PEEL: A Framework for Benchmarking Distributed Systems and Algorithms

Christoph Boden[1,2](✉), Alexander Alexandrov[1], Andreas Kunft[1], Tilmann Rabl[1,2], and Volker Markl[1,2]

[1] Technische Universität Berlin, Berlin, Germany
{christoph.boden,alexander.alexandrov,andreas.kunft,tilmann.rabl, volker.markl}@tu-berlin.de
[2] DFKI, Saarbrücken, Germany

Abstract. During the last decade, a multitude of novel systems for scalable and distributed data processing has been proposed in both academia and industry. While there are published results of experimental evaluations for nearly all systems, it remains a challenge to objectively compare different system's performance. It is thus imperative to enable and establish benchmarks for these systems. However, even if workloads and data sets or data generators are fixed, orchestrating and executing benchmarks can be a major obstacle. Worse, many systems come with hardware-dependent parameters that have to be tuned and spawn a diverse set of configuration files. This impedes portability and reproducibility of benchmarks. To address these problems and to foster reproducible and portable experiments and benchmarks of distributed data processing systems, we present *PEEL*, a framework to define, execute, analyze, and share experiments. PEEL enables the transparent specification of benchmarking workloads and system configuration parameters. It orchestrates the systems involved and automatically runs and collects all associated logs of experiments. PEEL currently supports Apache HDFS, Hadoop, Flink, and Spark and can easily be extended to include further systems.

1 Introduction and Motivation

During the last decade, the Big Data hype has led to the development of a plethora of novel systems for scalable data processing. Starting with the *MapReduce* paradigm [10] and its open-source implementation *Hadoop* [3], numerous successors have been proposed and implemented either as research prototypes or industry led open-source systems. Hadoop MapReduce was quickly embraced by practitioners, as it successfully abstracts away the complexity of scheduling a program's distributed execution on large clusters, managing the inter-machine communication as well as coping with machine failures by exposing a simple functional programming API to users. However, as focus shifted from rather simple extraction and aggregation jobs to the scalable execution of more complex workflows, such as inferring statistical models and machine learning algorithms, it quickly became apparent that Hadoop was inherently inefficient at executing such workloads. While many machine learning algorithms can easily be formulated in the functional *MapReduce* programming model on a logical level [9], the acyclic data flow model underlying Hadoop's implementation and the intricacies of its distributed implementation lead to unsatisfactory performance. Particularly the fixed *Map-Shuffle-Reduce* pipeline and the inability to efficiently execute *iterative computations* turned out to be major drawbacks of Hadoop.

This led to numerous contributions from both the database and systems communities, which address these shortcomings. Systems such as Spark [4,16] or Stratosphere [5] (now called Flink [1,8]) were among the first systems to support efficient iterative computations, GraphLab [12] proposed an asynchronous graph-based execution model, which was subsequently distributed [11]. Pregel [13] and its open source implementation Giraph [2] provided a vertex-centric programming abstraction and Bulk Synchronous Parallel (BSP) based execution model.

While nearly all systems have been presented in scientific publications containing an experimental evaluation, it remains a challenge to objectively compare the performance of each system. Different workloads and implementations, usage of libraries, data sets and hardware configurations make it hard if not impossible to leverage the published experiments for such a comparison. Furthermore, it is a challenge to assess how much of the performance gain is due to a superior paradigm or design and how much is due to a more efficient implementation, which ultimately impairs the scientific process due to a lack of verifiability.

For this reason, it is imperative to enable and establish benchmarks for big data analytics systems. However, even if workloads and data sets or data generators are fixed, orchestrating and executing benchmarks can be a major challenge.

The principle goal of a system experiment as part of such a benchmark is to characterize the behavior of a particular system under test (SUT) for a specific set of values, configured as system and application parameters. The usual way to achieve this goal is to

1. define a workload application which takes specific parameters (e.g., input and output path),
2. run it on top of the SUT with a specific configuration (e.g., allocated memory, degree of parallelism (DOP)) and,
3. measure key performance characteristics (e.g., runtime, throughput, accuracy).

Achieving these goals for benchmark experiments on modern data management systems such as distributed data processing systems is significantly more complex than evaluating a traditional RDBMS. Figure 1 illustrates, that rather than having a single system under test running in isolation, novel data processing systems require several systems such as a distributed file system and a distributed data processing system to be executed jointly. Current trends actually advocate an architecture based on interconnected systems (e.g. HDFS, Yarn, Spark, Flink, Storm). Each of these systems has to be set up and launched with their own set of, potentially hardware-dependent, configurations.

