The Power of Nested Parallelism in Big Data Processing – Hitting Three Flies with One Slap –

Gábor E. Gévay
Technische Universität Berlin
Berlin, Germany

Jorge-Arnulfo Quiané-Ruiz
Technische Universität Berlin
DFKI
Berlin, Germany

Volker Markl
Technische Universität Berlin
DFKI
Berlin, Germany

Abstract

Many common data analysis tasks, such as performing hyperparameter optimization, processing a partitioned graph, and treating a matrix as a vector of vectors, offer natural opportunities for nested-parallel operations, i.e., launching parallel operations from inside other parallel operations. However, state-of-the-art dataflow engines, such as Spark and Flink, do not support nested parallelism. Users must implement workarounds, causing orders of magnitude slowdowns for their tasks, let alone the implementation effort.

We present Matryoshka, a system that enables dataflow engines to support nested parallelism, even in the presence of control flow statements at inner nesting levels. Matryoshka achieves this via a novel two-phase flattening process, which translates nested-parallel programs to flat-parallel programs that can efficiently run on existing dataflow engines. The first phase introduces novel nesting primitives into the code, which allows for dynamic optimizations based on intermediate data characteristics in the second phase at run time. We validate our system using several common data analysis tasks, such as PageRank and K-means.

CCS Concepts

• **Information systems** → **Data model extensions**; **MapReduce languages**; **MapReduce-based systems**; • **Software and its engineering** → *Abstraction, modeling and modularity*; *Distributed programming languages*; Cloud computing.

Keywords

nested data; nested parallel collections; nested parallel operations

ACM Reference Format:
Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. The Power of Nested Parallelism in Big Data Processing – Hitting Three Flies with One Slap –. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457287

1 Introduction

The success of parallel dataflow engines, such as Spark [49, 50] and Flink [4, 17], is largely due to abstracting a dataset as an immutable, distributed collection. They process these datasets via a well-defined set of parallel operators that provide scalability and ease-of-use.

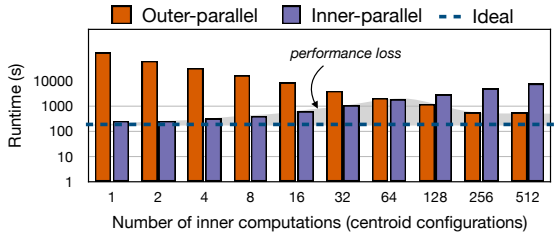


Figure 1: K-means runtimes.

Yet, many modern data analysis tasks, in domains ranging from web analytics to graph analytics and machine learning, are not well supported in these systems. Many of these modern tasks require *nested parallelism* [10], which means launching a parallel operation from the inside of another parallel operation. For instance, the user-defined function (UDF) of a `map` operator can invoke further parallel operations, such as another `map` operator.

We now explain through examples what nested parallelism is and when it occurs. First, there can be natural nesting in the data itself. For example, a nested collection might arise when treating a matrix as a vector of vectors [14] or when processing a set of graph partitions, which are themselves collections of graph vertices and edges. Second, even without nested data, a task can also be expressed by nested parallel operations. For instance, a task can perform linear algebra operations on one level [13] and hyperparameter optimization on a second, outer level [14, 43]. Third, skew in a grouping key can also raise the requirement of nested parallelism. Due to skew, there can be some large groups and also a large number of small groups. We, thus, require scalability in both the group sizes and the number of groups, i.e., on the level of processing an individual group and on the outer level of processing all groups.

As by design parallel dataflow engines do not support the nesting of parallel operations, users typically employ workarounds that parallelize on one level only. Specifically, they parallelize the task either (i) at the level of the outer collection and sequentially process an inner collection (*outer-parallel*) or (ii) the other way around (*inner-parallel*). For example, outer-parallel can use a Spark `RDD` at the outer level and an array at the inner level. An example of inner-parallel would be to use a list to sequentially try a set of hyperparameter values at the outer level and perform a machine learning model training for each value using Flink `DataSets`. Many existing systems which support nesting of operations in their languages, actually employ one of these two workarounds when executing programs [5, 14, 29, 36]. Only a few systems [20, 21, 26, 47] natively support nested parallelism. However, they do not support iterative

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457287

computations at inner nesting levels, which is a typical requirement in modern data analysis tasks, such as K-means [48].

Unfortunately, choosing between the outer- and inner-parallel workarounds for the task at hand is far from trivial. As an example, consider K-means clustering on Spark, which does not support nested parallelism. We ran K-means with a varying number of initial configurations, i.e., different sets of initial centroid values. At the same time, we also vary the computation size for each initial configuration opposite to the number of initial configurations. Therefore, we would expect the run time to be constant. Figure 1 shows the results of this experiment, with the ideal performance considered to be running on just a single initial configuration. We observe that inner-parallel (i.e., parallelizing one K-means run) is up to two orders of magnitude faster than outer-parallel (i.e., running non-parallel K-means instances in parallel with each other) for less than 64 initial configurations. This is because outer-parallel does not expose enough parallelization opportunities to utilize all the available CPU cores. The number of parallel workers is capped by the number of initial configurations. We also observe that, for more than 64 configurations, outer-parallel is up to one order of magnitude faster than inner-parallel: the latter has a high job-launch overhead because each K-means run launches new Spark jobs.

Considering these results, one might think about devising an optimizer to choose between the workarounds. However, both workarounds are far from the ideal performance (the blue line in Figure 1), with a performance gap (the gray area) of up to 6×. Note that this performance gap would increase along with the number of levels of parallel operations. For instance, adding hyperparameter optimization to choose a good value for K leads to three levels of parallelism, significantly increasing the performance gap.

We want to handle nested parallelism in a way that we are always as close as possible to the ideal case. However, devising such an approach is challenging for several reasons. First, we would like to keep existing parallel dataflow engines intact to rely on their existing code-base maturity and user base. Second, we have to avoid launching new jobs per inner collection to prevent high job-launch overheads. This implies that a large number of inner-collections must be processed within the same job, but existing parallel dataflow engines do not support this feature. Third, we have to maintain a parallelized execution on each of the inner-collections so that we expose enough parallelism to make use of all the CPU cores. Thus, we have to capture the parallelism of both levels inside the same dataflow job. This is not trivial because current dataflow engines provide only flat-parallel operations. Fourth, iterative tasks raise the need for scalability in the total number of iteration steps across all inner computations.

We propose Matryoshka, a system for nested parallelism that tackles all the above challenges in an efficient manner, even when the task involves control flow statements (e.g., iterations). Specifically, we make the following major contributions:

(1) We devise a novel two-phase flattening process (Sec. 4) that translates a nested-parallel program into a highly efficient flat program, which can run on an existing dataflow engine. Our two-phase flattening process comes with a set of nesting primitives that allow us to select the best physical operator implementations at runtime. We then present techniques to deal with closures in UDFs (Sec. 5).

(2) We show how to flatten programs even in the presence of control flow statements (e.g., the iteration in PageRank) at inner nesting levels. This is necessary for true compositionality: Users should be able to take a program that involves control flow and place it inside a larger program at an inner nesting level. (Sec. 6)

(3) We show how to leverage the program structure highlighted by our nesting primitives. Especially, we present three optimization techniques to produce a highly efficient flattened program. (Sec. 8)

(4) We experimentally validate our system using four common data analytics tasks and compare it to DIQL [21] as well as the outer- and inner-parallel workarounds on Spark for its lack of nested-parallelism support. The results show that Matryoshka is up to two orders of magnitude faster than the baselines. Importantly, our system achieves nearly linear scalability. (Sec. 9)

2 Motivating Examples

We detail three common examples where nested parallelism is essential. We then distill two desiderata for proper nested parallelism support in parallel dataflow engines.

2.1 Bounce Rate

When analyzing website traffic data, a commonly used metric is the *bounce rate* [30, 40], which denotes the ratio of the visitors who visited only one page to all the visitors. Assume we have a function for calculating the bounce rate from our entire page visit log. The function computes a single value from a `Bag` of page visits, where `Bag` is the collection abstraction in a dataflow engine (e.g., `RDD` in Spark). Consider if now we want to calculate the bounce rate per day (or per country). Intuitively, we can simply group by the days of visits and apply our Bounce Rate function on each group. Although this makes sense in theory, it is problematic in practice because current dataflow engines have the following (or similar) `groupBy` output type: `Bag[(Day, Array[Visit])]`. Here, the inner collection, which holds one group of visits, is not the system's collection abstraction. As a result, the already written Bounce Rate function cannot consume it, because its input can only be a `Bag`. One can try the usual workarounds explained before: While outer-parallel would be to rewrite the Bounce Rate function to consume an `Array` instead of a `Bag`, inner-parallel would be to rewrite `groupBy` to output an `Array[(Day, Bag[Visit])]`. However, besides requiring a considerable extra effort to code them, it is hard to select the best of the workarounds. Making such a selection depends on a complicated interplay of many factors, such as the group sizes, number of groups and CPU cores, and memory size. Also note that neither of the workarounds are suitable in case of skewed group sizes.

Ideally, we want to parallelize on both the outer and inner levels. This means running the operations and the different invocations of the Bounce Rate function in parallel. This requires the following `GroupBy` output type: `Bag[(Day, Bag[...])]`, which indicates to the dataflow engine that it should parallelize on both levels when processing the nested collection. One could then benefit from high parallelism, low job-launch overhead, and robustness to cluster- and data-characteristics, such as data skew.

