Nested Parallelism and Control Flow in Big Data Analytics Systems

vorgelegt von
M. Sc.
Gábor Etele Gévay
ORCID: 0000-0001-6915-644X

an der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Klaus-Robert Müller
Gutachter: Prof. Dr. Volker Markl
Gutachter: Prof. Dr. Torsten Grust
Gutachter: Prof. Dr. Matthias Boehm

Tag der wissenschaftlichen Aussprache: 13. Mai 2022

Berlin 2022

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Dr. Volker Markl. He provided a great work environment and always supported me, at some point believeing in my research ideas more than I did.

Alexander Alexandrov created the Emma system, which provided the context for both of my main research topics. He pointed my attention to static single assignment form, which is an essential ingredient in my work on control flow. He also introduced me to the flattening technique, which forms the basis of my work on nested parallelism.

Jorge-Arnulfo Quiané-Ruiz helped a lot in presenting my research in the form of papers that actually get accepted. When he started guiding me, my PhD took a sharp turn for the better.

I am thankful to many other people who helped me in various ways during my PhD. Tilmann Rabl and Sebastian Breß advised me when writing the early versions of the Mitos paper. Juan Soto helped in the writing process of the control flow survey paper. Loránd Madai-Tahy helped in the implementation of Mitos' compilation. Georgi Krastev helped mainly through his work on Emma. Ádám Kunos gave me advice about mathematics terminology in the proof sketches for Matryoshka's completeness and correctness. We had a fruitful collaboration with Muhammad Imran on two Datalog papers (not part of this thesis). I had fun discussions with Gábor Hermann on some stream processing research ideas, which unfortunately did not make it into a paper. Eleni Tzirita Zacharatou suggested the system name Mitos, which brought really good luck.

I had exceptionally good programming teachers in primary school and high school: Ferenc Beke and Zsolt Fodor. I am grateful to them, as they undoubtedly played a big role in my love for computer science.

Last but not least, I would like to thank my family and friends. My father sparked my interest in science when I was little, and my grandfather bought my first computer.

Declaration of Authorship

I, Gábor E. Gévay, declare that this thesis, titled "Nested Parallelism and Control Flow in Big Data Analytics Systems", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Berlin, May 17, 2022 .....

Abstract

Over the last 15 years, numerous distributed dataflow systems appeared for large-scale data analytics, such as Apache Flink and Apache Spark. Users of such systems write data analysis programs in a (more or less) high-level API, while the systems take care of the low-level details of executing the programs in a scalable way on a cluster of machines. The systems' APIs consist of distributed collection types (or distributed matrix, graph, etc. types), and corresponding parallel operations.

Distributed dataflow systems work well for simple programs, which are straightforward to express by just a few of the system-provided parallel operations. However, modern data analytics often demands the composition of larger programs, where 1) parallel operations are surrounded by control flow statements (e.g., in iterative algorithms, such as PageRank or K-means clustering), and/or 2) parallel operations are nested into each other. In such cases, an unpleasant trade-off appears: we lose either performance or ease-of-use: If users compose these complex programs in a straightforward way, they run into performance issues. Expert users might be able to solve the performance issues, albeit at the cost of a significant effort of delving into low-level execution details.

In this thesis, we solve this trade-off for the case of control flow statements as follows: Our system allows users to express control flow with easy-to-use, standard, imperative control flow constructs, and it compiles the program into a single dataflow job. Having a single job eliminates the job launch overhead from iteration steps, and enables several loop optimizations. We compile through an intermediate representation based on static single assignment form, which allows us to handle all the standard imperative control flow statements in a uniform way. A run-time component of our system coordinates the distributed execution of control flow statements, using a novel coordination algorithm, which leverages our intermediate representation to handle any imperative control flow.

Furthermore, for handling nested parallel operations, we propose a compilation technique that flattens a nested program, i.e., creates an equivalent flat program where there is no nesting of parallel operations. The flattened program can then be executed on a standard distributed dataflow system. Our main design goal was to enable users to nest any data analysis program inside a parallel operation without changes, i.e., to not introduce significant restrictions on how the system's API can be used at inner nesting levels. An important example is that, contrary to previous systems that perform flattening, we can even handle programs where there is an iterative algorithm at inner nesting levels. We also show three optimizations, which solve performance problems that arise when applying the flattening technique in the context of distributed dataflow systems.

Zusammenfassung

In den letzten 15 Jahren sind zahlreiche verteilte Datenflusssysteme für die groß angelegte Datenanalyse entstanden, wie Apache Flink und Spark. Die Benutzer solcher Systeme schreiben Datenanalyseprogramme in einer (mehr oder weniger) hochrangigen API, und die Systeme kümmern sich um die Low-Level-Details der Ausführung der Programme in einer skalierbaren Weise auf einem Cluster von Maschinen. Die APIs der Systeme bestehen aus verteilten Sammlungstypen (oder verteilten Matrix-, Graphen- usw. Typen) und dazugehörigen parallelen Operationen.

Verteilte Datenflusssysteme eignen sich gut für einfache Programme, die sich leicht durch einige wenige der vom System angebotenen parallelen Operationen ausdrücken lassen. Die moderne Datenanalyse erfordert jedoch häufig die Komposition größerer Programme, in denen 1) parallele Operationen von Kontrollflussanweisungen umgeben sind (z.B. in iterativen Algorithmen wie PageRank oder K-means-Clustering), und/oder 2) parallele Operationen ineinander verschachtelt sind. In solchen Fällen kommt es zu einem unangenehmen Trade-off: Wir verlieren entweder Leistung oder Benutzerfreundlichkeit: Wenn Benutzer diese komplexen Programme auf einfache Art und Weise komponieren, stoßen sie auf Leistungsprobleme. Erfahrene Benutzer können die Leistungsprobleme lösen, wenn auch um den Preis, dass sie sich mit den Details der Ausführung auf niedriger Ebene befassen müssen.