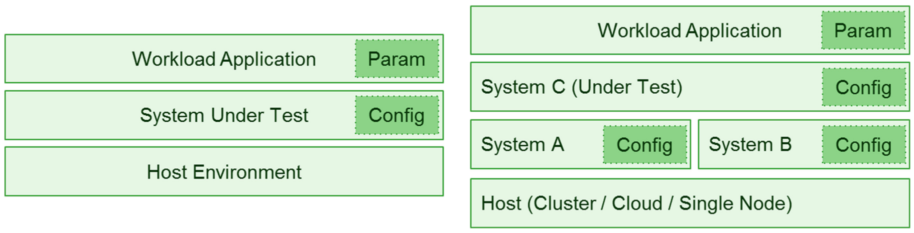

Fig. 1. The general setup: contrary to the setup when benchmarking traditional RDMB systems (left) where we evaluate only one System under Test (SUT), the landscape is more complicated when evaluating novel distributed data processing frameworks (right), as they usually require the interplay between multiple independent systems. Each of these systems has its own configurations with sets of parameters that have to be set, and potentially tuned.

Typically, one is not just interested in the insights obtained by a single experiment, but in trends highlighted by a suite of experiments where a certain system under test (SUT) configuration or application parameter value is varied and everything else remains fixed. When running a scale-out experiment with a varying number of nodes for example, the configuration of both the distributed file system as well as the distributed data processing system under test have to be changed, and the systems have to be appropriately orchestrated. This further complicates the benchmarking process. Additionally, hardware-dependent configurations hinder portability and thus reproducibility of benchmarks. When adjusting the systems is done manually, huge amounts of temporary files and generated data tend to clog up the disk space, as experiments may be run without proper tear-down of systems and cleaning of temporary directories. When such experiments are run on a shared cluster, as is often the case in an academic environment, this issue becomes even more severe.

Contribution: To address these problems and to enable and foster reproducible experiments and benchmarks of distributed data processing systems, we present *PEEL*[1], a framework to define, execute, analyze, and share experiments. On the one hand, PEEL automatically orchestrates experiments and handles the systems' setup, configuration, deployment, tear-down and cleanup as well as automatic log collection. On the other hand, PEEL introduces a unified and transparent way of specifying experiments, including the actual application code, system configuration, and experiment setup description. With this transparent specification, PEEL enables the sharing of end-to-end experiment artifacts, thus fostering reproducibility and portability of benchmark experiments. PEEL also allows for the hardware independent specification of these parameters, therefore enabling portability of experiments across different hardware setups. Figure 2 illustrates the process enabled by our framework. Finally, we make PEEL available as open-source software on GitHub.

[1] https://github.com/peelframework/peel.

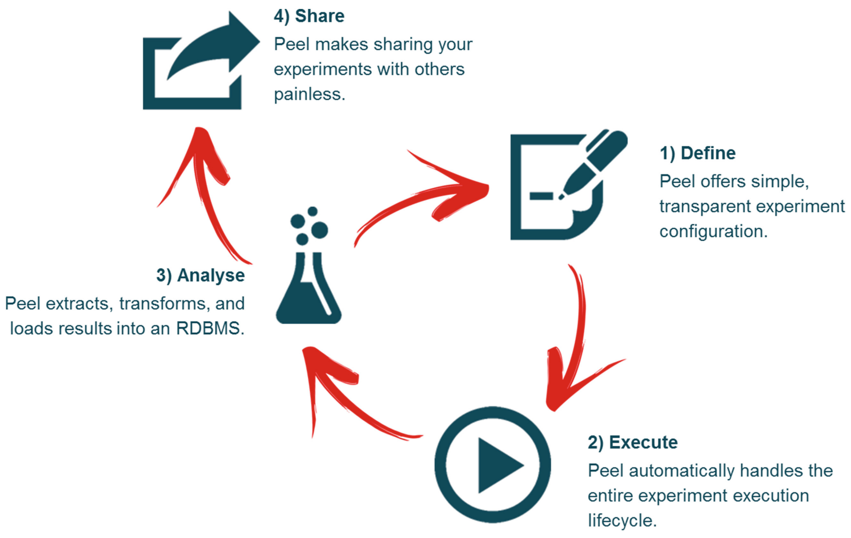
4) Share
Peel makes sharing your experiments with others painless.
1) Define
Peel offers simple, transparent experiment configuration.
2) Execute
Peel automatically handles the entire experiment execution lifecycle.
3) Analyse
Peel extracts, transforms, and loads results into an RDBMS.

Fig. 2. The Peel process: Peel enables the transparent specification of workloads, systems configurations, and parameters to be varied. It also automatically handles the distributed orchestration and execution as well as sharing of these experiment bundles. After successfully running all experiments in a bundle, PEEL automatically extracts, transforms and loads relevant measurements from collected log files and makes them available in an RDBMS.

2 Outline

The rest of this paper is structured as follows: In Sect. 3 we introduce the running example of a supervised machine learning workload, which we will use to explain the details of defining an experiment. Section 4 introduces experiment definitions. Next, Sect. 5 describes the basics of a bundle. Section 6 discusses the approach of a unified, global experiment environment configuration and Sect. 7 illustrates how PEEL bundles can be deployed and executed on cluster environments, before Sect. 8 provides an overview, how results can be gathered and analyzed within the framework. Finally, we describe how PEEL can be extended with additional systems in Sect. 9.