2.2 Partitioned Graph Analytics

We now discuss an example that highlights how proper nested parallelism support would enhance the composability of different algorithms. Graph partitioning, e.g., connected components, is

an important building block in many complex graph processing tasks [15, 31]. As an example of such a task, consider computing the average distance between all pairs of nodes in each component of an input graph. Assume that we have a library of graph processing functions [23], where both parts of this task are already available: computing the components, and computing the average distances *for an entire graph*. We should then be able to combine these parts to solve our task in the following straightforward way, because each component can be itself considered a graph:

```
connectedComps(g).map(avgDistances)
```

However, this does not work in current dataflow engines, as the output of the connected components function is not a nested `Bag`, but instead it is a `Bag` of vertices tagged with component IDs [51, 52]. We could try grouping by the component IDs, but we would then run into the same problem as in the previous subsection. We can solve this problem as before, i.e., with proper nesting support.

2.3 Hyperparameter Optimization

Hyperparameter optimization is a common task in machine learning, which aims at building a model with many different hyperparameter values to find out the setting that works best [8]. For example, consider the common case of a data scientist who aims at building a clustering model using the K-means algorithm. To do so, she would like to run the algorithm with many different random initializations of the centroids to find the best model for her needs.

In current systems, users typically employ the inner-parallel workaround to perform hyperparameter optimization: A loop in the driver program sequentially iterates over the hyperparameter values and launches dataflow jobs for training a model with each of these values. However, this workaround suffers from high job-launch overhead, especially with many hyperparameter values.

Instead of workarounds, native support for nested parallelism would enable users to express hyperparameter optimization in the following way: they create a bag of parameter values to try, call a `map` on it, and in the UDF of the `map` train and test a model. For the training and testing, they can use the system's parallel operations. This allows the system to parallelize on both levels: different hyperparameter values are tried in parallel, while at the same time individual model training steps are also parallelized. It is worth noting that machine learning training involves loops, and thereby a loop appears inside the UDF in this example. Despite this common characteristic in modern data analytics, state-of-the-art flattening-based systems [20, 21] do not support loops at inner nesting levels. Therefore, they are not suitable for this kind of tasks.

Moreover, some iterative hyperparameter optimization algorithms determine the hyperparameter values to try next based on the results of earlier values. These algorithms often try many parameter values in one iteration [19, 27]. Also, sampling-based techniques often dynamically vary the sample size [32]. Thus, it is important to efficiently handle both a large number of small samples and a small number of large samples.

2.4 Desiderata

We observe that, in all the above-mentioned examples, users can always employ one of the common workarounds: outer- or inner-parallel. However, besides the effort of implementing them, these two workarounds suffer from poor performance, being often far from the ideal (see Figure 1). The reader might think users can manually write flattened versions of their nested programs to solve the aforementioned problems. However, this is far from being realistic and practical: Already for simple cases, such as the Bounce Rate example (Listing 1), it is hard to devise a flattened version (Listing 3). This just becomes more challenging, even for expert users, when there are control flow statements at inner nesting levels. Furthermore, manual flattening is even less realistic when working with library functions written by someone else, such as in Sec. 2.2.

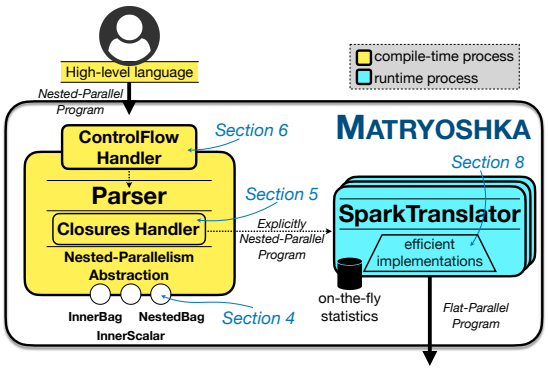


Figure 2: Matryoshka architecture.

To solve all these performance and usability problems, it is crucial to devise an automatic solution for nested parallelism that provides scalability, and ease-of-use: Users should not worry about workarounds or manual flattening. With this in mind, we identify two core desiderata for proper nested parallelism support: The system must (i) allow applications to use scalable operations both inside and outside their UDFs, and (ii) apply nested parallel operations over the provided dataset abstractions leading to nested collections.

3 Overview

In the following, we introduce Matryoshka, a system that *flattens* [12, 20, 46, 47] an input program that has multiple levels of parallelism (nested-parallel program) into a program with only one level of parallelism (flat-parallel program). This way, standard dataflow engines can execute the program fully in parallel. Note that Matryoshka is an integral part of the execution layer of Agora [45], a data infrastructure for data science and AI innovation.

Figure 2 illustrates the general architecture of Matryoshka. Users provide their programs in a high-level data analytics language that allows for nesting collections and parallel operations. We use Emma [5, 6] as our query language, which is an *embedded* domain-specific language in Scala. This means that Emma is expressed in a general-purpose programming language (similarly to Spark and Flink). For example, Listing 1 shows an Emma program for our per-day bounce rate example. The crucial difference to Spark and Flink is that Emma's data collection type (`Bag`) and the operations over a `Bag` can be nested. It also allows for imperative control flow, such as while loops and if statements. Note that our proposed techniques are compatible with other analytics languages that have nesting, such as Pig Latin [36], SQL+nested data [39], MRQL/DIQL [20, 21]. We could also integrate our techniques with Rheem [1–3] (now Apache Wayang), to execute on different dataflow engines.

Given a nested-parallel Emma program as input (the user's program), Matryoshka removes any nesting from (i.e., flattens) the

```
1  val visits: Bag[(Date, IP)] = readFile(...)
2  val visitsPerDay: Bag[(Date, Bag[IP])] =
3    visits.groupByKey()
4  visitsPerDay.map{
5      (day: Date, group: Bag[IP]) =>
6    val countsPerIP = group.map((_, 1)).reduceByKey(_+_)
7    val numBounces = countsPerIP.filter(_._2 == 1).count()
8    val numTotalVisitors = group.distinct().count()
9    val bounceRate =
10     numBounces / numTotalVisitors
11   return bounceRate
12 }
```

Listing 1: Bounce rate program (Sec. 2.1) using nested bags and nested parallel operations. The brown parts are not supported in current dataflow engines.

program so that it can be executed on a standard dataflow engine (without resorting to the inner-parallel or outer-parallel workarounds). Matryoshka performs this flattening in two phases. First, a *parsing phase* rewrites the input program by introducing into the code a set of new *nesting primitives* (`InnerScalar`, `InnerBag`, and `NestedBag`). These nesting primitives inform the next phase about the nesting structure of the program. Still in the parsing phase, Matryoshka turns imperative control flow constructs into higher-order function calls. At the same time, it also makes closures explicit, i.e., when a UDF refers to an outside variable, Matryoshka adds it as a parameter to the UDF.

One can see the output of the parsing phase as a logical plan, because the actual operator implementations are still left open. In other words, the operations of the `InnerScalar`, `InnerBag`, and `NestedBag` nesting primitives are not yet translated to flat operations of a parallel dataflow engine. Thus, our next phase, which we call *lowering phase*, is responsible for this final translation to a flat program: It executes the modified program outputted by the parsing phase, and when it encounters an operation of the above nesting primitives, it selects a concrete implementation and executes it. This lowering phase happens at runtime because the selection of the optimal implementation depends on the cardinality of intermediate datasets. Thus, for optimization purposes, Matryoshka keeps track of the cardinalities at runtime by exploiting the program structure highlighted by our nesting primitives.

For brevity reasons, in the following three sections, we assume that there are only two levels of parallelism in the input program.

4 Flattening

Our goal is to produce efficient flat-parallel programs from nested-parallel programs to enable execution on standard dataflow engines. This is challenging because finding the optimal operator implementations requires knowledge about data characteristics, which are typically not available at compile-time. We tackle this challenge by introducing a novel two-phase flattening process: the *parsing* (performed at compile time) and *lowering* (performed at runtime) phases. We then present the core concept of *lifting* [10, 46], which we rely on throughout both phases. Next, we explain the three primitives the parsing phase uses to make nested-parallel operations explicit (`InnerScalar`, `InnerBag`, and `NestedBag`). We also show how the lowering phase resolves these primitives into calls to the standard, flat data-parallel operations of standard dataflow engines.

```
1  val visits: Bag[(Date, IP)] = readFile(...)
2  val visitsPerDay: NestedBag[Date, IP] =
3    visits.groupByKeyIntoNestedBag()
4  visitsPerDay.mapWithLiftedUDF {
5      (day: InnerScalar[Date], group: InnerBag[IP]) =>
6    val countsPerIP = group.map((_, 1)).reduceByKey(_+_)
7    val numBounces = countsPerIP.filter(_._2 == 1).count()
8    val numTotalVisitors = group.distinct().count()
9    val bounceRate =
10     binaryScalarOp(numBounces, numTotalVisitors)(_ / _)
11   return bounceRate
12 }
```

Listing 2: Explicitly nested-parallel bounce rate program.

Finally, we show how to handle such bag operations that appear in a UDF of an operation other than `map`.

4.1 Two-Phase Flattening

We perform the flattening of a nested-parallel program in two phases so that we can enable further optimizations (Sec. 8). We first make explicit all nested-parallel operations in a nested-parallel program (the *parsing phase*). We then translate these explicit nested-parallel operations into efficient implementations having a single level of parallelism (the *lowering phase*).