In dieser Arbeit lösen wir diesen Trade-off für den Fall von Kontrollflussanweisungen wie folgt: Unser System ermöglicht es den Benutzern, den Kontrollfluss mit einfach verwendbaren, standardmäßigen, imperativen Kontrollflusskonstrukten auszudrücken, und es kompiliert das Programm in einen einzelnen Datenflussjob. Dieser Einzelne Job eliminiert den Job-Start-Overhead von Iterationsschritten und ermöglicht verschiedene Schleifenoptimierungen. Wir kompilieren über eine Zwischenrepräsentation, die auf der Static-Single-Assignment-Darstellung basiert, und es uns ermöglicht alle standardmäßigen imperativen Kontrollflussanweisungen auf eine Weise zu behandeln. Eine Laufzeitkomponente unseres Systems koordiniert die verteilte Ausführung des Kontrollflusses mit Hilfe eines neuartigen Koordinationsalgorithmus, der unsere Zwischenrepräsentation nutzt, um beliebigen imperativen Kontrollfluss zu verarbeiten.

Darüber hinaus schlagen wir für den Umgang mit verschachtelten parallelen Operationen eine Kompilierungstechnik vor, die ein verschachteltes Programm abflacht, d.h. ein äquivalentes flaches Programm erzeugt, in dem es keine Verschachtelung von parallelen Operationen gibt. Das abflachte Programm kann dann auf einem standardmäßigen verteilten Datenflusssystem ausgeführt werden. Unser Hauptziel bei der Entwicklung war es, den Benutzern die Möglichkeit zu geben, jedes beliebige Datenanalyseprogramm ohne Änderungen innerhalb einer parallelen Operation zu verschachteln, d.h. keine wesentlichen Einschränkungen bei der Verwendung der API des Systems auf den inneren Verschachtelungsebenen einzuführen. Ein wichtiges Beispiel ist, dass wir sogar Programme mit einem iterativen Algorithmus auf inneren Verschachtelungsebenen handhaben können, im Gegensatz zu früheren Systemen mit Abflachung. Wir zeigen auch drei Optimierungen, die Leistungsprobleme lösen, die bei der Anwendung der Abflachungstechnik im Kontext von verteilten Datenflusssystemen auftreten.

Contents

Chapter 1

Introduction

The success of distributed dataflow systems depends on both their performance (efficiency, scalability) and ease-of-use. However, there is often a tension between these requirements: when designing a system, we often have to place restrictions on how a user is allowed to write her programs if she expects good performance. For example, executing an iterative algorithm is more efficient in a single dataflow job, rather than launching a series of dataflow jobs, one for each iteration step [1, 65, 72, 132]. To be able to incorporate a loop into a single dataflow job, many systems offer functional loop APIs [8, 47, 65, 132, 154, 186]. These APIs are harder to use than the standard, imperative control flow constructs, but if users want good performance then they have no choice but to learn and use the functional loop APIs.

Another example for the tradeoff between user effort and performance is the handling of nested collection operations. Distributed dataflow systems offer parallel collection operations that cannot be nested, e.g., a `map` cannot be inside a `map`. Therefore, if a user has a task where nesting appears, she has two main choices:

(1) *parallelize only one level*: implement only one level using the system's parallel operations, and use non-parallel operations at the other level, or

(2) *flatten the program manually*: transform the program into an equivalent program that has no nesting of either collections or collection operations.

(1) is relatively easy to do, but the program will probably run into performance issues (or out-of-memory crashes), since a significant part of the program is not parallelized. (2) avoids the performance issues, but it can be much harder, especially when the inner nesting level involves control flow, such as an iterative algorithm. If the inner level is calling a library (e.g., a graph- or machine learning library), then this makes it even harder to flatten the program manually. In such a case, the user who is calling the library will not have the necessary expertise on the library's internals to perform the flattening. Conversely, the library's implementor will not have the necessary context, as she does not see the calling code.

The vision of this PhD work was a system where the above trade-offs are solved: Without compromising on performance, the user can freely nest parallel collections and parallel operations, and can write easy-to-use imperative control flow constructs anywhere, even inside parallel operations. We tackled this challenge by separating the efficient control flow handling from the nested parallelism, and addressing them individually in two separate systems. The thesis' main contributions are

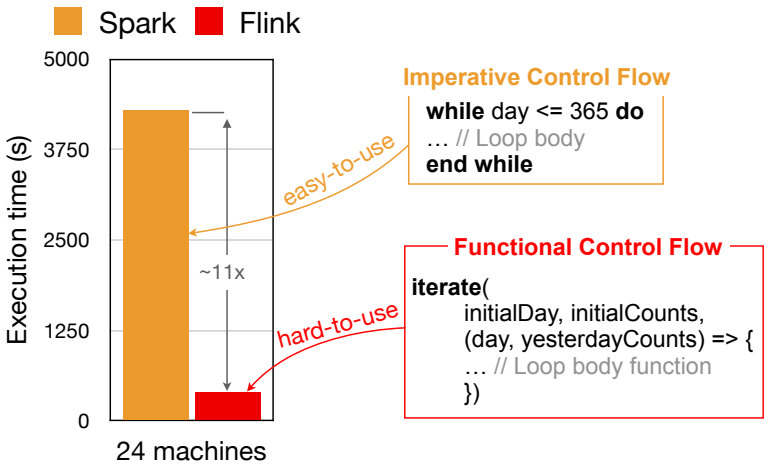


Figure 1.1: Imperative vs. functional control flow.

(1) compiling imperative control flow to efficient dataflows (Mitos system, Chapter 3);

(2) compiling nested-parallel programs to flat-parallel programs, which can then be executed on a standard dataflow engine (Matryoshka system, Chapter 4);

(3) a comprehensive survey on control flow in distributed dataflow systems (Section 5.1).

In the rest of the Introduction, we will present an overview of these topics.

1.1 Control Flow in Distributed Dataflow Systems

Modern data analytics heavily relies on control flow statements. For example, many graph analysis tasks are iterative, such as PageRank [140] or computing connected components by label propagation [98]. Other data science pipelines are also often composed of iterative programs [182]. K-means clustering [182] and gradient descent [195] are just two of the most commonly occurring iterative tasks. Additionally, control flow is just getting more complex: An iterative machine learning training task can be inside another loop for hyperparameter optimization or k-fold cross-validation; a nested loop can also appear inside a single algorithm, such as the coloring algorithm for computing strongly connected components [137]; algorithms may contain if statements inside loops, such as in simulated annealing [104].