3 Running Example: Benchmarking a Supervised Machine Learning Workload

As a running example, we will consider a supervised machine learning workload. This workload is part of an extensive machine learning benchmark for distributed data flow systems [7] (please see the paper for further details and experimental results).

More concretely, we train a logistic regression model for *click-through rate prediction* using a batch gradient descent solver. Click-through rate prediction for online advertisements forms a crucial building block in the multi-billion dollar online advertising industry and logistic regression models have their merit for this task [14,15]. Such prediction models are trained on hundreds of terabytes of data with hundreds of billions of training samples, which happen to be very high dimensional. For this reason, we are interested in evaluating the scaling behavior of the systems not just with respect to growing data set sizes and increasing number of compute nodes but also with respect to increasing model dimensionality as suggested in [7].

For the experiments, we use a subset of the *Criteo Click Logs*[2] data set. This dataset contains feature values and click feedback for millions of ad impressions drawn from a portion of Criteo's traffic over a period of 24 days. Since this data set contains categorical features, we use feature hashing as a pre-processing step. Feature hashing (also called the hashing trick) vectorizes the categorical variables by applying a hash function to the feature values and using the hash values as indices. It thus maps the sparse training data into a space with fixed

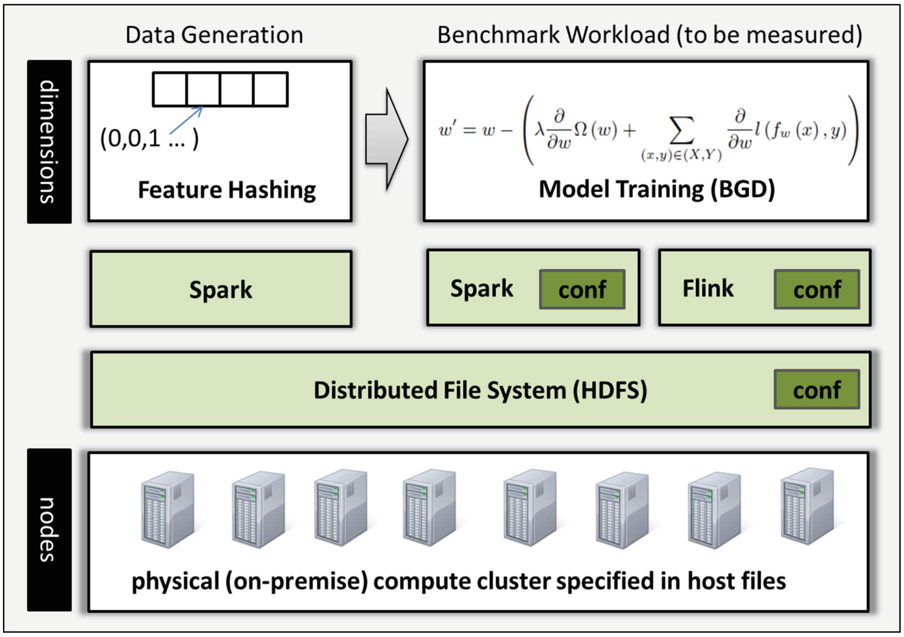
Data Generation
Benchmark Workload (to be measured)
dimensions
(0,0,1 ...)
Feature Hashing
$w' = w - \left(\lambda \frac{\partial}{\partial w}\Omega(w) + \sum_{(x,y)\in(X,Y)} \frac{\partial}{\partial w} l(f_w(x), y)\right)$
Model Training (BGD)
Spark
Spark
conf
Flink
conf
Distributed File System (HDFS)
conf
nodes
physical (on-premise) compute cluster specified in host files

Fig. 3. An illustration of the running example: we evaluate *batch gradient decent* training of a supervised learning model as a workload on Apache Spark and Apache Flink as *systems under test*. The individual experiments depend on two parameters: the number of physical compute *nodes* and the dimensionality of the training data set (*dimensions*), which specify how the benchmark workload shall be executed. The data generation job is executed on a system independent of the system under test.

[2] http://labs.criteo.com/downloads/download-terabyte-click-logs/.

dimensionality, which can be controlled by a parameter. This feature hashing will be executed as a pre-processing job on Spark.

Figure 3 illustrates the running example. We want to evaluate *batch gradient decent* training of a supervised learning model as a workload on Apache Spark and Apache Flink as a *system under test*. Only for this part do we want to time the execution and record the system performance characteristics of the individual compute nodes involved. In order to evaluate the scalability of the systems with respect to a varying number of compute nodes as well as a varying dimensionality of the training data (and thus also the model to be trained) the two parameters: *nodes* and *dimensions* have to be varied accordingly.

For each particular configuration of physical compute nodes, a new distributed file system (HDFS) has to be set up. Next the raw criteo data, which is ingested from some external storage, has to be transformed to feature vectors of the desired dimensionality via feature hashing. Any system may be used for this step - independent of the actual system under test. In the example, we choose to run a Spark Job, which writes out the experimentation data into the temporary HDFS instantiated for the current (cluster) configuration. Next, the actual system under test (Spark or Flink in our example) will have to be setup and instantiated with its proper configuration. Once it is up and running, the benchmark workload can be submitted as a job to the system under test and its execution is timed. In order to record the performance characteristics on the individual compute nodes, an additional monitoring system such as *dstat* will have to be started on all compute nodes.