4.1.1 Parsing Phase

This phase receives a nested-parallel program as input and outputs a program where all nested-parallel operations are made explicit. This is carried out at compilation time leveraging meta-programming, i.e., manipulating the abstract syntax tree of the input nested-parallel programs provided by the user. Compile-time meta-programming is necessary for two reasons. First, we need to turn *scalar*[1] operations and control flow operations into staged computations.[2] These staged versions create a representation of the computation, which the system then can translate to a flat-parallel computation in the lowering phase. Second, it is easier to distinguish between `Bag`s in different nesting situations at compile time while looking at the code as data, rather than at runtime. This distinction allows us to represent them differently with flat `Bag`s.

Let us illustrate this phase through the Bounce Rate example in Listing 1. The parsing phase takes as input this program and outputs the explicitly nested-parallel program in Listing 2 by performing the following main changes (highlighted in brown). First, it wraps scalars inside UDFs as `InnerScalar`s (the lowering phase will need to turn these into `Bag`s). For example, in Listing 1, `numBounces` and `numTotalVisitors` in Lines 7–8 are scalars (integers), while in Listing 2 they are `InnerScalar`s. Second, it turns `Bag`s inside UDFs into `InnerBag`s. An example is the `group` variable in ln. 5 of Listing 1 and 2. Third, it turns nested bags, i.e., `Bag[(A,Bag[B])]`, into `NestedBag[A,B]`, e.g., `visitsPerDay` in ln. 2.

4.1.2 Lowering Phase

This phase receives an explicitly nested-parallel program (Listing 2) and outputs a flat-parallel program (Listing 3). More specifically, it resolves the operations of the `InnerScalar`, `InnerBag`, and `NestedBag` nesting primitives to flat physical implementations that are executable on a parallel dataflow engine. The resulting

[1] We use the term *scalar* for any non-`Bag` type, even tuple types, such as `(A,B)`.
[2] Staging a computation means creating a representation of the computation instead of executing it directly. In general, staging allows a system to inspect a computation, transform it, instrument it, or execute it lazily.

```
1  val visits: Bag[(Date, IP)] = readFile(...)
2  val countsPerIPPerDay: Bag[((Date, IP), Int)] =
3    visits.map((_, 1)).reduceByKey(_+_)
4  val numBouncesPerDay: Bag[(Date, Int)] =
5    countsPerIPPerDay.filter(_._2 == 1)
6    .map{case ((day,ip),count) => (day,1)}.reduceByKey(_+_)
7  val numTotalVisitorsPerDay: Bag[(Date, Int)] =
8    visits.distinct()
9    .map{case (day,ip) => (day,1)}.reduceByKey(_+_)
10 val bounceRatePerDay: Bag[(Date, Double)] =
11   (numBouncesPerDay join numTotalVisitorsPerDay)
12   .map{case (day, (numBounces, numTotalVisitors)) =>
13     numBounces / numTotalVisitors}
```

Listing 3: Flat-parallel bounce rate program.

flat-parallel program is both equivalent to the initial input nested-parallel program (Listing 1) and executable on any standard parallel dataflow engine. For example, in our system architecture (Figure 2), *SparkTranslator* performs this phase for Spark. There could be more translators added for other parallel dataflow engines, e.g., for Flink.

4.2 Lifting UDFs

Let us start defining what it means to *lift* a UDF. As current dataflow engines cannot handle parallel `Bag` operations inside a `map` UDF, we have to remove the `map` and perform the UDF's operations at the top level. This is known as lifting UDFs. Formally, if the original UDF had the type `A=>B`, the lifted version will have the type `Bag[A]=>Bag[B]`. Intuitively, lifting a UDF consists of moving all the operations that were originally inside the UDF to the top level.

Lifting an operation does not only move it but also changes the operation to make it work. Originally, an operation that is inside a UDF is invoked as many times as the UDF is invoked (disregarding loops and other control flow for now). In a single call, the operation computes just one bag or one scalar as its output inside the UDF. However, the lifted version of an operation is invoked just once *in total*. During this single invocation, it needs to compute what the original version computed over all the invocations of the UDF. If the original operation computed a scalar `S`, then the lifted operation has to compute a `Bag[S]`. If the original operation computed a `Bag[A]`, then the lifted operation has to compute many bags, which could be expressed as a `Bag[(T,Bag[A])]`. Here, `T` denotes a tag type, which identifies inner bags by which UDF call they appeared in, see Sec. 4.3. However, even though lifting bag operations in this way would indeed remove the nesting of operations, but it would also introduce nested bags. Thus, the actual lifted operation needs to represent the nested bag with a flat bag (as explained in Sec. 4.4).

For example, the operations in Listing 1 ln. 7 have lifted versions in Listing 3 ln. 4–6. Observe the change in the variable names: the original operations computed `numBounces` for a single day (an `Int`), while the lifted versions compute `numBouncesPerDay` (a `Bag[Int]`), i.e., the number of bounces for each of the days.

In our two-phase flattening, lifting a `map` UDF is then performed as follows. The parsing phase does not remove the `map` yet, it just changes the `map` into a `mapWithLiftedUDF`, to make it explicit to the lowering phase that this UDF needs to be lifted. The reason we do not directly perform lifting in a single phase is that our optimizations in Sec. 8 can be performed only at runtime: They need the information of which operations were in the same UDF. In contrast to a normal `map`, a `mapWithLiftedUDF` calls its UDF only once (during the lowering phase), and this single execution operates on all the elements of the bag that `mapWithLiftedUDF` is called on. Inside the UDF of the `mapWithLiftedUDF`, a scalar that was in the original UDF is replaced with an `InnerScalar`, which is represented by a `Bag` after the lowering phase. Similarly, a `Bag` in the original UDF is replaced with an `InnerBag`, which has the same information as a `Bag[(T,Bag[A])]`, but is represented by a flat bag after the lowering phase.

4.3 InnerScalar

Lifting a UDF requires us to lift all its scalar operations. The challenge here is that scalar variables originally do not involve any system-provided types, e.g., just a single `Int` value, and not something like a `Bag[Int]`. This is a problem because the system needs to manipulate these values to lift their operations.

We tackle this challenge by leveraging compile-time metaprogramming [16], to change the code. Specifically, the parsing phase wraps scalar variable types inside `InnerScalar`, i.e., a scalar type `S` turns into `InnerScalar[S]`. At the same time, the parsing phase wraps scalar operations in the operations of `InnerScalar`. Specifically, `b = f(a)`, where `a` and `b` are scalars, and `f` is a unary scalar operation, turns into `b = unaryScalarOp(a)(f)`. Similarly, `c = f(a,b)`, where `a`, `b`, and `c` are scalars and `f` is a binary scalar operation, turns into `c = binaryScalarOp(a,b)(f)`. For example, if we have `c = a + b` somewhere in a UDF, then what the system has to know is that `c` is computed from `a` and `b` by some operation. Thus, we translate such a line of code as a binary scalar operation `c = binaryScalarOp(a,b)(_+_)`, where `a`, `b`, and `c` are all `InnerScalar`s. As a more concrete example, Listing 2 shows such a lifted UDF with several `InnerScalar`s in it. In summary, `InnerScalar` makes the scalar operations inside UDFs explicit in the logical plan outputted by the parsing phase, enabling optimized flat implementations in the lowering phase.

The lowering phase then resolves an `InnerScalar` to a flat `Bag` containing all the values that the variable would have in all the invocations of the original UDF. By using a `Bag`, we are scalable in the number of UDF invocations of the original code. For example, in Listing 1, the `map` UDF is originally called for each day, and `numBounces` is a scalar that is computed for each day. The lowering phase replaces this scalar with `numBouncesPerDay`, a `Bag` containing the values of `numBounces` for each day.

In detail, the implementation of `unaryScalarOp` is a `map`, which applies the given unary function (negation in the above example) to each of the scalars. The implementation of `binaryScalarOp` is more sophisticated because we first have to bring together matching pairs of scalar values from its two inputs. That is, we must find such pairs of scalar values that would have belonged to the same original UDF invocation. Doing so allows us to execute the scalar operation over such pairs with a `map`. To achieve this scalar match up, we add a *tag* to each element of the `Bag` that represents the `InnerScalar`. A tag identifies invocations of the original UDF. For instance, in Listing 3 the tags identify the days, and therefore `numBouncesPerDay` is a `Bag[(Date, Int)]`. We can then perform an equi-join between the two input `Bag`s representing the two `InnerScalar`s, being the tag the join key. In more detail, we implement `binaryScalarOp(a,b)(f)` as

`a'.join(b').map(f)`, where `a'` and `b'` are the `Bag`s representing the `a` and `b` `InnerScalar`s. For example, the flat version of Listing 2 ln. 10 is Listing 3 ln. 10–13.

We create tags for all `InnerScalar`s as follows: If the `InnerScalar` is created from another `InnerScalar` (or `InnerBag`), we simply propagate the tags from the input; if `mapWithLiftedUDF` runs on a `NestedBag` (Sec. 4.5), we then propagate the tags from the `NestedBag`; if `mapWithLiftedUDF` runs on a non-nested `Bag`, we create the tags using the standard `zipWithUniqueId` operation, which assigns unique tags. Note that the set of tags is the same for all `InnerScalar`s inside a UDF, which is important for our optimizations in Sec. 8.