However, despite that control flow statements are at the core of modern data analytics, supporting control flow efficiently and effectively is still a weakness of dataflow systems: They either suffer from poor performance or are hard to use. On the one hand, in some systems, such as Apache Spark, users express loops inside the driver program, using the standard, *imperative* control flow constructs. Although this imperative approach is easy to use, it launches a new dataflow job for every iteration step, which hurts performance because of a high inherent job launch overhead. On the other hand, some other systems, such as Apache Flink, provide *native control flow* support [65], i.e., users can include loops in their (cyclic) dataflow jobs. This removes the job launch overhead, which is present in Spark, resulting in much better performance. However, this high performance comes at a price: Users have to express loops by calling higher-order functions, which are harder to use than the imperative control flow of Spark.

To better illustrate this problem, we ran an experiment to evaluate Spark and Flink, using a program that computes the visit counts from a year of page visit logs. This program has a loop that reads a different file at each iteration step and compares the visit counts with the previous day[1]. Figure 1.1 shows the results of this experiment. We observe that Spark is more than an order of magnitude slower than Flink because it does not support native loops. Spark launches a new dataflow job for every iteration step, incurring a high overhead. However, on the other side, Flink is harder to use than Spark. In Flink users call the `iterate` higher-order function and give the loop body as an argument (see the functional control flow box in Figure 1.1). The loop body is a function that builds the dataflow job fragment representing the actual loop body operations. This API is hard for non-expert users, such as data scientists[2]. In contrast, users prefer the imperative control flow present in Spark, similar to, e.g., Python, R, or Matlab (see the imperative control flow box in Figure 1.1).

Ideally, the system should allow users to express control flow using simple imperative control flow statements, while matching the performance of native control flow. In other words, we want a system that marries the ease-of-use of Spark with the high efficiency of Flink. The research community has paid attention to this problem and recently proposed a number of solutions [11, 92, 125]. For example, Emma [9, 11, 12] can translate imperative control flow to Flink's native loops, but only when there is a single while-loop without any other control flow statement in its body. This makes it not suitable for many tasks in modern data analytics, such as hyper-parameter optimization, simulated annealing, and strongly connected components [137]. AutoGraph [125] and Janus [92] compile imperative control flow to TensorFlow's native loops [186]. However, they do not support general data analytics other than machine learning.

Supporting general imperative control flow (e.g., iterative tasks) without sacrificing performance is challenging for two main reasons. First, normally a dataflow job is built from just the method calls (e.g., `map`, `join`) that the user program makes to the system. However, to build a complete cyclic dataflow job from imperative control flow, the system also needs to inspect other parts of the user code, such as the control flow statements: It also has to insert special nodes and edges into the dataflow job for such parts of the code. More specifically, the system needs to add a) back-edges for loops to pass data between iteration steps inside the dataflow job, b) condition nodes that determine control flow (e.g., loop exit conditions), c) nodes that handle control flow merge points (e.g., when a variable is assigned in two different if-branches, and then later used after the if statement), d) nodes for scalar variables (scalar variables are originally handled by the driver program, but now we need to bring them into the dataflow job).

Second, to fully take advantage of the entire program being in a single dataflow job, we want to support loop pipelining, i.e., overlapping subsequent executions of a loop body. This means that we cannot simply insert a full synchronization barrier between iteration steps, and just reset all operators at the barrier. Instead, we need to deal with different operators (and their different physical instances) processing different iteration steps at the same time.

We present Mitos[3], a system where control flow support matches Spark's ease-of-use, and that significantly outperforms both Spark and Flink. Specifically, it outperforms Spark because of native loops, and it outperforms Flink's native loops because of loop pipelining. Mitos uses

[1] We provide the details of this experiment in Section 3.6 and provide the code for both the imperative and functional control flow APIs in Listing 3.1.

[2] A simple search on stackoverflow.com for the terms *Flink iterate* or *TensorFlow while_ loop* shows that a large number of users are indeed confused by such a functional control flow API.

[3] The name comes from Greek mythology: Mitos is the thread that Ariadne gave to Theseus to help him get out of the labyrinth.

compile-time metaprogramming to parse an imperative user program into an intermediate representation (IR) that abstracts away specific control flow constructs. This IR facilitates the building of a single (cyclic) dataflow job from any program with imperative control flow. At run time, Mitos coordinates the distributed execution of control flow statements using a novel coordination algorithm that leverages our IR to handle any general imperative control flow. In summary, we make three major contributions:

1. We present a compilation approach based on metaprogramming to build a single dataflow job of a distributed dataflow system from a program with general imperative control flow statements. Specifically, we leverage Scala macros [37] to inspect and rewrite the user program's abstract syntax tree such that the system can produce a single dataflow job. By this, we can bring the power of native control flow to data scientists, who like to use high-level languages that have imperative control flow statements. (Section 3.3)

2. We devise a mechanism that coordinates and communicates the control flow decisions between machines in a non-intrusive manner. In particular, our coordination mechanism enables two core optimizations that speed up the dataflow job execution: loop pipelining, i.e., overlapping iteration steps, and loop-invariant hoisting, i.e., reusing loop-invariant (static) datasets during subsequent iteration steps. As a result, our system not only supports any control flow statement but also outperforms dataflow systems with native control flow support. (Section 3.4)

3. We experimentally evaluate Mitos using real tasks (Visit Count and PageRank) and microbenchmarks. We mainly compare its performance to Flink (as a system supporting native control flow) and Spark (as a system providing ease-of-use). Our results show that Mitos is more than one order of magnitude faster than Spark, and, surprisingly, it is also up to 10.5× faster than Flink (the system with native control flow support). (Section 3.6)

1.2 Nested Parallelism in Distributed Dataflow Systems

The success of distributed dataflow engines, such as Spark [188, 189] and Flink [10, 41], 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.

Yet, these systems do not support nested parallelism, i.e., launching a parallel operation from the inside of another parallel operation. For instance, the UDF of a `map` operator cannot invoke further parallel operations, such as another `map` operator. We now explain three cases when nested parallelism 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 [26] 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 [25] and hyperparameter optimization on a second, outer level [26, 171]. 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.