After successful execution, the system under test will have to be shut down. In order to archive all aspects of the benchmark experiments, various logs of the different systems involved (dstat, system under test) will have to be gathered from all the compute nodes. Next, all temporary directories have to be cleaned, and the next system has to be set up and instantiated. Once all systems have been evaluated for a concrete dimensionality, the data set has to be deleted from the distributed file system and the next one, with a new dimensionality, has to be created. When all parameter settings for a particular Node configuration (i.e. all dimensionalities) have been evaluated, the distributed file system will have to be torn down and a new one, with a different node configuration, will have to be set up.

Manually administering all these steps is a tedious and error-prone process as jobs can run for long periods of time, and may fail. PEEL automatically takes care of all the steps outlined above and thus reduces the operational complexity of benchmarking distributed data processing systems significantly. In the following sections, we will explore how a workload like the supervised learning example has to be specified in PEEL in order to benefit from this automation.

4 Experiment Definitions

Experiments are defined using a Spring dependency injection container as a set of inter-connected beans. Figure 4 displays the available bean types as a

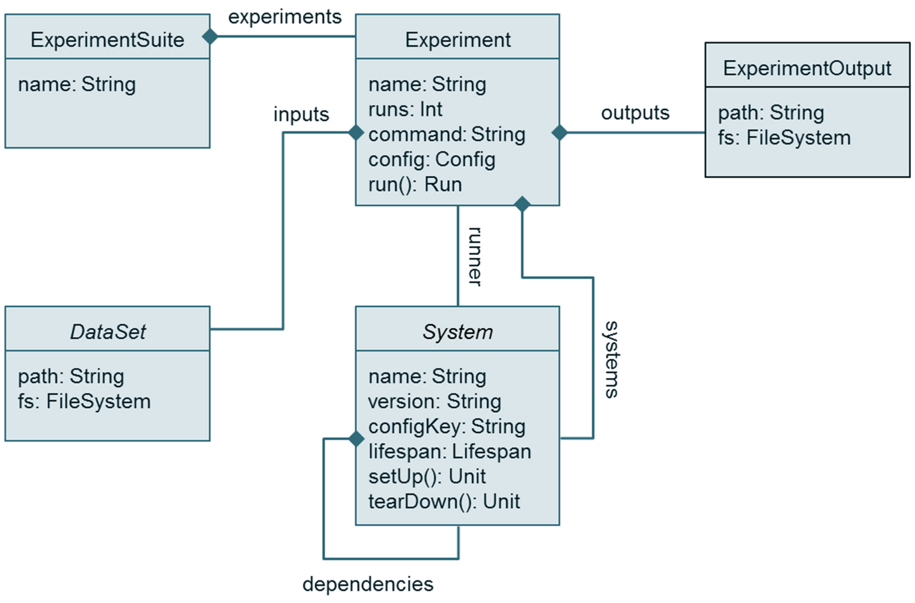

Fig. 4. A domain model of the PEEL experiment definition elements.

domain model. Beans definitions can be done either in XML or in annotated Scala classes. Scala is used for all examples in this paper. The beans required to define an experiment realize the system experiments domain introduced in the Motivation section.

We discuss the individual beans in light of our example of running *batch gradient descent* training of a logistic regression model for click-through rate prediction. Listing 1.1 shows the complete experiment definition of the definition of our example.