Earlier, we showed simplified generic type parameters for `InnerScalar`s. However, the full type involves the type of the tags as well as the type of the original scalar: `InnerScalar[T,S]`. After the lowering phase, this is represented as a `Bag[(T,S)]`. Formally, `unaryScalarOp(s)(f)` is resolved by the lowering phase to `s'.map((t,x)=>(t,f(x)))`, where `s'` is a `Bag[(T,S)]` representing `s`. Moreover, `binaryScalarOp(a,b)(f)` is resolved to `a'.join(b').map((t,(x,y))=>(t,f(x,y)))`, where `a'` and `b'` are the bags representing the `a` and `b` `InnerScalar`s.

4.4 InnerBag

Similarly to scalars, we must lift each `Bag` operation that is inside a UDF when lifting it. Originally, a `Bag` operation in a UDF creates many bags in many UDF invocations. The lifted version creates just one flat bag for all the invocations of the UDF. This eliminates the per-bag overhead occurring in each UDF invocation with the inner-parallel workaround. Still, lifting bag operations is tricky because the optimal physical implementation for joins and cross products in some flattened operations depends on intermediate data characteristics visible only at runtime. We rely on our two-phase flattening process to overcome this difficulty.

The parsing phase introduces an `InnerBag` variable instead of each `Bag` variable in the UDF. An `InnerBag` represents a collection of bags, where each bag belonged to one invocation of the original UDF. For example, in Listing 1 `countsPerIP` is a different bag in each invocation of the original `map` UDF. In the parsing phase, it is replaced by an `InnerBag` (Listing 2 ln. 6), which represents all these bags. `InnerBag[A]` contains the same information as a `Bag[(T,Bag[A])]`: The `T` tags identify a UDF invocation, where the corresponding inner bag occurred. However, the operations of `InnerBag` work with the inner bags: For each of the classic operations of `Bag[A]` (e.g., `map`, `filter`, `join`, etc.), `InnerBag[A]` has a corresponding operation that performs the same computation on all the inner bags. Formally, if there is a bag operation `op: Bag[A]=>Bag[B]`, then its lifted version `op'` has the type `Bag[(T,Bag[A])]=>Bag[(T,Bag[B])]`, and performs `op'(xss) = xss.map(op)`. However, this is a nested bag and we have to represent `InnerBag`s with flat bags.

The lowering phase resolves an `InnerBag` to be a flat `Bag`, which consists of all the elements of all the bags that the `InnerBag` replaces. Similarly to `InnerScalar`, each element is tagged with an identifier of the original UDF invocation. Formally, `InnerBag[T,E]` is resolved by the lowering phase to `Bag[(T,E)]`, where `T` is the tag type and `E` is the original `Bag`'s element type. For example, from Listing 2 to Listing 3 `countsPerIP` is replaced by `countsPerIPPerDay`, which contains all the values from all the bags that `countsPerIP` has, tagged by the day. As a more concrete example, assume that, inside a UDF, there is a `Bag` variable whose value is *{apple, orange}* in one UDF invocation and *{dog, cat}* in another. Then, the lowering phase could represent the corresponding `InnerBag` as the flat bag *{(0, apple), (0, orange), (1, dog), (1, cat)}*.

`InnerBag`'s operations mirror the operations of normal `Bag`s: their signatures are the same but their inputs and outputs are `InnerBag`s and/or `InnerScalar`s. The implementations of the operations are the lifted versions of the corresponding `Bag` operations. We lift stateless `Bag` operations, which perform over individual elements (such as `map`, `flatMap`, and `filter`), by performing the UDF on the second component of the pairs and by forwarding the tags unchanged. Still, some other `Bag` operations are stateful (e.g., aggregations). We lift these operations by keeping the state per tag. For example, a `reduce` turns into a `reduceByKey`, where the key is the tag. Calling `reduce` on an `InnerBag` then results in an `InnerScalar`. In case a `Bag` operation already has a per-key state, we lift it by creating a composite key from the original key plus the tag. For instance, we lift `b.reduceByKey(f)` (Listing 1 and 2, ln. 6) as:

```
b'.map{case (t, (k, v)) => ((t, k), v)}
  .reduceByKey(f)
  .map{case ((t, k), v) => (t, (k, v))}
```

We also lift `joins` with a similar rekeying. Some other operations' lifted versions are simply identical to the original operations, such as `distinct` and `union`. To handle operations that produce output for empty input bags (e.g., `count` has to produce 0), we additionally need to store all the tags in a separate `Bag[T]`. This is because `InnerBag`'s representation `Bag[(T,A)]` does not have any element corresponding to empty inner bags. We store the bag of tags once per lifted UDF because they are same for all `InnerBag`s in a UDF.

4.5 NestedBag

While `InnerScalar` and `InnerBag` are representations for scalars and non-nested `Bag`s inside a UDF, we still need to lift a nested `Bag` that is outside a UDF. The parsing phase introduces the `NestedBag` for a nested `Bag` outside a UDF. This is the case for Listing 1 ln. 2, where a nested `Bag` appears from a `groupBy`. Recall `NestedBag`s are a typical case in nested-parallel programs (Sec. 2.1–2.2).

To explain how the lowering phase translates a `NestedBag` to a flat `Bag`, we first focus on the simplest case of a nested bag: `Bag[Bag[I]]`, where `I` is some arbitrary scalar type. Similarly to an `InnerBag`, we represent this as a flat `Bag` containing all the elements of the inner bags. Each element of this flat `Bag` has a tag `T` that identifies which of the inner bags it originally belonged to: `Bag[(T,I)]`. For instance, if the original nested bag is *{{apple, orange}, {dog, cat}}*, then the lowering phase could represent the `NestedBag` with the flat bag *{(0, apple), (0, orange), (1, dog), (1, cat)}*. Note that this is exactly the `InnerBag` type.

Still, in the more general case, the element type of the outer `Bag` is usually more complicated. It usually has some other components besides the inner `Bag`. We capture these other components in an arbitrary type `O`. We, thus, have a nested `Bag` before the parsing phase as follows: `Bag[(O,Bag[I])]`, i.e., the element type of the outer `Bag` is a pair of a scalar and an inner `Bag`. The parsing phase turns this into a `NestedBag[O,I]`. Our flat representation of such a nested `Bag` is composed of an `InnerScalar[T,O]` and an `InnerBag[T,I]`. For example, if our original nested `Bag` was

{(fruit, {apple, orange}), (animal, {dog, cat})}, then the `NestedBag` could be represented by the `InnerScalar` *{(0, fruit), (1, animal)}* and the `InnerBag` *{(0, apple), (0, orange), (1, dog), (1, cat)}*.

4.6 Lifting non-Map UDFs

So far, we focused on lifting the UDF of a `map`. We now explain how to lift UDFs of other operations. We reduce other cases to lifting `map` UDFs via some simple program transformation. We basically split a complex operation into a `map` with a UDF plus the UDF-less version of the original operation. For example, consider the case where the input program has `xs.groupBy(keyFunc)`. We can change this into `xs.map(x=>(keyFunc(x),x)).groupByKey()`, where `groupByKey` is the UDF-less version of `groupBy` (i.e., it uses the key that is already in the input tuples instead of a UDF). Similarly, we can use the same splitting process for `joins` whose keys are given in UDFs, and for `filter`. In contrast, FlatMap is a slightly different case. Here, we change `xs.flatMap(f)` into `xs.map(f).flatten()`, where `flatten` is a special operation that removes the nesting structure. `Flatten`'s implementation simply removes the tags from an `InnerBag`.

5 Dealing with Closures

Previously, we saw how to lift UDFs via three primitives for nested-parallelism, namely `InnerBag`, `InnerScalar`, and `NestedBag`. We now address the case where a UDF refers to a variable that is defined outside the UDF, a.k.a., *closure*. For example, when initializing the ranks in PageRank, we first have to compute the initial weight from the number of pages and then use this value inside a UDF:

```
val initWeight: Double = 1.0d / pages.count()
val initPR = pages.map(x => (x, initWeight))
```

The challenge is that `initWeight` is in the memory of the driver program, while the UDF typically runs on the worker nodes. Dataflow engines handle this situation by simply broadcasting `initWeight` to all workers in the cluster. However, as we will shortly see, we have to make additional considerations in our system. We will distinguish between two cases below, depending on whether the UDF that has the closure is lifted.

5.1 Unlifted UDF Case

We first explain when this case happens. Consider the above two lines, where the UDF of the `map` is not lifted but has a closure. Assume these two lines are themselves inside an outer UDF and that such an outer UDF gets lifted because of its bag operations. In this case, `initWeight` becomes an `InnerScalar` and the `map` becomes a lifted `map` (but its UDF is still not lifted).

The difficulty here is that the original reference to `initWeight` referred to just a single scalar value. Leaving the code unchanged, the reference to `initWeight` would refer to all the scalars that are in the `InnerScalar`. Instead, we model a `map` as a two-input operation: one input is the `pages` bag and the other is the `initWeight` `InnerScalar`. It now becomes apparent that we need a similar join on the tags as in a binary `InnerScalar` operation (see Sec. 4.3). That is, each different value of `initWeight` has to meet the appropriate values of `pages`, i.e., those with the same tag. We do so by introducing `mapWithClosure`, which takes the closure as an extra argument and hands it into the inner UDF as an extra argument:

```
pages.mapWithClosure(initWeight, (x, clos) => (x, clos))
```

In more detail, `mapWithClosure` performs a join on the tags between the bags representing `pages` and `initWeight`. Note that, due to this example's simplicity, the inner UDF ended up being an identity function, but this is not the case in general.