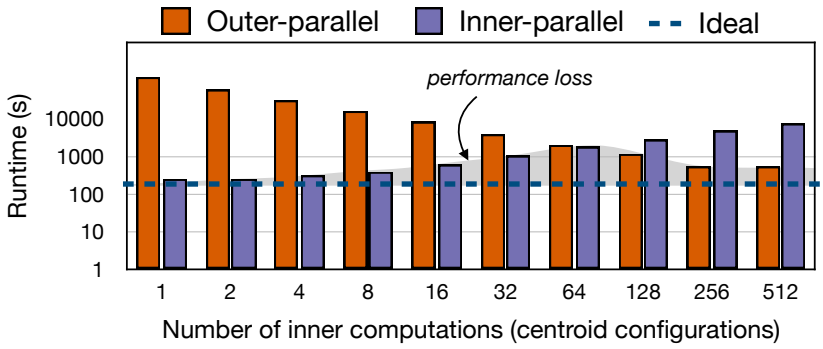


Figure 1.2: K-means run times.

As by design distributed 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 Resilient Distributed Dataset (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 `DataSet`s. Many existing systems which support nesting of operations in their languages, actually employ one of these two workarounds when executing programs [11, 26, 100, 136]. Some systems [66, 67, 84, 176] natively support nested parallelism, but they do not support iterative computations at inner nesting levels, which is a typical requirement in modern data analysis tasks, such as K-means [182] clustering or PageRank [140].

Unfortunately, choosing between the outer- and inner-parallel workarounds for the task at hand is far from easy. 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 1.2 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 [26] to choose between the workarounds. However, both workarounds are far from the ideal performance (the blue line in Figure 1.2), 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., loops). Specifically, we make the following major contributions:

1. We devise a novel two-phase flattening process (Section 4.3) that translates a nested-parallel program into a highly efficient flat program, which can run on an existing dataflow engine (which is unmodified Spark in our implementation). Our two-phase flattening process comes with a set of nesting primitives that allow us to select the best physical operator implementations at run time. We then present techniques to deal with closures in UDFs (Section 4.4).

2. We show how to flatten programs even in the presence of control flow statements (e.g., the loop 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. (Section 4.5)

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. (Section 4.6)

4. We experimentally validate our system using five common data analytics tasks and compare it to DIQL [67], SystemDS's parallel loops [24, 26], 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 and scales better than the baselines. (Section 4.8)

1.3 Publications and Other Contributions

Most of the material in this thesis is based on the following publications. Mitos (Chapter 3) was published at ICDE 2021 [72], with a *best paper award* and an *ACM SIGMOD Research Highlight Award*. Matryoshka (Chapter 4) was published at SIGMOD 2021 [71]. The part of the related work chapter concerning control flow handling (Section 5.1) is based on a survey paper in ACM Computing Surveys [73].

During my PhD, I contributed to some other publications, which are not part of this thesis. Cog [87] compiles from Datalog to Flink's delta iterations. Subsequently, we extended Cog with aggregations, asynchronous iterations, and streaming [88]. I also contributed to a book chapter on large-scale data stream processing systems [40].

Additionally, I made some open-source contributions to Emma[4] and Apache Flink[5]. A part

[4] `https://github.com/emmalanguage/emma/graphs/contributors`
[5] `https://github.com/apache/flink/graphs/contributors`

of my contributions to Emma is Mitos.

1.4 Structure of the Thesis

Chapter 2 provides some background knowledge and terminology, which we rely on later. Chapters 3 and 4 present Mitos and Matryoshka. Chapter 5 discusses related work, and Chapter 6 concludes.

Chapter 2

Background and Terminology

This section reviews some background knowledge and terminology, which we rely on throughout the thesis.

2.1 Iteration vs. Iteration Step

A loop (or iteration) is the repeated execution of a block of programming instructions. Such a block is called *loop body*. We will use the term *iteration step* to mean a single execution of a loop body. Note that in other works it is also common practice to use *iteration* for both *loops* and *iteration steps*. However, we we will avoid this usage to prevent ambiguity.

2.2 Distributed Dataflow Systems

Distributed Dataflow Systems (DDSs) have become a standard technology in the last 15 years. These systems model distributed computation by *dataflow jobs*, which are directed graphs, where nodes represent computation and edges represent data flowing between nodes. A *logical* dataflow graph is executed in parallel by creating many *physical* tasks from each logical node and distributing these on a compute cluster.

Typically, DDSs run on shared-nothing clusters, with a single machine designated as the *main* node and several machines designated as *worker* nodes [156]. The main node coordinates distributed program execution, whereas the workers perform the actual processing. *Driver* programs (also known as *clients*) run on a single machine (which can also coincide with the main node). Drivers submit dataflow jobs (or just *jobs*, for short) to the system via the main node. These jobs describe the distributed programs that the system should execute. For example, in MapReduce, jobs include map- and reduce functions as well as input- and output paths. Note that driver programs can submit multiple jobs to a system. For example, the simplest approach to implement an iterative algorithm is to manage the loop in the driver program and submit separate jobs for each iteration step.

Modern DDS typically have collection-based APIs [187], where each dataflow node is represented by an object of a special collection type[1], such as RDDs [188] in Spark. Users build dataflow jobs by creating these (immutable) distributed collections. Distributed collections are created via one of three means:

[1] Mathematically, these are *multisets* or *bags*, which are unordered collections of elements, allowing duplicates.

- read input data from external systems, such as the Hadoop Distributed File System (HDFS) [164],
- convert a collection from the host language (e.g., a standard array) to the system's distributed collection type,
- transform an existing distributed collection (e.g., by calling `map`, which takes a so-called *User-Defined Function* (UDF), applies the function to all elements of a collection, and returns all of the results as a new distributed collection).

The above are *lazy operations*, i.e., they do not perform their actual work immediately, but just add nodes and edges to a dataflow job. Once a special operation called an *action* is invoked, the dataflow job is executed. Example actions include aggregating a collection into a single value, writing a collection to an external system (such as HDFS), and fetching a collection into the driver program as a non-distributed, standard collection, such as a Java list.