Listing 1.1. The main experiment definition of our running example.

```
class ExperimentsDimensionScaling extends ApplicationContextAware {
  var ctx: ApplicationContext = null
  def setApplicationContext(ctx: ApplicationContext): Unit = {
    this.ctx = ctx
  }

  def sparkFeatureHashing(i: String, numF: Int, perc: Double): SparkJob = new SparkJob(
    timeout = 10000L,
    runner = ctx.getBean("spark-1.6.2", classOf[Spark]),
    command =
      s"""
        |--class dima.tu.berlin.generators.spark.SparkCriteoExtract \\
        |$${app.path.datagens}/peel-bundle-datagens-1.0-SNAPSHOT.jar \\
        |--inputPath=$i \\
        |--outputPath=$${system.hadoop-2.path.input}/train/$numF/$perc \\
        |--numFeatures=$numFeatures \\
        |--percDataPoints=$perc \\
      """.stripMargin.trim
  )
  def ‘bgd.output‘: ExperimentOutput = new ExperimentOutput(
21      path = "{system.hadoop-2.path.input}/benchmark/",
22      fs  = ctx.getBean("hdfs-2.7.1", classOf[HDFS2])
23    )
24    def `bgd.input`(D: Int): DataSet = new GeneratedDataSet(
25      src = sparkFeatureHashing("/criteo/", numFeatures, perc),
26      dst = s"$${system.hadoop-2.path.input}/train/" + numFeatures + "/" + perc,
27      fs = ctx.getBean("hdfs-2.7.1", classOf[HDFS2])
28    )
29    def `bgd.flink`(D: Int, N: String) = new FlinkExperiment(
30      name   = s"flink.train.$D",
31      command =
32        s"""
33           |--class dima.tu.berlin.benchmark.flink.mlbench.NewLogReg \\
34           |$${app.path.apps}/peel-bundle-flink-jobs-1.0-SNAPSHOT.jar \\
35           |--trainDir=$${system.hadoop-2.path.input}/train/$D \\
36           |--outputDir=$${system.hadoop-2.path.input}/benchmark/$N/$D/flink/ \\
37           |--degOfParall=$${system.default.config.parallelism.total} \\
38           |--dimensions=$D \\
39         """.stripMargin.trim,
40      config = ConfigFactory.parseString(
41        s"""
42           |system.default.config.slaves        = $${env.slaves.$N.hosts}
43           |system.default.config.parallelism.total = $${env.slaves.$N.total.parallelism}
44         """.stripMargin.trim),
45      runs   = 3,
46      runner = ctx.getBean("flink-1.0.3", classOf[Flink]),
47      systems = Set(ctx.getBean("dstat-0.7.2", classOf[Dstat])),
48      inputs = Set(`bgd.input`(D), classOf[DataSet])),
49      outputs = Set(`bgd.output`)
50    )
51    def `bgd.spark`(D: Int, N: String) = new SparkExperiment(
52      name   = s"spark.train.$D",
53      command =
54        s"""
55           |--class dima.tu.berlin.benchmark.spark.mlbench.RUN \\
56           |$${app.path.apps}/peel-bundle-spark-jobs-1.0-SNAPSHOT.jar \\
57           |--trainDir=$${system.hadoop-2.path.input}/train /$D \\
58           |--outputDir=$${system.hadoop-2.path.input}/benchmark/$N/$D/spark \\
59           |--numSplits=$${system.default.config.parallelism.total} \\
60         """.stripMargin.trim,
61      config = ConfigFactory.parseString(
62        s"""
63           |system.default.config.slaves        = $${env.slaves.$N.hosts}
64           |system.default.config.parallelism.total = $${env.slaves.$N.total.parallelism}
65         """.stripMargin.trim),
66      runs   = 3,
67      runner = ctx.getBean("spark-1.6.2", classOf[Spark]),
68      systems = Set(ctx.getBean("dstat-0.7.2", classOf[Dstat])),
69      inputs = Set(`bgd.input`(D), classOf[DataSet])),
70      outputs = Set(`bgd.output`)
71    )
72    def `bgd.dimensions.scaling`: ExperimentSuite = new ExperimentSuite(
73      for {
74        Dims <- Seq(10, 100, 1000, 10000, 100000, 1000000)
75        Nodes <- Seq("top020", "top010", "top005")
76        Exps <- Seq(`bgd.spark`(Dims, Nodes), `bgd.flink`(Dims, Nodes))
77      } yield Exps
78    )
79  }
80  @Bean(name = Array("bgd.dimensions.scaling"))
```

Experiment: The central class in the domain model shown in Fig. 4 is *Experiment*. In our example definition in Listing 1.1 we specify two Experiments: one for Flink (lines 29–50) and one for Spark (lines 51–71). Each experiment specifies the following properties: the experiment name, the command that executes the experiment's job, the number of runs (repetitions) the experiment is executed, the inputs required and outputs produced by each run, the runner system that carries the execution, other systems, upon which the execution of the experiment depends (e.g. *dstat* in line 47 for monitoring the resource usage on the compute nodes) as well as the experiment-specific environment config which is discussed in Sect. 6.

System: The second important class in the model is *System*. It specifies the following properties: the system name, usually fixed per System implementation, e.g. flink for the Flink system or spark for the Spark system, the system version (e.g. 1.0.3 for Flink or 1.6.2 for Spark), a configKey under which config parameters will be located in the environment configuration, usually the same as the system name, a Lifespan value (one of *Provided*, *Suite*, *Experiment*, or *Run*) which indicates when to start and stop the system and a list of systems upon which the current system depends.

ExperimentSuite: A series of related experiment beans are organized in an *ExperimentSuite*. In our example listing, we define an ExperimentSuite in lines 72–78. Recall that our original motivation was to compare the scale-out characteristics of Spark and Flink with respect to both: scaling the nodes and scaling the model size. To accomplish this, we vary two parameters: *Dims* which specifies the dimensionality of the training data and *Nodes*, which refers to a list of hosts the experiment should run on. The for-comprehension creates a cartesian product of all parameter values and the two experiments. With this, we ensure that we only generate a new data set whenever either the node configuration or the desired dimensionality changes, but not for each experiment separately. Experiments typically depend on some kind of input data, represented as abstract *DataSet* elements associated with a particular *FileSystem* in our model. The following types are currently supported:

- *CopiedDataSet* - used for static data copied into the target FileSystem;
- *GeneratedDataSet* - used for data generated by a Job into the target FileSystem.