5.2 Lifted UDF Case

Consider our hyperparameter optimization in Sec. 2.3, where the `Bag` containing the training data is defined at the outermost level but it is used inside a lifted UDF. This case is different from the unlifted UDF case above because a lifted UDF is called inside the driver program. As a result, we do not need to broadcast the closure to the worker nodes for the UDF to access it.

The difficulty in this case resides in that the closure is just a normal bag or scalar (not `InnerBag`). We, thus, create a lifted version of the referenced bag (or scalar), where it is replicated for each different tag value that is in the lifted UDF. However, this can make it very large, as it involves replicating the bag (or scalar) as many times as the non-lifted version of the UDF would have been invoked. To mitigate this problem, we create "half-lifted" operations, where only some of the inputs are lifted. For instance, the following three lines of code represent a half-lifted equi-join between `left` and `right`, where `left` is an `InnerBag` and `right` is a normal bag (`left.repr` accesses the flat bag representing the `InnerBag`):

```
val rekeyed = left.repr.map{case (l,(k,v)) => (k,(l,v))}
val joined = rekeyed join right
joined.map{case (k, ((l, v), w)) => (l, (k, (v, w)))}
```

6 Handling Control Flow Statements

We now discuss how to flatten programs in the presence of control flow statements inside UDFs. If Matryoshka has to lift a UDF containing such statements, then it also needs to lift these statements. However, doing so is challenging because control flow might go differently in different executions of the UDF (e.g., loops exit at different iterations, or different if-branches are taken). The lifted version of a control flow statement must cover all these different executions. We leverage our two-steps flattening process to tackle this challenge. In the parsing phase, we first substitute control flow statements with staged function calls (Sec. 6.1). In the lowering phase, we then lift `while loop` and `if` statements (Sec. 6.2).

6.1 Control Flow as Higher-Order Functions

As a first step in the parsing phase (before we represent nested operations with our primitives), we change control flow statements into (higher-order) function calls. This enables us to change the function calls to the lifted versions. We, thus, encapsulate the lifted versions inside functions, which run during the lowering phase. This is similar to the operations of our nesting primitives (`InnerScalar`, `InnerBag`, and `NestedBag`), which encapsulate the lifted versions of bag and scalar operations. The function signatures are as follows. We express an `if` statement as a function that takes as arguments the condition as a boolean value and a function for each of its branches. Likewise, we express a while loop statement as a function that takes as argument the body as a function. This body function takes the previous values of the loop variables as input and returns both the next values of the loop variables and the value of the exit condition. Note that these higher-order functions are similar to how several parallel dataflow engines support control flow statements.

6.2 Lifting Loops and If Statements

We now focus on how we lift while loops and if statements. A lifted loop is basically a loop that performs the work of many unlifted

```
1 var bodyIn: InnerBag[T,A] = initialBodyIn
2 var result = Bag.empty
3 do {
4  val (bodyOut, cond) = bodyFunc(bodyIn)
5  val bodyOutWithCond = bodyOut.joinOnTags(cond)
6  bodyIn = bodyOutWithCond.filter(_._2).map(_._1)
7  val finished = bodyOutWithCond.filter(not _._2).map(_._1)
8  result = result.union(finished)
9 } while (bodyIn.repr.notEmpty)
```

Listing 4: Lifted while loop.

loops. In other words, the first iteration of a lifted loop executes the first iteration of all the original loops, the second iteration of a lifted loop executes the second iteration of all the original loops, and so on. The challenge is that the original loops might finish at different iterations from each other.

We tackle this challenge by relying on the abstractions for lifted operations introduced in Sec. 4. Assume we have already lifted all the bag and scalar operations inside the loop body, i.e., substituted scalars and bags with `InnerScalar`s and `InnerBag`s. In this case, we do not need to further modify the loop body when lifting the loop. This is because the lifted versions of all the scalar and bag operations inside the loop body already do exactly what the lifted loop needs: it executes the original operation on many scalars or bags at the same time. Note that we also turn variables that are passed between iterations into `InnerBag`s and/or `InnerScalar`s.

Still, we must manage data that enters or leaves the body at each iteration and lift the loop control logic. Specifically, we need to:
(P1) discard those parts of `InnerBag`s and `InnerScalar`s from the iterations whose original loops have finished;
(P2) save the result of the discarded parts as soon they finish; and
(P3) exit the lifted loop when all the parts are discarded, i.e., when all the original loops have finished.

To check if an original loop has finished, we leverage the internal flat bag representation of `InnerScalar`s (i.e., `Bag[(T,A)]`). Recall that `T` is a tag identifying the original UDF invocations and `A` is the type of the original scalar. Thus, the `InnerScalar` of the exit condition is represented as a `Bag[(LoopID, Boolean)]`, which tells us for each original loop if it should continue. We leverage this bag to achieve the above P1-P3 as follows (see Listing 4):
Impl. of (P1). We join each `InnerBag` and `InnerScalar` that enters the loop body with the lifted exit condition on the tag to identify and discard those loops that already finished (Lines 5 & 6);
Impl. of (P2). We save into results bags exactly those values that we filtered out above, which will contain all final results once the lifted loop exits (ln. 7–8);
Impl. of (P3). If (P1) did not let through any element, then we exit the lifted loop. (ln. 9)

In case of `if` statements, the challenge is that some of the executions of the original `if` statement would have executed the *then* branch, while some others the *else* branch. Therefore, a lifted if statement executes both branches but lets get in, into each of them, only the values for those tags for which the if condition had the appropriate value. For this, it uses join and filter in the same way the lifted loops handle the data from loops that have finished.

7 Completeness and Correctness

We show proof sketches for the completeness and the correctness of our flattening procedure. Before proceeding, let us mention that we assume that bags do not appear inside other data structures, such as `Array[Bag[...]]`. We believe this is a negligible limitation because such nesting structures mainly arise when employing different variations of the inner-parallel workaround only. We also assume that the UDFs of bag *aggregations*, such as `reduce`, do not contain bag operations, which is an uncommon case in practice.

Theorem 1. *(Completeness) Matryoshka can flatten any nested program expressed with the standard bag operations and without bags embedded in other data structures or in aggregation UDFs.*

Proof. *(Sketch)* For brevity, we first show the proof for two levels of parallelism. The proof shows that the parsing phase can always transform nested bags and UDFs with bag operations into the `InnerBag`, `InnerScalar`, `NestedBag` nesting primitives (which have flat implementations). As mentioned before, a preparation step of the parsing phase eliminates those non-`map` operations that have UDFs with bag operations (Sec. 4.6), eliminates closures (Sec. 5), and transforms control flow statements into a functional representation (Sec. 6.1). Then, the parsing phase traverses the code statement-by-statement (compound statements are broken down into atomic statements), and makes local changes on certain statements. Thus we focus on proving that the parsing phase can handle any statement in the input program. The next two paragraphs cover statements outside and inside UDFs, respectively.

The parsing phase modifies a top-level statement, i.e., that is not inside any UDF, based on whether its UDF contains bag operations and whether its inputs and/or outputs are nested, which leads to three cases: (1) *The operation's UDF contains bag operations.* A top-level operation can only be a `map` in this case, because all operations whose UDFs involve bags were eliminated by our aforementioned preparation step. Thus, the parsing phase turns the `map` into a `mapWithLiftedUDF`[3]; (2) *Flat input and nested output.* If the top-level operation is a `map`, the parsing phase modifies it as in the previous case. Otherwise, the top-level operation is a `groupByKey` (no other operation could introduce a nested bag from a flat bag) and hence the parsing phase turns it into a `groupByKeyIntoNestedBag`; (3) *Nested input.* This case occurs because of earlier statements whose outputs were already transformed into a `NestedBag`. Here, if the top-level operation is a `map`, the parsing phase turns it into a `mapWithLiftedUDF`. Otherwise, the top-level operation can only be a UDF-less bag operation, which all have their flattened versions on `NestedBag`.

For statements inside UDFs, the parsing phase has to change them only if it lifts the UDF (i.e., when the UDF has bag operations). In this case, it turns each scalar value into an `InnerScalar` and each `Bag` into an `InnerBag`. Operations of these types are substituted in place of the original operations as explained in Sec. 4.3 and 4.4. It also turns control flow operations into their lifted equivalents (Sec. 6.2). Note that the lifting of operations that are in the bodies of control flow constructs proceeds as usual, i.e., a surrounding control flow construct has no effect on the lifting of an operation.

We now consider the case of handling more than two levels of parallelism. Here, we create a more complex `NestedBag` type, which has a separate instance of `InnerScalar` for each intermediate level and one instance of `InnerBag` for the innermost level.

[3]Recall that `mapWithLiftedUDF`'s UDF's input/output types involve `InnerBag` and/or `InnerScalar` based on the input/output types of the original `map` UDF.