There are two main approaches to implement loops. In one approach, a loop is implemented as separate dataflow jobs in each iteration step. In this case, a driver program calls an action (e.g., performing an aggregation as part of evaluating the termination condition) at each step. In the second approach, which we call in-graph loops, the focus is on incorporating the loop (and other control flow) into the dataflow job itself. In this case, an action is *not* called for each step, but rather only once for the entire loop.

Note that implementing a loop as separate dataflow jobs does not require built-in loop support from the system. This means that we can use this as a workaround to implement loops when using almost any system. However, without support from the system, performance is often suboptimal due to missed optimization opportunities and job launch overhead at each step.

2.3 Domain-Specific Language Design Approaches

Domain-Specific Languages (DSLs) are specially designed languages for a particular use. SQL is a well-known example of a DSL that was created specifically for data processing.

We distinguish between *external* DSLs and *embedded* DSLs. External (a.k.a. standalone) DSLs are independent of a general-purpose programming language. Examples include SQL, Hive, and SystemDS' DML language [24, 25, 74]. In contrast, embedded (a.k.a. internal) DSLs are built on top of a host programming language. Examples include LINQ embedded in C#, or Flink's and Spark's APIs embedded in Java, Scala, and Python. Embedded DSLs can be further subdivided into type-based and metaprogramming-based embedded DSLs. Next, we discuss each of these categories of DSLs.

External DSLs have their own syntax, compiler, and possibly other tools, such as a debugger. Their main advantage is that they offer greater flexibility in the language design, since they are not restricted by a given host language. However, they have several drawbacks. One, they impose a considerable effort on language designers, since they have to implement every tool related to the language. Two, users need to expend more time to learn the syntax. Third, interoperability with existing code in a general-purpose programming language is more difficult to achieve.

Type-based embedded DSLs (sometimes called library-based embedding) consist of types created by the language designer in a host language. Examples of such DSLs include the RDD API of Spark and the *DataSet* API of Flink, where the DSL is a Java or Scala API expressed as classes and methods. Unfortunately, type-based embeddings constrain systems in that they

are only aware of those parts of a program that are expressed as classes (types) and method calls. For example, Spark is unaware of loops and other control flow statements because these are expressed using built-in language features of the host language, such as Java or Scala *while* loops. Consequently, Spark is unable to compile an iterative program to an in-graph loop, which yields lower performance. *Language virtualization* [43,52,128,151] solves this problem by changing built-in language constructs (e.g., loops) to be overloadable, i.e., allow the DSL designer to give them a special meaning. This allows the DSL designer to *stage* [152,153] a loop, i.e., make the looping construct build a representation of the loop instead of executing it immediately, and thus allowing the system to optimize the execution by, e.g., making it an in-graph loop.

Metaprogramming-based embedded DSLs (also called quotation-based DSLs) rely on source code analysis and transformations [11,126,134,173]. In contrast to the simple type-based embeddings of Spark or Flink, a metaprogramming-based DSL can examine any part of the program, including control flow constructs. Consequently, systems are able to compile programs to in-graph loops or otherwise optimize programs by taking control flow into account (e.g., via cache call insertion [11]). Note that metaprogramming-based DSLs often include similar types as type-based embeddings, and therefore they typically operate also at the type level.

In Mitos and Matryoshka, we rely on Emma's metaprogramming infrastructure [9], which is based on Scala macros [37]. The user wraps the entire program in a call to Emma's *parallelize* macro, which transforms the program in order to execute it on a distributed dataflow system. Emma's API has a scalable collection type, which we will refer to as *bag*. When the *parallelize* macro is not called on the program, bags have a simple default implementation, which is single-threaded and runs on the local machine. When the *parallelize* macro is called, it changes bags to be implemented by the backend dataflow system's distributed collection type, such as RDD in Spark. The user can temporarily remove the *parallelize* macro call when working with small data for debugging purposes.

2.4 Compiler Concepts

In this section we briefly discuss some compiler concepts that we rely on later.

2.4.1 Control Flow Analysis

A *basic block* [6] is a maximal continuous sequence of instructions which always execute one after the other. This means that the execution can only enter a basic block at its beginning and only leave it at its end. For example, the body of a while-loop is a basic block if there are no other control flow instructions inside.

The *control flow graph* [6] of a program is a graph whose nodes correspond to the basic blocks of the program, and whose edges show the possible control flow paths. Specifically, let u and v be two nodes, with corresponding basic blocks U and V. A directed edge goes from u to v, if control flow can directly go from U to V. For example, in case of an if statement, edges go from the node of the basic block before the if statement to the nodes of the basic blocks of the then- and else-branches.

A *control flow merge point* [6] is a basic block whose corresponding node in the control flow graph has an in-degree of at least two. For example, the basic block after the two branches of an if statement is a merge point, because at this point the different possible paths merge.

$a = 0$	$a_1 = 0$
$b = a + 1$	$b_1 = a_1 + 1$
$a = 5$	$a_2 = 5$
$c = a + 1$	$c_1 = a_2 + 1$

(a) No control flow.

$a = 0$	$a_1 = 0$
if ... **then**	**if** ... **then**
$a = a + 1$	$a_2 = a_1 + 1$
else	**else**
$a = a + 2$	$a_3 = a_1 + 2$
end if	**end if**
$b = a + 5$	$a_4 = \Phi(a_2, a_3)$
	$b_1 = a_4 + 5$

(b) Program with an *if* statement.

$i = 1$	$i_1 = 1$
do	**do**
$i = i + 1$	$i_2 = \Phi(i_1, i_3)$
while $i < 100$	$i_3 = i_2 + 1$
$a = i$	**while** $i_3 < 100$
	$a_1 = i_3$

(c) Program with a loop.

Listing 2.1: Three examples of transforming a program to SSA.

2.4.2 Static Single Assignment Form

Static Single Assignment form (SSA) is a widely used intermediate representation in compilers [148]. The basic defining characteristic of SSA is that there is a one-to-one correspondence between variables and assignment statements. An important consequence of this is *referential transparency*: every variable reference refers to a value that was written to the variable at its unique assignment statement. In contrast, when a program is not in SSA, the value of a variable reference depends on its context:

1. It can depend on the position of the variable reference. Specifically, which of the assignment statements to the variable precede the reference in the program text.