In the example we rely on a `GeneratedDataSet` to trigger the Spark job for feature hashing (lines 24–28). In addition, each experiment bean is associated with an *ExperimentOutput* which describes the paths the data is written to by the experiment workload application (lines 20–23). This meta-information is used to clean those paths upon execution.

5 Bundle Basics

A *bundle* packages together the configuration data, datasets, and workload jobs required for the execution of a particular set of experiments. Table 1 provides an overview of the top-level elements of such a bundle. It is self-contained and can be pushed to a remote cluster for execution as well as shared for reproducibility purposes. The main components of a bundle can be grouped as follows:

Table 1. The top-level elements of a bundle. (Non-fixed paths can be customized.)

Default path	Config parameter	Fixed	Description
./apps	app.path.apps	Yes	Workload applications
./config	app.path.config	Yes	configurations and experiment definitions.
./datagens	app.path.datagens	No	Data generators
./datasets	app.path.datasets	No	Static datasets
./downloads	app.path.downloads	No	Archived system binaries
./lib	app.path.log	Yes	Peel libraries and dependencies
./log	app.path.log	Yes	Peel execution logs
./results	app.path.results	No	State and log data from experiment runs
./systems	app.path.systems	No	Contains all running systems
./utils	app.path.utils	No	Utility scripts and files
./peel.sh	app.path.cli	Yes	The Peel command line interface

At the center of a bundle is the *PEEL command line tool (PEEL CLI)*, which provides the basic functionality of PEEL. While running, the Peel CLI spawns and executes OS processes. It can be used to start and stop experiments, and to push and pull bundles to and from remote locations. The *log* folder contains the `stdout` and `stderr` output of these processes, as well as a copy of the actual console output produced by PEEL itself. The *config* folder contains *.conf files written in HOCON[3] syntax which defines the environment configuration, as well the actual experiments defined in scala. The *apps* folder contains the binaries of the experiment workload applications. The *datasets folder* contains static, fixed-sized datasets required for the experiments. The datagens folder contains programs for dynamic generation of scalable datasets required for the experiments. The *downloads* folder contains system binary archives for the systems in the experiment environment. The archives are per default extracted in the *systems* folder. The *results* folder contains all the data collected from attempted and successful Peel experiment runs in a hierarchy following `$suite/$expName.run$NN` naming convention. Finally, the *utils folder* contains utility scripts (e.g., SQL queries and gnuplot scripts) that can be used next to or in conjunction with Peel CLI commands. A PEEL bundle is the unit to be shared when making available benchmarks that utilize the framework.

6 Environment Configurations

Environments are instantiated with a concrete set of configuration values (for the systems) and parameter values (for the experiment application). A number of problems can arise with a naïve approach for manual configuration (per system and experiment) of the environments:

[3] https://github.com/typesafehub/config/blob/master/HOCON.md.

Syntax Heterogeneity. Each system (HDFS, Spark and Flink) has to be configured separately using its own special syntax. This requires basic understanding and knowledge in the configuration parameters for all systems in the stack. (For example, the number of processing slots is called `spark.executor.cores` in Spark and `taskmanager.numberOfTaskSlots` in Flink.)

Variable Interdependence. The sets of configuration variables associated with each system are not mutually exclusive. Thus, care has to be taken that the corresponding values are consistent for the overlapping fragment (e.g., the slaves list in all systems should be the same).

Value Tuning. For a series of related experiments, all but a very few set of values remain fixed. These values are suitably chosen based on the underlying host environment characteristics in order to maximize the performance of the corresponding systems (e.g., memory allocation, degree of parallelism, temp paths for spilling).

PEEL associates one global environment configuration to each experiment. In doing this, it promotes:

- configuration reuse through layering
- configuration uniformity through a hierarchical syntax

At runtime, experiments are represented by experiment beans. Each experiment bean holds a *HOCON* config that is first constructed and evaluated based on the layering scheme and conventions discussed below, and then mapped to the various concrete config and parameter files and formats of the systems and applications in the experiment environment.

In our running example, this means that for varying the number of nodes between three different configurations (20, 10, and 5 nodes) - each of the six experiments (3x SparkBGD + 3x FlinkBGD) will have an associated config property - a hierarchical map of key-value pairs which constitute the configuration of all systems and jobs required for that particular experiment. This is illustrated in Fig. 5.

Configuration Layers. The configuration system is built upon the concept of layered construction and resolution. Peel distinguishes between three layers of configuration:

- **Default.** Default configuration values for Peel itself and the supported systems. Packaged as resources in related jars located in the bundle's `app.path.lib` folder.
- **Bundle.** Bundle-specific configuration values. Located in `app.path.config`. Default is the *config* subfolder of the current bundle.
- **Host.** Host-specific configuration values. Located in the `$HOSTNAME` subfolder of the `app.path.config` folder.

For each experiment bean defined in an experiment suite, an associated configuration will be constructed according to the entries in Table 2 (higher in the list means lower priority).