Lifting tags for three or more levels are composed of one lifting tag for each outer level. These tags are combined into a composite key, which ensures that the implementations of the lifted operations are the same for `InnerBag`s and `InnerScalar`s at any level. □

Theorem 2. *(Correctness) Matryoshka always produces a flat program that is equivalent to the original input nested program.*

Proof. *(Sketch)* The proof first shows that changing the data from the original representation to our flattened representation is an *isomorphism*. That is, we can go from the original data representation to our flattened data representation by such a map[4] m, that 1) m is a *bijection*, and 2) m *preserves* all the bag- and scalar operations. *Bijection* here means that different original data structures are mapped to different flattened data structures. m *preserving* an operation f means that $m(f(x)) = f'(m(x))$, where f is an original operation in the user's program, and f' is the flattened version of the operation, operating on the flattened data. In other words, if we first perform an original operation and then change to the flattened data representation, we get the same result as if we first changed to the flattened data representation and then performed the flattened version of the operation. For binary operations, preservation means $m(f(x, y)) = f'(m(x), m(y))$. After we establish the isomorphism property for all the operations, the next step of the proof is to note that the entire program from the inputs up to just before the final output operation[5] is a composition of operations that are each preserved by m. Thus, the entire program up to just before the output operation is also preserved by m. As a result, an output operation in the flattened program receives the same data as if we ran the original program and just changed to the flat data representation at the last moment before the output operation. The final step of the proof is to show that for an output operation o, we can implement a flattened output operation o', for which $o(x) = o'(m(x))$ (or, equivalently, $o'(x') = o(m^{-1}(x'))$). That is, the flattened output operation creates the same output file from the flat data representation as the original output operation would have created from the nested representation. This way, the flattened program will produce the same output as the original program. □

8 Optimizations

We now discuss how the lowering phase provides concrete operator implementations for the logical plan outputted by the parsing phase. It uses an optimizer to choose the right physical operator implementations at runtime, based on different data characteristics. Most of the optimizations depend on the sizes of the bags representing the `InnerScalar`s. Fortunately, the structure of the program, which is visible at the logical plan level, gives vital information about the sizes of `InnerScalar`s to the optimizer. In the remainder of this section, we first discuss how we track the sizes of `InnerScalar`s and discuss optimizations based on these sizes.

8.1 Partition Counts of InnerScalars

Dataflow engines distribute programs across a cluster by partitioning bags and the computations that create bags. If a bag is small, then each partition gets only a few elements, causing a high relative per-partition overhead that dominates runtime. Thus, it is important to set the number of partitions in accordance with the bag's size. In general, this is not possible to do for every bag because we know the size of a bag only once the bag is already fully computed: at this point, the per-partitioning overhead already occurred. Fortunately, we can do this for `InnerScalar`s, as we explain below.

We exploit an important observation to track the sizes of `InnerScalar`s: *All `InnerScalar`s inside a lifted UDF have the same size.* Recall that the bags representing `InnerScalar`s consist of `(tag, scalar-value)` pairs, where the tag is a unique key. Therefore, the size of these bags depends on the number of different tags, which is constant across all lifted operations inside a lifted UDF. This is because tags are in one-to-one correspondence with calls that would have been made to the original UDF. This means that all `InnerScalar`s inside a lifted UDF have indeed the same size, and this size is known at the beginning of a lifted UDF. The optimizer uses this size information when making decisions about physical operator implementations, such as partition counts.

We track `InnerScalar` sizes as follows. Each lifted UDF has an associated `LiftingContext` object, which stores some metadata, such as the `InnerScalar` size. Operations inside the lifted UDF always get the `LiftingContext` as an implicit argument. When the `LiftingContext` is created, the `InnerScalar` size can be determined in several different ways, depending on whether the current UDF is of a `map` whose input argument is a flat or a `NestedBag`.

8.2 Joins between InnerBags and InnerScalars

Recall from Sec. 5 that the `mapWithClosure` operation is implemented as a join between the bag representing the `InnerBag` (the primary input) and the bag representing the `InnerScalar` (the closure). A similar join occurs in the implementation of a lifted do-while loop (ln. 5 in Listing 4).

To implement these joins, there exist multiple join algorithms in dataflow engines: A broadcast join is better when one of the join inputs is small while a repartition join is better when both inputs are large, and the key cardinality is also large enough. In many cases, selecting the wrong join implementation results in a program failing or with more than an order of magnitude worse performance. Here we again exploit the previously collected information about the sizes of `InnerScalar`s to select the right join implementation. Specifically, we choose a repartition join when there are enough elements in the `InnerScalar` to give work to all CPU cores. Otherwise, we choose a broadcast join.

Note that dataflow engine optimizers can make similar join algorithm choices by themselves in some cases. However, we have more information here than what is typically available to an engine optimizer [7]: we know `InnerScalar` sizes already before they are computed (which enables, e.g., fusing the join shuffle's map side with preceding operations), and we also know that the join key is a unique key in `InnerScalar`s. We currently use this information as explained above, but in the future it might be also possible to have a closer integration with a dataflow engine optimizer, such as Catalyst [7]: Instead of directly making the join algorithm choice, we could give the above information as hints to the engine optimizer.

8.3 Half-lifted MapWithClosure

Recall from Sec. 5, that there exist half-lifted operations in lifted UDFs, where only one of the inputs comes from inside the UDF,

[4]The word *map* is used here in the mathematical sense, as opposed to the bag operation `map` in other parts of the paper.
[5]By *output operation* we mean writing a bag to a distributed filesystem, such as HDFS.

while the other input comes from outside. One of these operations that have a half-lifted version is `mapWithClosure`, with the closure being an `InnerScalar` from inside the UDF and the primary input being a closure of the enclosing UDF. For example, this occurs in K-means, when we compute the new center assignment: We call `mapWithClosure` on the bag of points (which does not change between K-means runs, and is therefore outside the lifted UDF), with the closure being the current means. In this case, a `mapWithClosure` is a cross product between the bag representing the `InnerScalar` and the primary input bag.

One can implement this cross by broadcasting one of its inputs. However, the challenge resides in selecting which input to broadcast. We address this as follows: If the `InnerScalar` has only 1 partition, we then broadcast it. This is quick to check, and it is also the common case due to the optimization in Sec. 8.1. Otherwise, we use Spark's `SizeEstimator` to compare the sizes of the two inputs and broadcast the smaller one.

9 Evaluation

We carried out several experiments to demonstrate that the performance of our system is consistent across a wide range of input dataset characteristics. We compare our system to common practices for running nested-parallel programs on dataflow engines: namely the inner-parallel and outer-parallel workarounds. We designed our experiments to answer the following questions: How does Matryoshka (i) handle a varying number of inner computations? (ii) scale with the number of machines? (iii) handle skewed inner computation sizes? (iv) chooses different operator implementations to achieve high efficiency?

9.1 Setup

Hardware. We ran our experiments on a cluster of 25 machines: each with two 8-core AMD Opteron 6128 processors, 32 GB main memory, 4×1 TB hard disks, and 1 Gb network.

Dataflow engine. We used Spark-3.0 on OpenJDK 14 and HDFS 2.7.1. We dedicated 22 GB memory per machine to Spark processes and set the Spark parallelism to 3× the total number of cores.

Tasks and Datasets. We considered data analytics tasks from different areas: namely *PageRank* (graph analytics), *Average Distances* (graph analytics), *K-means* (machine learning), and *Bounce Rate* (web analytics). While Bounce Rate and Average Distances are explained in Sections 2.1 and 2.2, PageRank and K-means are well-known tasks. To put PageRank at the inner nesting level, we perform a grouping of the graph edges and compute a separate PageRank for each group (similarly to the Bounce Rate example). This way, the program computes many PageRanks in parallel, similarly to Topic-Sensitive PageRank [25] and BlockRank [28]. Note that K-means, PageRank, and Bounce Rate have two levels of data-parallel operations, while Average Distances has three levels. Bounce Rate has no control flow statements while the other three do.

Baselines. We considered the inner- and outer-parallel workarounds as well as DIQL [21] as baselines. However, as DIQL does not support control flow statements in the inner levels, we only consider it for the Bounce Rate task.

Repeatability. All the numbers we report are the median of three runs. We provide the code of all our experiment programs[6].

[6] https://github.com/anonymous873543/Matryoshka-anonymized

9.2 Weak Scaling

We start by evaluating scalability in both inner and outer collection sizes. In detail, we varied two parameters (such as in a weak scaling experiment): (i) the number of inner computations (outer scalability), and inversely, (ii) the input sizes of inner computations inner scalability). We vary these two parameters at the same time so that the total input dataset size remains constant (e.g., 20 GB for PageRank) thereby we can better evaluate the impact of nested parallelism. We, thus, expect the runtime to stay nearly constant.

Figure 3 shows the results for all our iterative tasks, i.e., with control flow statements (K-means, PageRank, and Average Distances). We observe that Matryoshka scales gracefully with the number of inner bags. This is not the case for the two workarounds, whose performance is heavily affected by the number of inner computations. Overall, we observe that Matryoshka is highly superior to the two baselines. It is up to two orders of magnitude faster than outer-parallel (for K-means) and up to 48× faster than inner-parallel (for PageRank). More importantly, in the worst case, it achieves similar performance as both baselines. Specifically, it is similarly good as inner-parallel for a very low number of inner computations and similarly good as outer-parallel for a very large number of inner computations. Also note that our system obtains the best performance compared to baselines for Average Distances, because this task has three levels of parallelism. In such cases, outer-parallel can parallelize only the first level while inner-parallel only the third. Matryoshka can parallelize all levels.