2. Which of the preceding assignments was executed at run-time may depend on how the control flow proceeded.

When there is no control flow in the program, only the first issue can arise. In this case, we can transform to SSA, and thus eliminate the problem by (a) changing the variable names on the left-hand sides of assignment statements so that each of them assigns to unique variables, and (b) changing the variable references appropriately, so that they reference the variable of the most recent assignment. We can see an example of this in Listing 2.1a, where the variable a is split into a_1 and a_2.

However, if there is control flow in the program, then the second issue can also occur. In this case, after performing the change of variable names on the left-hand sides of assignments, we cannot change the references in the above-described way. This is because variable references at control flow merge points can refer to the value assigned in either one of the possible control flow paths. SSA resolves this problem by introducing a new variable at a merge point, and using a so-called Φ-function to assign a value to this new variable. The only purpose of these Φ-functions is to disambiguate these references, i.e., to choose one of their inputs based on how the control flow actually proceeded. We can see an example in Listing 2.1b. Note that Emma has static typing (because of building on Scala), which means that the different inputs of a Φ-function always have the same static type.

Note that even though there is exactly one assignment statement for each variable, variables inside a loop are assigned multiple times during the program execution. We can see an example of how SSA handles loops in Listing 2.1c.

Chapter 3

Efficient and Easy-To-Use Control Flow in Dataflow Systems

Control flow is vital in many data analysis programs. However, current dataflow systems either provide functional APIs, which are hard to use, or launch separate dataflow jobs for each iteration step, which incurs performance problems. In this chapter, we discuss how our system, Mitos, compiles programs with imperative control flow constructs into a single dataflow job, thus achieving both ease-of-use and high performance.

3.1 Running Example and Motivation

We now show an example to illustrate the problems of current dataflow systems when faced with imperative control flow. Consider a program that computes the visit counts for each page per day in a year of page visit logs. Assume that the log of each day is read from a separate file and that each log entry is a page ID, which means that someone has visited the page.

```
1: for day = 1 .. 365 do
2:   visits = readFile("PageVisitLog_" + day) // page IDs
3:   counts = visits.map(x => (x,1)).reduceByKey(_ + _)
4:   counts.writeFile("Counts_" + day)
5: end for
```

We cannot express this simple program in Flink's native loops, because Flink does not support reading and writing files inside native loops. On the other hand, implementing a loop in the driver program instead of native loops would cause each iteration step to launch a new dataflow job, which has an inherent high overhead[1] (see Spark in Figure 1.1). Note that this simple task could be solved without using a loop at all by reading all the files into a single distributed collection, and then processing all days at the same time. However, we will build a more complex example in the next paragraphs, where this approach would become tedious.

[1]Note that a new job is not launched if there is no *action* inside the loop body. However, actions are needed in most iterative algorithms to compute a loop exit condition from the current state of the algorithm. Moreover, Spark's job launch overhead is mostly the task launch overhead, which will still be present at each iteration step even without actions (see Section 3.6.5).

Now imagine that instead of just writing out the visit counts for each day separately, we want to compare the visit counts of consecutive days. For this, we replace Line 4 with the following comparison between the counts of the current and the previous days[2]:

```
4: if day != 1 then
5:   diffs =
6:       (counts join yesterdayCounts)
7:       .map((id,today,yesterday) => abs(today - yesterday))
8:   diffs.sum.writeFile("diff" + day)
9: end if
10: yesterdayCounts = counts
```

If it is not the first day, we join the current counts with the previous day's counts (Line 6). We then compute pairwise differences (Line 7), sum up the differences (Line 8), and write the sum to a file. At the end, we save the current counts so that we can use them the next day (Line 10). We can see that it is natural to use an if statement inside the loop. On top of that, we could replace the computation of visit counts (Line 3) with a more complex computation that itself involves a loop, such as PageRank [140]. This would result in having nested loops. Unfortunately, Flink does not provide native support for either nested loops or if statements inside loops. On the other side, Spark does not have native support for any control flow at all.

Yet, this program can become even more complex. Imagine we are interested only in a certain page type. As the logs do not contain information about the page type (each log line is just a page ID), we have to read a dataset containing the types of all pages before the loop. Inside the loop, we then add the line below before Line 3, which performs a join between the visits and page type datasets, and filters based on page type:

```
3: visits = (visits join pageTypes).filter(p => p.type=...)
```

It is worth noting that the *pageTypes* dataset does not change between iteration steps, i.e., it is *loop-invariant*. This clearly opens an opportunity for optimization: Even though the join method is called inside the loop, we can build the hash table of the join only once before the loop and probe it at every iteration step. This is straightforward to implement if the loop is a native loop of the system, since in this case all iteration steps are in a single dataflow job, which enables the join operator to keep the hash table throughout the entire loop. Nevertheless, we cannot express this program using Flink's native loops because of the aforementioned issues.

Listing 3.1 compares functional control flow APIs and Mitos' imperative API through the above example program. For the functional version, we show an idealized version of Flink's API: we extend it with 1) file I/O inside loops, 2) if statements, 3) support for multiple loop variables, and 4) a *Scalar* type for wrapping non-bag values to make them part of the dataflow job. However, even all these extensions cannot hide the inconvenience of the functional API, as we can see in the listing.