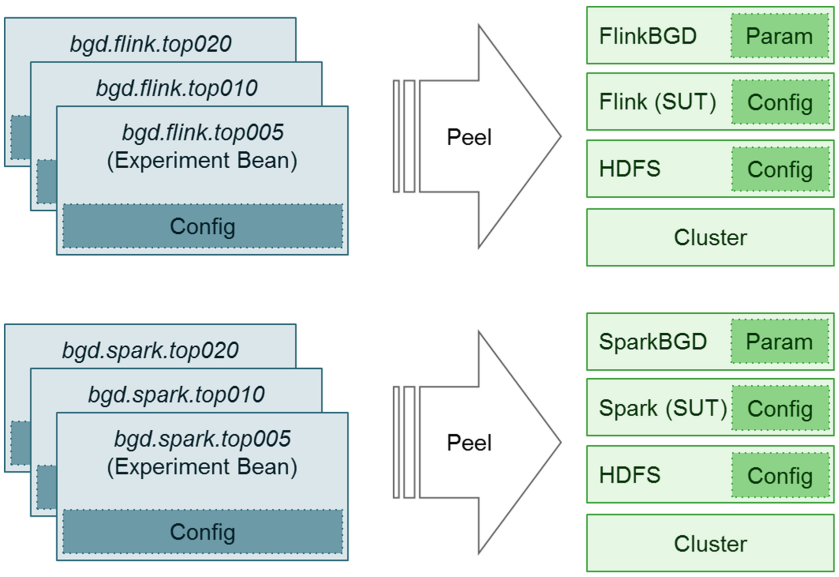

Fig. 5. Mapping the environment configurations for the six Batch Gradient Decent experiments

Table 2. Hierarchy of configurations which are associated with an experiment bean (higher in the list means lower priority).

Path	Description
`reference.peel.conf`	Default Peel config
`reference.$systemID.conf`	Default system config
`config/$systemID.conf`	Bundle-specific system config (opt)
`config/hosts/$hostname/$systemID.conf`	Host-specific system config (opt)
`config/application.conf`	Bundle-specific Peel config (opt)
`config/hosts/$hostname/application.conf`	Host-specific Peel config (opt)
Experiment bean *config* value	Experiment specific config (opt)
System	JVM system properties (constant)

First comes the *default* configuration, located in the `peel-core.jar` package. Second, for each system upon which the experiment depends (with corresponding system bean identified by `systemID`), PEEL tries to load the default configuration for that system as well as bundle- or host-specific configurations.

Third, bundle- and host-specific `application.conf`, which is a counterpart and respectively overrides bundle-wide values defined in `reference.peel.conf`.

Above follow the values defined the `config` property of the current experiment bean. These are typically used to vary one particular parameter in a sequence of experiments in a suite (e.g. varying the number of workers and the DOP).

Finally, a set of configuration parameters derived from the current JVM System object (e.g., the number of CPUs or the total amount of available memory) are appended.

7 Execution Workflow

In the previous Sections we explained the internals and the code required to configure the environment and define the experiments in a PEEL bundle. In this section, we will explain how to make use of the commands provided by the Peel CLI in order to deploy and run the experiments in a bundle.

As a first step, the bundle has to be assembled from the sources with `mvn deploy`. For large-scale applications, the environment where the experiments need to be executed typically differs from the environment of the machine where the bundle binaries are assembled. In order to start the execution process, the user therefore needs to first deploy the bundle binaries from the local machine to the desired host environment. The Peel CLI offers a special command for this. In order to push the peel-bundle to the remote cluster, one has to run: `./peel.sh rsync:push remote-cluster-name`. The command uses rsync to copy the contents of the enclosing Peel bundle to the target environment. The connection options for the rsync calls are thereby taken from the environment configuration of the local environment. The remote environment has to be specified in the `application.conf`.

As explained above, PEEL organizes experiments in sequences called *experiment suites*. The easiest option is to start an entire suite via `./peel.sh suite:run` which will automatically step through the entire execution lifecycle for each experiment:

- **Setup Experiment.** Ensure that the required inputs are materialized (either generated or copied) in the respective file system. Check the configuration of associated descendant systems with *provided* or *suite* lifespan against the values defined in the current experiment config. If the values do not match, it reconfigures and restarts the system. Set up systems with *experiment* lifespan.
- **Execute Experiment.** For each experiment run which has not been completed by a previous invocation of the same suite: Check and set up systems with *run* lifespan, execute experiment run, collect log data from the associated systems and clear the produced outputs.
- **Tear Down Experiment.** Tear down all systems with *experiment* lifespan.

Next to simply running a Full Suite which automatically executes all experiments specified, each of the above steps can be executed individually. This is particularly useful when developing and debugging a benchmark, as it allows to validate that each step is executed correctly.

Since PEEL also keeps track of failed experiments, one can simply re-run an entire suite in order to re-attempt the execution of the failed experiments. PEEL will automatically skip all experiments, which have already been successfully run.

8 Results Analysis

The results of all experiments are stored in a folder structure which contains log file data collected from the systems involved in the experiment. In order to make sense of the data, Peel ships with an extensible ETL pipeline that extracts relevant data from the log files, transforms it into a relational schema, and loads it into a database. One can then analyze various aspects of the obtained results by querying the underlying result schema with SQL statements.