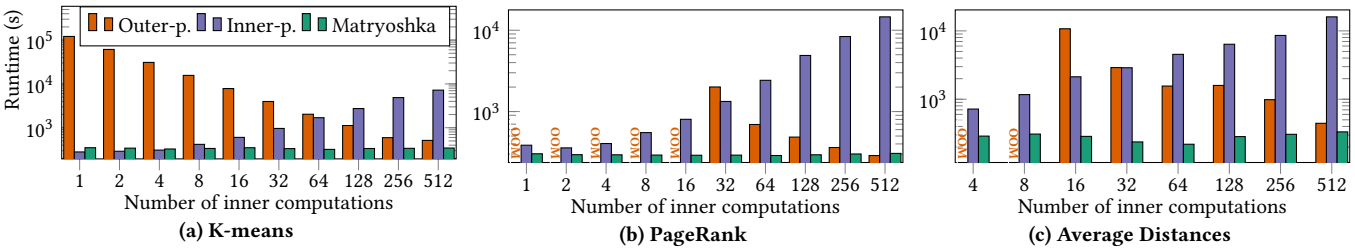
Overall, Matryoshka's high performance comes from two main aspects. First, it makes use of parallelization opportunities inside each inner computation, e.g., inside one K-means run with a certain starting centroid configuration. Second, the number of Spark jobs it launches is independent of the number of inner computations, keeping its overhead low and constant. This is also why it maintains its performance close to constant for any number of inner computations. In contrast, outer-parallel suffers from not fully parallelizing inner levels: it brings inner levels into a single machine. On the other side, inner-parallel suffers from a high total job-launch overhead, which just gets amplified with iterative tasks.

We also observe that in the sweet spots, i.e., where both baselines are equally good (at 32 and 64 inner computations), our system is still at least ~5× and up to 12× faster than both baselines. This shows that even if users (or an optimizer) could select the best workaround for a given number of inner computations, they still have significant drop-downs in performance compared to Matryoshka.

9.3 Scaling Out

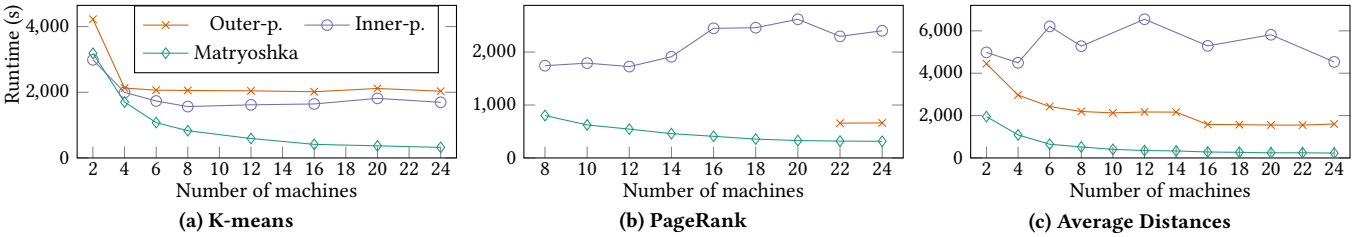
We performed an experiment for each task varying the number of machines. We set the input sizes and number of inner computations to 64. For each experiment, we start the line from where there is enough total memory to avoid crashes or extra spilling to disk.

Figure 4 shows the results of this experiment. Overall, we observe our system scales gracefully while the workarounds do not. With the maximum number of machines, our system is 5–20× faster than inner-parallel, and 2–7× faster than outer-parallel. Interestingly, we observe that our system is close to linear scalability while the two workarounds remain constant in many cases, i.e., they cannot benefit from using more machines. The superiority of our system comes from the same reasons as in the previous experiment: namely



(a) K-means (b) PageRank (c) Average Distances

Figure 3: Scalability in the number and sizes of inner computations.



(a) K-means (b) PageRank (c) Average Distances

Figure 4: Scalability in the number of machines.

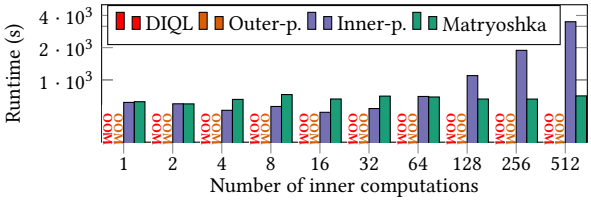


Figure 5: Bounce Rate (no control flow).

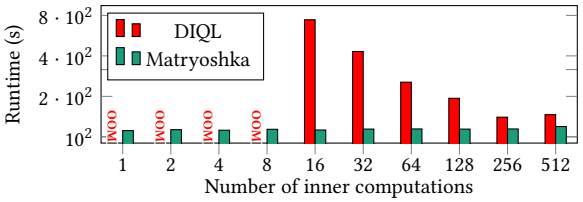


Figure 6: Performance against DIQL for Bounce Rate.

outer-parallel lacks inner-level parallelism and inner-parallel has a high job-launch overhead. In fact, we observe that the overhead of inner-parallel just gets worse as we increase the number of machines because of two main factors: more partitions mean more (i) scheduling and (ii) task-launch overheads [24, 37]. Matryoshka does not suffer from any of these problems.

9.4 Performance Without Control Flow

We now evaluate the performance of Matryoshka when a task has no control flow statements. We repeated the experiments from Sec. 9.2 and Sec. 9.3 but using the Bounce Rate task, which has no control flow statements. We considered DIQL as a baseline and 256 inner computations for the scale-out experiment.

Figure 5 shows the results. We observe that the performance of our system is again nearly constant with respect to the number of inner computations. In contrast, DIQL and outer-parallel run out of memory in all the cases and inner-parallel suffers from the job-launch overhead. Surprisingly, DIQL was not able to flatten this program: It applied the outer-parallel workaround instead, resulting in out-of-memory errors. In constrast, our system is up to 5× faster than inner-parallel. For 4–32 inner computations, inner-parallel is ~1.3× faster than Matryoshka. This is because this program is constrained by memory when the entire input data is processed at the same time, and hence spilling to disk occurs for Matryoshka.

As DIQL ran out of memory in all cases for 48 GB input, we ran another experiment with only 12 GB, to be able to compare execution times. Figure 6 shows the results. We observe that Matryoshka is faster than DIQL in all cases, by up to 6.6×.

9.5 Data Skew

We evaluate Matryoshka under data skew. We created skewed versions of Bounce Rate and PageRank by changing the input generation to draw the grouping keys from a Zipf (instead of uniform) distribution. This resulted in a few large groups and many small groups (1024 in total).

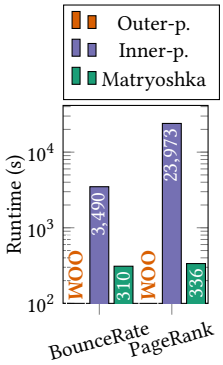


Figure 7: Skew.

Figure 7 shows the results. We observe that Matryoshka significantly outperforms both workarounds. It is 11×–71× faster than inner-parallel while outer-parallel always fails with out-of-memory. This experiment severely hits the already-explained issues of both workarounds. More interestingly, we observe that our system is not significantly affected by skew: its run times are within 15% of running on unskewed data of the same size.

9.6 Optimizations

We also study the efficiency of our optimizations discussed in Sec. 8.

InnerBag-InnerScalar Joins. We performed an experiment with PageRank to evaluate the effectiveness of Matryoshka to select the right join algorithm (broadcast vs. repartitioned) when varying the number of inner computations. Figure 8 (left) shows the results. We observe that Matryoshka's optimizer is highly effective in selecting the right algorithm at any number of inner computations. This prevents our system to fall into cases where one of the algorithms fails or is more than an order of magnitude slower than the other. For instance, the repartition join is up to 15× slower than the broadcast

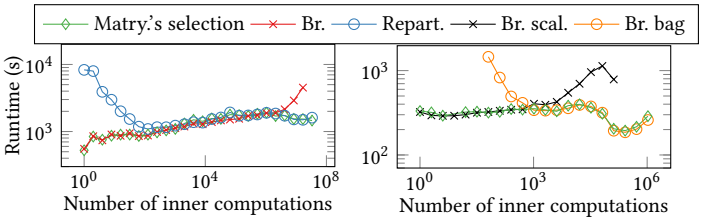
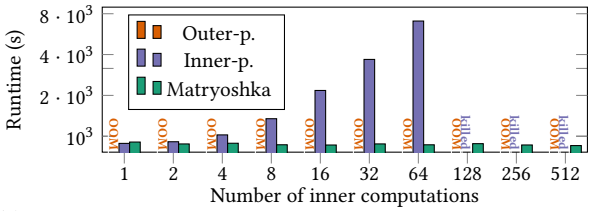
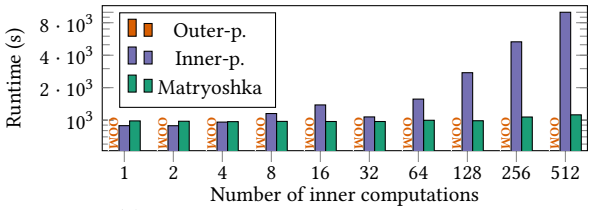


Figure 8: Optimization experiments.



(a) PageRank with input size of 160 GB. Inner-parallel was killed when the run time exceeded 10× of Matryoshka.



(b) Bounce Rate with input size of 384 GB.

