with the same specifications as our other experiments (10 instances each with 16 vCPUs and 64 GiB memory).

5.2 Composability

We now evaluate combinations of different optimization passes. Table 2 summarizes the results.

Skewed Data on Geo-distributed Resources: In this experiment, we use the same 1.0-Zipf workload for the skew handling experiment in Section 5.1, because the workload showed the

Table 2. JCT When Using Different Combinations of DefaultPass (DP), GeoDistPass (GDP), SkewCTPass (SKP), TransientResourcePass (TP), LargeShufflePass (LSP), and SkewSamplingPass (SSP)

Skewed data on Geo-distributed	Large Shuffle on Transient	Large Shuffle with Skewed
DP: OOM	DP: 100m	DP: OOM
GDP: OOM	TP: OOM	LSP: OOM
SKP: 27.2m	LSP: 100m	SSP: OOM
GDP + SKP: 14.9m	TP + LSP: 48.2m	LSP + SSP: 31.4m

largest load imbalance. We use 10 EC2 instances representing geo-distributed sites with heterogeneous network speed in between 25 Mbps to 2 Gbps. Here, DP and GDP run into out-of-memory errors due to the reduce tasks with skewed keys that are requested to process excessively large portions of data. SKP and GDP+SKP both successfully complete the job with the skew handling technique in SKP, but GDP+SKP outperforms SKP by also benefiting from the scheduling optimizations in GDP.

Large Shuffle on Transient Resources: For this experiment, we use the same 1 TB workload for the large shuffle experiment in Section 5.1, to use sufficiently large data that incurs disk seek overheads. In this case, we use 10 reserved instances and 10 transient instances with the 20-minute mean time to eviction setting.

Most notably, DP and LSP fail to complete even after 100 minutes, at which point we stop the job and TP runs into out-of-memory errors. We have observed that heavy recomputation caused by frequent resource eviction significantly slows down the DP and LSP cases. We have also found out that the LSP optimization makes the application much more vulnerable to resource evictions compared to DP. The main reason is that with LSP, eviction of a single receiving task in the shuffle boundary leads to the entire recomputation of the sending tasks of the shuffle operation to completely re-shuffle the intermediate data in memory. In contrast, DP does not need to recompute shuffle sending tasks whose output data are not evicted and stored in local disks. TP by itself also is not sufficient, as it leads to out-of-memory errors while pushing large shuffle data in memory from transient resources to reserved resources.

TP+LSP is the only case that successfully completes the job by leveraging both optimizations in TP and LSP. With TP+LSP, the job pushes the shuffle data from transient to reserved resources and also streams them to local disks on reserved resources that are safe from evictions. This allows TP+LSP to handle frequent evictions on transient resources and also to utilize disks for storing large shuffle data with minimum disk seek overheads. However, TP+LSP incurs the overhead of using only half of the resources (transient or reserved) for each end of the data shuffle. As a result, the JCT for TP+LSP with transient resources is around twice the JCT for LSP without using transient resources, which is displayed in Section 5.1. Nevertheless, we believe that this overhead is worthwhile, taking into account that transient resources are much cheaper than reserved resources from the perspective of datacenter utilization [35, 54].

Large Shuffle with Skewed Data: For this experiment, we generate a synthetic key-value dataset with a skewed key distribution that is around 1 TB in size, as the datasets used in Section 5.1 for skew handling are not sufficiently large to incur disk seek overheads. This dataset has the distribution where heaviest 20 keys represent 30% of the total data size. Using this dataset, we run the same application that we have used for the skewed data experiment in Section 5.1 on 20 EC2 instances.

In this experiment, only SSP+LSP successfully completes the job, whereas all other cases run into out-of-memory errors. DP and LSP fails to complete the job, due to particular tasks assigned with

excessively large portions of data, incurring out-of-memory errors. SSP by itself also runs into out-of-memory errors although it repartitions data across the receiving tasks of the shuffle boundary. We have observed that with large data size, the absolute size of the heaviest keys is significantly larger compared to smaller scale experiments with skewed data shown in Section 5.1. Without the LSP optimization, this problem is combined with random disk read overheads that degrade the running time of the shuffle receiving tasks, leading to out-of-memory errors. In contrast, SSP+LSP successfully completes the job by leveraging both of the optimizations from SSP and LSP.