The experiment suite defined by the running example in Sect. 3 will produce results similar to Table 3. (for detailed experimental results please see [7])

Table 3. Exemplary table listing the results of experiment runs.

Experiment	Nodes	Dimensions	Runtime in ms
flink.train	top023	10	165612
flink.train	top023	100	265034
flink.train	top023	1000	289115
flink.train	top023	10000	291966
flink.train	top023	100000	300280
flink.train	top023	1000000	315500
spark.train	top023	10	128286
spark.train	top023	100	205061
spark.train	top023	1000	208647
spark.train	top023	10000	219103
spark.train	top023	100000	222236
spark.train	top023	1000000	298778
...	...	...	...

Backends. Peel supports multiple relational database engines as a possible backend for your experiment data. The decision which backend to use depends on the scope and complexity of the use case.

H2. The H2 backend is the easy and quick option for beginners. If the experiment logs are small, this is the best way to go as it requires zero overhead for setup. With the default H2 connection h2, PEEL will initialize and populate a results database in a file named h2.mov.db located in the `${app.path.results}` folder.

MonetDB. If the experiments generated a lot of data or more advanced analytics on the extracted database instance are required, we recommend using a column store like MonetDB.

Analysis. To visually explore and analyze the results of the experiments, one can connect the database schema produced by Peel with a reporting tool like JasperReports, an OLAP cube analysis tool like Pentaho, or a visual data exploration tool like Tableau.

9 Extending Peel

Currently, PEEL supports various versions of the following systems out of the box: Hadoop MapReduce, Spark, Flink, HDFS, dstat and Zookeeper. However, the framework can easily be extended. Adding support for a new system is uncomplicated and only requires the definition of system specific sub-classes for the System and Experiment base-classes that were discussed in Sect. 4. The communication between the framework and the systems is typically done by calling scripts via external processes with the abstractions provided in PEEL. Thus, the range of systems that can be supported is not strictly limited to JVM-based ones.

In order to add support for a new system, one simply has to define the startup and shutdown behavior of the system, the configuration files and their management, and the way log files are to be collected inside the system class. As was presented in the example definition in Listing 1.1, the experiment bean then defines how jobs for the system are started and which arguments are passed. For cluster configurations without a network file system, PEEL also provides utility functions to distribute the required system files among the cluster nodes, as well as the collection of log files.

10 Conclusion

In this paper we introduced PEEL as a Framework for benchmarking distributed systems and algorithms. PEEL significantly reduces the operational complexity of performing benchmarks of novel distributed data processing systems. It automatically orchestrates all systems involved, executes the experiments and collects all relevant log data. Through the central structure of a peel-bundle, a unified approach to system configurations and its experiment definitions, PEEL fosters the transparency, portability, and reproducibility of benchmarking experiments. Based on the running example of a supervised machine learning workload, we introduced all the major concepts of PEEL, including experiment definitions and its experimentation process. We have sucessfully used PEEL in practice to orchestrate the experiments published in [6,7] and hope that it will be a useful tool for many in the benchmarking community, as PEEL is freely available as open-source software available at https://github.com/peelframework/peel.

Acknowledgments. This work has been supported through grants by the German Science Foundation (DFG) as FOR1306 Stratosphere, the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A) and by Oracle Labs.

References

1. https://flink.apache.org/
2. https://giraph.apache.org/
3. https://hadoop.apache.org/
4. https://spark.apache.org/
5. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The stratosphere platform for big data analytics. VLDB J. **23**(6), 939–964 (2014)
6. Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., Markl, V.: Implicit parallelism through deep language embedding. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, 31 May–4 June, 2015, pp. 47–61 (2015)
7. Boden, C., Spina, A., Rabl, T., Markl, V.: Benchmarking data flow systems for scalable machine learning. In: Proceedings of the 4th Algorithms and Systems on MapReduce and Beyond, BeyondMR 2017, pp. 5:1–5:10. ACM, New York (2017)
8. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink$^{\text{TM}}$: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
9. Chu, C.-T., Kim, S.K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Proceedings of the 19th International Conference on Neural Information Processing Systems, NIPS 2006, pp. 281–288. MIT Press, Cambridge (2006)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
11. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
12. Low, Y., Gonzalez, J.E., Kyrola, A., Bickson, D., Guestrin, C.E., Hellerstein, J.: Graphlab: a new framework for parallel machine learning. arXiv preprint arXiv:1408.2041 (2014)
13. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 135–146. ACM, New York (2010)
14. McMahan, H.B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., Chikkerur, S., Liu, D., Wattenberg, M., Hrafnkelsson, A.M., Boulos, T., Kubica, J.: Ad click prediction: a view from the trenches. In: KDD 2013. ACM (2013)
15. Richardson, M., Dominowska, E., Ragno, R.: Predicting clicks: estimating the click-through rate for new ads. In: WWW 2007. ACM (2007)
16. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI 2012 (2012)