Note that in a functional control flow API, somewhat counter-intuitively, a user function for a loop body is always executed *exactly once*, as it only builds a dataflow job instead of doing the actual work. This invites mistakes: when writing such a loop body function, users have to

[2]The astute reader might notice that even this example could be solved without a loop, by using SQL window functions or self joins. However, for non-expert users, those approaches are feasible only if the computation for an individual day is quite simple, e.g., it can be performed as a grouped aggregation. This is not the case if, for example, we need to perform a PageRank on each day. (Expert users might still be able to make the loop-less approach work by lifting the by-day computation, see Section 4.3.2. Alternatively, Matryoshka would also help in implementing the loop-less approach.)

```
pageTypes = readFile("pageTypes")
yesterdayCounts = null
day = 1
while day ≤ 365 do
 // Read all page-visits for this day
 visits = readFile("pageVisitLog" + day) // pageIDs
 // Want to examine only pages of a certain type, so
 // we get the page types from a large lookup table:
 visits = visits.join(pageTypes).filter(p=>p.type=...)
 // Count how many times each page was visited:
 counts = visits.map(x=>(x,1)).reduceByKey(_+_)
 // Compare to previous day (but skip the first day)
 if day != 1 then
  diffs =
    (counts join yesterdayCounts)
    .map((id,today,yesterd)=>abs(today-yesterd))
  diffs.reduce(_ + _).writeFile("diff" + day)
 end if
 yesterdayCounts = counts
 day = day + 1
end while
```

(a) Imperative control flow (Mitos).

```
pageTypes = readFile("pageTypes")
initialCounts = EmptyBag
initialDay = Scalar(1) // Manually wrap non-bag value
whileLoop( // Higher-order function call
  // First two arguments are the initial values of the loop vars:
  initialDay, initialCounts,
  // Third arg is the func building the dataflow for the body:
  (day, yesterdayCounts) => {
   fileName = day.map(d => "pageVisitLog" + d)
   visits = readFile(fileName)
   visits = visits.join(pageTypes).filter(p => p.type = ...)
   counts = visits.map(x => (x,1)).reduceByKey(_ + _)
   if( // Higher-order function call
    // First arg is the func building the dataflow for the cond:
    () => day.map(d => d != 1),
    // 2nd arg is the func building the datafl for then-branch:
    () => (counts join yesterdayCounts)
      .map((id,today,yesterday) => abs(today - yesterday))
      .reduce(_ + _).writeFile("diff" + day)
   )
   day = day.map(d => d + 1)
   exitCond = day.map(d => d ≤ 365)
   // next values of the loop vars and exit cond:
   return (day, counts, exitCond)
  }
)
```

(b) Functional control flow.

Listing 3.1: A comparison of control flow APIs through the Visit Count example program.

pay attention to not accidentally write code that would directly (in the driver program) do work that is intended for every iteration step. Instead, code in loop body functions should build a representation of the work into the dataflow job. An example for this issue are non-bag variables, such as a loop counter or a learning rate. Manipulating such variables would normally happen in the driver program in systems such as Spark. However, in functional control flow APIs, the user has to pay attention to wrap these variables in system-provided types so that the system can incorporate them in the dataflow job. Mitos performs this wrapping automatically during its compilation, as explained in Section 3.3.1.

Note that loops are at the core of machine learning training algorithms and hyperparameter search. This makes Mitos an important piece in modern analytics, such as the ones targeted by Agora [174].

3.2 Mitos Overview

We present Mitos, a system that compiles a data analysis program with imperative control flow statements into a *single* dataflow job for distributed execution on a dataflow system. The main goal of Mitos is to bring ease-of-use to users while achieving high efficiency for their programs. Overall, users write their programs using imperative control flow. The system, in turn, parses an

imperative program into an intermediate representation, from which it builds a single[3] (cyclic) dataflow job. At run time, the system coordinates the distributed execution of control flow statements among workers in the underlying dataflow system. Below, we describe these steps in more detail.

Figure 3.1 illustrates the general architecture of Mitos. A user provides a data analysis program in a high-level language with imperative control flow support. We use the Emma language [9, 11, 12] because of its metaprogramming infrastructure and because it is similar to the languages of typical dataflow systems, such as Flink and Spark: The user expresses a data analysis program in Scala using a scalable collection type, which we call *bag* henceforth.

Given an imperative program, Mitos first simplifies it to make each assignment statement have only a single bag operation (e.g., a `map`). It then parses this simplified imperative program to an Intermediate Representation (IR). From there, it creates a dataflow job of a distributed dataflow system (Section 3.3). Recall that running many dataflow jobs sequentially significantly deteriorates the execution time of a program as illustrated in Figure 1.1 (the Spark case). Thus, it is crucial to generate as few dataflow jobs as possible: Mitos creates a single job for the entire program. This eliminates the overhead of scheduling and launching new tasks at every step of a loop, and it enables certain optimizations, such as loop pipelining.

Next, the system sends the job for execution to the underlying dataflow system. Then, Mitos coordinates the distributed execution of control flow statements via two components: the *Control Flow Manager* and the *Bag Operator Host* (Section 3.4). The control flow manager communicates control flow decisions among the worker machines. (This does not involve the driver program. Since we compile the entire program to a single dataflow job, control is not returned to the driver program at every step of a loop.) The bag operator host bridges the gap between Mitos' and the underlying dataflow system's operators. While Mitos' operators take input bags and compute output bags, the underlying dataflow system's operators do not know about bags. The bag operator host provides an interface for implementing Mitos' operators at the level of bags instead of directly with the dataflow system's operator interface. Note that our control flow coordination enables loop pipelining, i.e., overlapping different iteration steps.

Generality for Backends. Although we use Flink as our target dataflow system, Mitos could be implemented with other distributed dataflow systems as backends. It only requires a dataflow system that allows for arbitrary stateful computations in the dataflow vertices, and supports arbitrary cycles in the dataflow graph. Examples of systems that support cycles are Flink, Naiad [132], Dandelion [154], and TensorFlow. (Spark does not support cyclic dataflow jobs, and therefore it could not be the backend of Mitos.) Note that, for Mitos' loop pipelining to have a significant effect, the system should support pipelining, i.e., starting an operator execution already when just a part of its input data has arrived.

Generality for Languages. Although we use the Emma language [9, 11, 12] for Mitos, one could use other high-level data analytics languages that have imperative control flow support. Importantly, the language should provide the system with means to get information about the imperative control flow statements. In the case of Emma, this is achieved by compile-time metaprogramming. Specifically, we use Scala macros [37]. Julia [18] and Python [127] also have the required metaprogramming capabilities. Alternatively, SystemML [25] could also be integrated with Mitos. SystemML's language is an *external* [11] domain-specific language, and thereby SystemML's compiler can naturally inspect the control flow.

[3]Mitos builds a single dataflow job even if the program is composed of multiple functions. This is facilitated by Emma's compiler infrastructure, which inlines function calls before Mitos performs its compilation. We do not support recursive functions.