These various results confirm that Nemo can apply combinations of distinct optimization passes to further improve performance for deployment scenarios with a combination of different resource and data characteristics.

5.3 Reusability

Finally, we evaluate how the same Nemo policy optimizes different applications. As Nemo optimization passes are simply functions that transform IR DAGs to optimized IR DAGs, they can be freely reused on an arbitrary IR DAG, with an exception on that conflicts between multiple optimizations have to be resolved. With this experiment, in addition to different applications used in prior experiments, we apply the policies on several ad hoc BeamSQL [40] TPC-H [46] **queries (Q)** with different **scale factors (SF)**, as they are widely used for benchmarking distributed data processing systems. Here, 1 SF is approximately 1 GB of input data. We specifically use workloads that handle smaller input and intermediate data compared to the previous experiments and thus are much less affected by the issues that occur in the specific scenarios such as disk-seek overheads and resource evictions.

First, using 20 nodes with the `LargeShufflePass`, we observe 20.8 minute JCT for SF1000 Q3 that is 25% smaller than the JCT without the optimization, but no significant performance improvements for SF1000 Q14. We also observe 41.1 minute JCT for SF3000 Q12 that brings 22% performance improvements. Second, we do not observe meaningful performance improvements for SF100 Q4 and Q13 with the `SkewCTPass` on 10 nodes, as the dataset is not skewed. Finally, using eight transient nodes with the 10-minute expected eviction rate and two reserved resources, we apply the combination of the `TransientResourcePass` and the `LargeShufflePass` on SF100 Q4 and Q14. For the respective queries, we observe JCTs of 8.2 minutes and 3.4 minutes, which are smaller than when not applying the optimizations by 9% and 15%.

These results as well as the results of different workloads in previous experiments confirm that the same optimization passes on Nemo can speed up different workloads instantly, with varying degrees of effectiveness.

6 RELATED WORK

Nemo builds on many years of research in dataflow processing, relational database, and compiler optimizations. Nevertheless, we believe the set of tradeoffs we have chosen to design the IR DAG, optimization passes, and runtime extensions for optimizing distributed dataflow processing makes Nemo a unique system.

Dataflow processing: Nemo differentiates itself from the existing application-level [17] and runtime-level [15, 17, 37, 44] approaches to dataflow scheduling and communication optimizations by taking a middle-ground approach. Nemo provides a policy interface that transforms an **intermediate representation (IR)** of applications to express indirect but fine control over distributed scheduling and communication.

Our decoupled system design and our DAG-based IR are similar to Musketeer [10]. The frontend design of transforming different applications written in different programming interfaces to an IR is quite similar. However, our work is complementary to Musketeer, as Musketeer focuses on

enabling the execution of its IR graph on a range of different execution runtimes, to dynamically map its IR to the runtimes where the application runs faster, while we focus on providing fine control for optimizations over the physical scheduling and communication with our IR DAG abstraction.

The SparkSQL Catalyst optimizer [1] takes as input a SparkSQL application and outputs a Spark RDD application, which Nemo can take as input. Compared to Nemo, Catalyst has more information about application semantics, but has less fine control over scheduling and communication (e.g., speculative task cloning).

Recently proposed dynamic query optimizers [24, 25] for distributed dataflow processing runtimes operate on high-level logical plans for SQL queries. Leveraging the semantics of SQL queries and the runtime information, these optimizers focus on choosing an optimal logical plan, for example, by finding an optimal join order. Nemo operates on a lower-level IR DAG that supports general dataflow processing applications and provides the methods to configure scheduling and communication methods of each data-parallel operation in the applications.

Weld [29] takes as input code that composes imperative libraries such as Pandas [26] and Numpy [38], creates a combined Weld IR program, and outputs optimized assembly code using LLVM. Weld can reduce data movement overheads across such imperative libraries, but it is not designed to optimize distributed scheduling and communication like Nemo.

Works similar to those proposed by Elseidy et al. [9] suggest new operators, such as a new join operator that is scalable and adaptive to online statistics, that performs task-level optimizations. Nemo performs optimizations on top of the existing operators through annotations, instead of developing a new one, by defining the runtime actions for scheduling and communication on the IR DAG, as well as by inserting new control logics by inserting Nemo utility vertices.