Figure 9: Larger total input size.

join when the number of inner computations is small. In contrast, the broadcast join can be up to 3× slower than the repartitioned join when the number of inner computations is big. Moreover, at the end of the plot, the broadcast join fails with an out-of-memory, because it cannot fit the broadcasted dataset on a single machine.

Half-lifted MapWithClosure. We performed an experiment with K-means where we tried the different half-lifted `mapWithClosure` strategies. We can see in Figure 8 (right) that our optimizer always makes the optimal choice, which prevents Matryoshka to crash or to fall into big performance degradations (up to 4.6×).

9.7 Larger Datasets

We also used a larger cluster to run the weak scaling experiment (Sec. 9.2) with 8× larger input sizes. This cluster has 36 machines, each with two Intel Xeon E5-2630V4 CPUs (40 threads per machine). We gave 100 GB memory to each Spark worker. Figure 9 shows the results. Compared to the inner-parallel workaround, we observe similar speedups in case of PageRank as in the smaller experiments: Matryoshka gets more than one order of magnitude faster from 128 inner computations. In case of Bounce Rate, we observe almost twice as large speedups as in the smaller experiments: with 512 inner computations Matryoshka is 8.9× faster than inner-parallel. The outer-parallel workaround runs out of memory in all cases.

10 Related Work

There are several works on flatenning for handling nested parallelism [10, 20, 21, 26, 42, 46, 47], especially in the field of compilers [11, 18, 38, 41]. Smith et al. [42], MRQL [20] and DIQL [21] are the closest work to ours: these systems flatten nested-parallel queries and translate them to parallel dataflow engines. However, these systems do not support flattening in the case when there are control flow statements at inner nesting levels, which are common in modern data analysis tasks. Also, DIQL and MRQL do not perform runtime optimizations, which is crucial for achieving true flexibility in input data characteristics as shown by our experiments in Sec. 9.6. Our two-phase flattening process enables Matryoshka to perform runtime optimizations.

Other works focus on compiling from Haskell by utilizing flattening [18, 38, 46, 47]. Still, Haskell is a purely functional language and thus these works do not support imperative control flow either. Henriksen et al. [26] compile from Futhark, a purely functional array language allowing for nesting, to parallel GPU code. However, their work cannot be directly applied to dataflow engines as they require different abstractions and optimizations. Boehm et al. [14] introduced a *parallel for* construct in SystemML [13, 22], which allows for adding outer levels of task parallelism on top of the data parallelism of linear algebra operators. Still, as SystemML does not employ flattening, it can run into the problems of the inner- or outer-parallel workarounds. Similarly, Ray/RLlib [34, 35] allows for nested parallelism but does not employ flattening, which requires users to carefully control parallelization at each level. Katsogridakis et al. [29] extended Spark to launch Spark jobs from inside Spark jobs, but their system suffers the same problems as the inner-parallel workaround. Other works automate the outer-parallel workaround, but they inherit all its drawbacks [5, 6, 36].

Lastly, data skew handling in large-scale data processing is a well-studied problem [9, 33, 44], but those works are all orthogonal to ours: By flattening nested programs, we remove skew problems that would arise when using the inner- or outer-parallel workarounds. Our system could thus benefit from any general skew technique in dataflow engines, e.g., Hurricane's task cloning can mitigate skew issues in joins or grouped aggregations [9]. Smith et al. [42] handle skew by separating a bag into light and heavy keys, and executing different operator implementations for the two cases.

11 Conclusion

Although modern data analytics tasks often involve nested-parallel operations, current parallel dataflow engines do not natively support them. Users, thus, utilize different workarounds that parallelize on only one level, which typically does not yield optimal performance. We presented Matryoshka, a system that takes a nested-parallel task as input and outputs an equivalent flat program, which can be executed efficiently on an existing dataflow engine. It frees users from the burden of implementing and choosing between workarounds. Our experimental evaluation showed that it provides uniform performance across varying data characteristics, e.g., (skewed) inner computation sizes. It is up to two orders of magnitude faster than baselines (the DIQL system as well as the outer- and inner-parallel workarounds) and achieves nearly linear scalability. In particular, the results showed that Matryoshka can flatten programs that DIQL is not able to flatten.

Acknowledgments

We thank Ádám Kunos for advice on some mathematics terminology. This work was funded by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

References

[1] D. Agrawal, M. L. Ba, L. Berti-Équille, S. Chawla, A. K. Elmagarmid, H. Hammady, Y. Idris, Z. Kaoudi, Z. Khayyat, S. Kruse, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki. Rheem: Enabling Multi-Platform Task Execution. In F. Özcan, G. Koutrika, and S. Madden, editors, *SIGMOD*, pages 2069–2072, 2016.

[2] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. K. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi. RHEEM: enabling cross-platform data processing - may the big data be with you! *PVLDB*, 11(11):1414–1427, 2018.

[3] D. Agrawal, S. Chawla, A. K. Elmagarmid, Z. Kaoudi, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and M. J. Zaki. Road to Freedom in Big Data Analytics. In *EDBT*, pages 479–484, 2016.

[4] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *VLDB Journal*, 23(6):939–964, 2014.

[5] A. Alexandrov, G. Krastev, and V. Markl. Representations and optimizations for embedded parallel dataflow languages. *ACM Transactions on Database Systems (TODS)*, 44(1):4, 2019.

[6] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61. ACM, 2015.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[8] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[9] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[10] G. E. Blelloch. *Vector models for data-parallel computing*, volume 2. MIT press Cambridge, 1990.

[11] G. E. Blelloch. *NESL: a nested data parallel language.* Carnegie Mellon Univ., 1992.

[12] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[13] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

[14] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *Proceedings of the VLDB Endowment*, 7(7):553–564, 2014.

[15] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[16] E. Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*. ACM, 2013.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[18] M. M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal – nested data parallelism in Haskell. In *European Conference on Parallel Processing*, pages 524–534. Springer, 2001.

[19] T. Desautels, A. Krause, and J. W. Burdick. Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization. *Journal of Machine Learning Research*, 15:3873–3923, 2014.

[20] L. Fegaras. An algebra for distributed big data analytics. *Journal of Functional Programming*, 27, 2017.

[21] L. Fegaras and M. H. Noor. Compile-time code generation for embedded data-intensive query languages. In *2018 IEEE International Congress on Big Data (BigData Congress)*, pages 1–8. IEEE, 2018.

[22] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[23] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[24] G. E. Gévay, T. Rabl, S. Breß, L. Madai-Tahy, J.-A. Quiané-Ruiz, and V. Markl. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *IEEE 37th International Conference on Data Engineering (ICDE)*, 2021.

[25] T. H. Haveliwala. Topic-sensitive PageRank: A context-sensitive ranking algorithm for web search. *IEEE transactions on knowledge and data engineering*, 15(4):784–796, 2003.

[26] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 53–67, 2019.

[27] K. Jamieson and A. Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[28] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing PageRank. Technical report, Stanford, 2003.

[29] P. Katsogridakis, S. Papagiannaki, and P. Pratikakis. Execution of recursive queries in Apache Spark. In *European Conference on Parallel Processing*, pages 289–302. Springer, 2017.

[30] A. Kaushik. 'Bounce Rate' as the Sexiest Web Metric Ever. http://www.marketingprofs.com/7/bounce-rate-sexiest-web-metric-ever-kaushik.asp. [Online; accessed 29-May-2020].

[31] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A system for big data cleansing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1215–1230. ACM, 2015.

[32] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, 2017.

[33] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2012.

[34] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 2018.

[35] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[37] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[38] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[39] P. Pistor and F. Andersen. Designing a generalized NF2 model with an SQL-type language interface. In *VLDB*, volume 86, pages 25–28. Citeseer, 1986.

[40] D. Sculley, R. G. Malkin, S. Basu, and R. J. Bayardo. Predicting bounce rates in sponsored search advertisements. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1325–1334, 2009.

[41] A. Slesarenko. Lightweight polytypic staging: a new approach to nested data parallelism in Scala. *Scala Days*, 2012.

[42] J. Smith, M. Benedikt, M. Nikolic, and A. Shaikhha. Scalable querying of nested data. *Proceedings of the VLDB Endowment*, 14(3):445–457, 2021.

[43] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380, 2015.

[44] Z. Tang, X. Zhang, K. Li, and K. Li. An intermediate data placement algorithm for load balancing in Spark computing environment. *Future Generation Computer Systems*, 78:287–301, 2018.

[45] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. *SIGMOD Record*, 49(4):6–11, 2020.

[46] A. Ulrich. *Query Flattening and the Nested Data Parallelism Paradigm.* PhD thesis, Universität Tübingen, 2018.

[47] A. Ulrich and T. Grust. The flatter, the better: Query compilation based on the flattening transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1421–1426. ACM, 2015.

[48] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

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

[51] Spark GraphX – connected components library function. https://github.com/apache/spark/blob/branch-3.0/graphx/src/main/scala/org/apache/spark/graphx/lib/ConnectedComponents.scala#L34, 2013. [Online; accessed 21-Sep-2020].

[52] Flink Gelly – connected components library function. https://github.com/apache/flink/blob/2d10acf8189309edc42d57d603887a3431a2ae18/flink-libraries/flink-gelly/src/main/java/org/apache/flink/graph/library/ConnectedComponents.java#L45, 2017. [Online; accessed 28-Feb-2020].