SpongeFiles [8] suggest its mechanisms to store large chunks of data in different locations in the cluster and to spill the data to the nearest location with sufficient capacity (local memory/remote memory/local disk/remote disk, in the given order). While Nemo currently provides support for local memory and disks, it can be extended to Crail [41], SpongeFiles [8], and other storages to improve on its performance. Such works can be supported and added on to our list of optimizations for further improvements.

Relational databases: Many of the optimizations in Nemo, such as parallelization and distributed scheduling optimizations, can be traced to research in parallel databases [6, 11]. Nemo enables expressing and composing various types of such optimizations for distributed dataflow processing applications by introducing a policy interface that provides fine control and at the same time ensures correctness.

Our idea of annotating operators with execution properties is similar to using query hints in relational databases to influence the optimizer [4]. Nevertheless, these works focus on restricting the search space of SQL query execution plans, whereas Nemo focuses on tuning the scheduling and communication of dataflow processing applications.

Recent database systems like SageDB [19] enables the DB to take a learning-based model to specialize its workload on the data distribution, workload, and hardware environments to tune DB components such as index structures, sorting algorithms, and the query executor. In Nemo, instead of replacing the core components with the learned components in our system, we aim to optimize the scheduling and the communication mechanisms within the boundary of our APIs and the runtime modules that we define and implement, while it can be our future work to take a learning-based approach to automatically generate a policy for scheduling and communication.

Compilers: Our approach of expressing optimizations as passes that transform an IR is similar to LLVM [22]. However, in contrast to the LLVM IR that represents assembly code, the Nemo IR explicitly captures the dependencies and the communication patterns of coarse-grained,

data-parallel operations. This enables passes on Nemo to express various distributed scheduling and communication optimizations.

Duboscq et al. [7] suggest a graph-based IR for speculative optimizations in a dynamic compiler, where it deoptimizes the machine code to create an IR to allow dynamic speculative optimizations for high-level languages in the compiler. In constrast to the approach mentioned above, Nemo optimizes the application by annotating the optimized runtime actions on the IR DAG and executing it as instructed by the Nemo optimizations, instead of by manually optimizing the control flow and the compiler optimization performance.

Verified compilers, such as CompCert [23], aim to ensure the correctness of optimized assembly code using formal verification methods. Nemo aims to ensure the correctness of optimized distributed execution of dataflow processing applications by introducing utility vertices and execution properties that make it simple to ensure correctness.

7 DISCUSSION

Nemo provides a programming interface for building correct, reusable, and composable optimization policies. We discuss several directions to extend the interface and further facilitate the development of new policies.

Ensuring resource constraints: Although Nemo provides execution properties to specify where to place computations and data, Nemo relies on the runtime to determine the actual resources to acquire. To ensure that the resource constraints are met in the execution, we can incorporate the information into the IR DAG on the resource availability and acquisition or pre-launch the execution DAG on a small set of data to test the runtime actions.

Declaring optimizations ahead of time: To enable compile-time analysis of runtime pass conflicts and optimizations, we can provide the option to declare intended optimizations ahead of time. For example, we can receive more explicit information on the predicates (e.g., is a shuffle edge) and actions (e.g., store in memory) that a runtime pass intends to use.

Leveraging historical information: We can enable passes to use information on previous executions of the same application and employ more sophisticated techniques such as machine learning to determine how to transform the IR DAG. To facilitate this, we can maintain a database that stores the information of the executed IR DAGs and their performance metrics and provide an interface for passes to access the information in the database.

8 CONCLUSION

We presented Nemo, an optimization framework that provides fine control over distributed scheduling and communication of data-processing applications and at the same time ensures correct application semantics. We hope Nemo serves as a platform for data processing optimization research and development. Nemo is available at https://nemo.apache.org.

ACKNOWLEDGMENTS

We thank the members of the Software Platform Lab at Seoul National University for their valuable input.

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the ACM SIGMOD Conference*.

[2] Bert Hubert. 2020. Linux Traffic Control. Retrieved from https://lartc.org/manpages/tc.txt.

[3] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the European Conference on Computer Systems (EuroSys'18)*.

[4] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Power hints for query optimization. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'09)*.

[5] CAIDA. 2020. The CAIDA Anonymized Internet Traces 2016 Dataset. Retrieved from https://www.caida.org/data/passive/passive_2016_dataset.xml.

[6] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The gamma database machine project. *IEEE Trans. Knowl. Data Eng.* 2, 1 (Mar. 1990), 44–62.

[7] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL'13)*.

[8] Khaled Elmeleegy, Christopher Olston, and Benjamin Reed. 2014. SpongeFiles: Mitigating data skew in MapReduce using distributed memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*.

[9] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 441–452.

[10] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: All for one, one for all in data processing systems. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*.

[11] Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[12] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting code optimizations in data-parallel pipelines through periSCOPE. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[14] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching LAN speeds. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*.

[16] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. 2011. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.* 4, 6 (Mar. 2011), 385–396.

[17] Qifa Ke, Michael Isard, and Yuan Yu. 2013. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the European Conference on Computer Systems (EuroSys'13)*.

[18] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (Aug. 2009), 30–37.

[19] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'19)*.

[20] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2010. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the ACM Symposium on Cloud Computing*.

[21] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: Mitigating skew in MapReduce applications. In *Proceedings of the ACM SIGMOD Conference*.

[22] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*.

[23] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115.

[24] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. 2018. RIOS: Runtime integrated optimizer for Spark. In *Proceedings of the ACM Symposium on Cloud Computing*.

[25] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. 2018. Dynamic query re-planning using QOOP. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[26] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*.

[27] Microsoft. 2020. Dryad Research Prototype. Retrieved from https://github.com/MicrosoftResearch/Dryad.
[28] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the ACM Symposium on Operating Systems Principles*.
[29] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'17)*.
[30] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. In *Proceedings of the ACM SIGCOMM Conference*.
[31] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. 2012. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proceedings of the ACM Symposium on Cloud Computing*.
[32] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A framework for large scale data processing. In *Proceedings of the ACM Symposium on Cloud Computing*.
[33] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. 2012. Themis: An i/o-efficient MapReduce. In *Proceedings of the ACM Symposium on Cloud Computing*.
[34] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*.
[35] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2020. *Google Cluster-usage Traces: Format + Schema*. Technical Report. Google. Retrieved from https://github.com/google/cluster-data.
[36] Peter J. Rousseeuw and Gilbert W. Bassett Jr. 1990. The Remedian: A robust averaging method for large data sets. *J. Amer. Statist. Assoc.* 85, 409 (Mar. 1990), 97–104.
[37] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. Apache Tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the ACM SIGMOD Conference*.
[38] SciPy.org. 2020. NumPy. Retrieved from https://www.numpy.org.
[39] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. 2016. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the European Conference on Computer Systems (EuroSys'16)*.
[40] The Apache Software Foundation. 2020. Apache Beam. Retrieved from https://beam.apache.org.
[41] The Apache Software Foundation. 2020. Apache Crail. Retrieved from https://crail.apache.org/.
[42] The Apache Software Foundation. 2020. Apache Flink. Retrieved from https://flink.apache.org/.
[43] The Apache Software Foundation. 2020. Apache Hadoop. Retrieved from https://hadoop.apache.org.
[44] The Apache Software Foundation. 2020. Apache Spark. Retrieved from https://spark.apache.org.
[45] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive—A petabyte scale data warehouse using Hadoop. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'10)*.
[46] TPC. 2020. TPC-H. Retrieved from http://www.tpc.org/tpch.
[47] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-aware optimization for analytics queries. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.
[48] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global analytics in the face of bandwidth and regulatory constraints. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.
[49] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russel Sears, Beysim Sezgin, and Julia Wang. 2015. REEF: Retainable evaluator execution framework. In *Proceedings of the ACM SIGMOD Conference*.
[50] Wikimedia. 2020. Page view statistics for Wikimedia projects. Retrieved from https://dumps.wikimedia.org/other/pagecounts-raw.
[51] Yahoo! 2020. Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0. Retrieved from https://webscope.sandbox.yahoo.com/catalog.php?datatype=r.
[52] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient computing for big data analytics. In *Proceedings of the ACM Symposium on Cloud Computing*.
[53] Youngseok Yang, Jeongyoon Eo, Geon-Woo Kim, Joo Yeon Kim, Sanha Lee, Jangho Seo, Won Wook Song, and Byung-Gon Chun. 2019. Apache Nemo: A framework for building distributed dataflow optimization policies. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*.
[54] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the European Conference on Computer Systems (EuroSys'17)*.

[55] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation.*

[57] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. 2018. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the European Conference on Computer Systems (EuroSys'18)*.

[58] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation.*

Received July 2020; revised March 2021; accepted May 2021