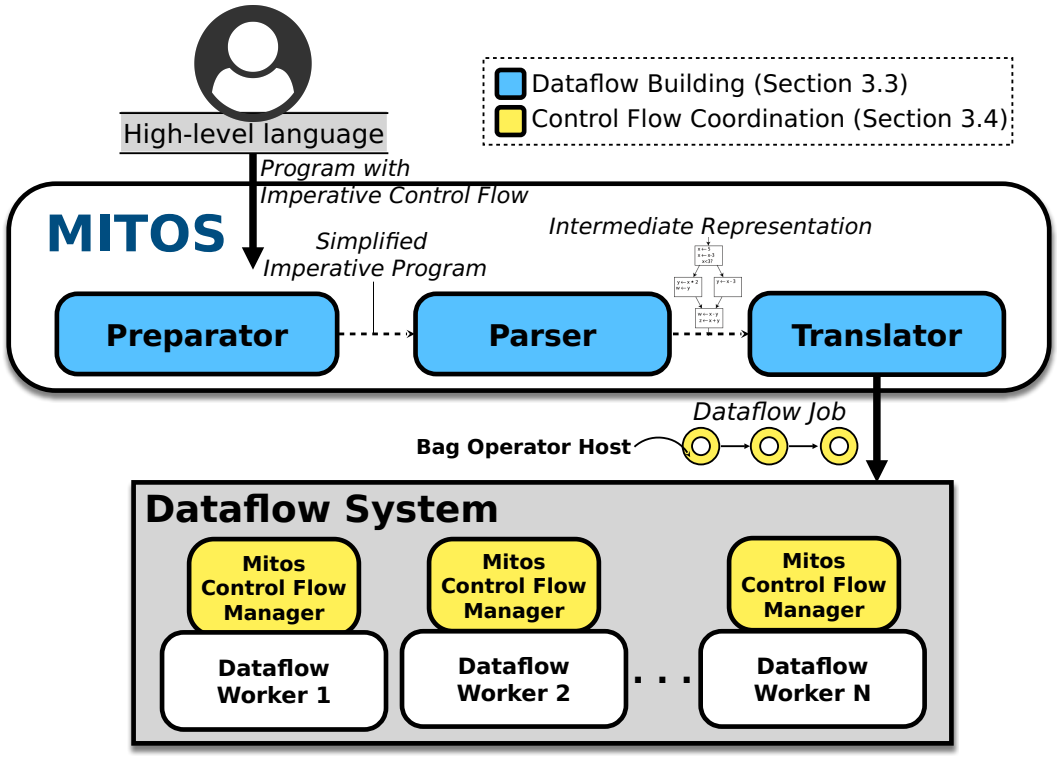


Figure 3.1: Mitos architecture.

3.3 Building Dataflows from Imperative Control Flow

Our goal is to produce a *single* dataflow job from a user's imperative program that has arbitrary imperative control flow constructs. For this, we need to inspect control flow statements and add extra edges. For example, in iterative algorithms, there is typically a dataflow node near the end of the loop body whose output has to be fed into the next iteration step. A more specific example is passing the current PageRanks from one step to the next. Additionally, we need to include non-bag variables into our dataflow jobs.

We leverage compile-time metaprogramming to overcome the above-mentioned challenges and hence create a dataflow job containing all the operations of an imperative program. Specifically, we leverage Scala macros [37] to inspect and rewrite the user program's abstract syntax tree. In more detail, we first simplify the imperative program (Section 3.3.1), and then parse it into an intermediate representation (Section 3.3.2). Both of these facilitate the translation of the user's program into a single dataflow job (Section 3.3.3).

3.3.1 Simplifying an Imperative Program

As a first step, we split those assignment statements that have more than one operation on their right-hand side. For example, we split $b = a.map(...).filter(...)$ into two assignments:

$tmp = a.map(...)$; $b = tmp.filter(...)$. For instance, Lines 8 & 9 in Figure 3.2a are the splitted version of Line 3 in Section 3.1.

Next, we take care of non-bag variables, e.g., an *Integer* loop counter or a *Double* learning rate. We wrap all these variables into one-element bags (singleton bags). This normalization step simplifies later dataflow-building by ensuring that it needs to deal with only bag operations instead of introducing special cases for non-bag variables. More specifically, we perform the following transformations: any operation that creates a non-bag value is substituted with an equivalent operation that puts the same value inside a one-element bag (e.g., creating a constant, such as $a = 1$ becomes $a = newBag(1)$); a unary function f that acts on a non-bag value is substituted with a *map* operator, whose UDF is f (e.g., $b = -a$ is substituted by $b = a.map(x => -x)$); a binary function that acts on two non-bag values is substituted by a cross product and a *map*. The cross product creates a one-element bag that contains a pair with the elements of the two input bags. The *map* operates on this pair and has f as its UDF (e.g., $c = a + b$ is substituted by $c = (a\ cross\ b).map(\_ + \_)$). Note that we can apply further simplifications in some cases. For example, $b = a + 1$ can be transformed into $b = a.map(x => x + 1)$ instead of $tmp = newBag(1)$; $b = a.cross(tmp).map((x, y) => x + y)$.

3.3.2 Intermediate Representation for General Control Flow

To handle all imperative control flow statements uniformly, Mitos transforms the program into an IR that is based on SSA [148]. As part of this transformation, Mitos introduces a different variable for each assignment statement: if a variable in the original program had more than one assignment statement, we rename the left-hand sides of all these assignments to unique names. At the same time, we update all references to these variables with the new names. However, this updating step is not directly possible if there are different control flow paths that assign different values to a variable. In this case, the different assignments in the different control flow paths are renamed to different names and hence there is no single name to change a reference into. For example:

```
1: if ... then
2:    a = ...
3: else
4:    a = ...
5: end if
6: b = a.map(...)
```

Note that after we change the left-hand sides of the assignments in Line 2 and 4 to different names, we cannot simply change the variable reference in Line 6 to just one of them at compile time. Therefore, we have to choose the value to refer to at run time, based on the actual control flow path that the program execution takes. SSA solves this problem by introducing Φ-functions, which make this run-time choice explicit (Line 6):

```
1: if ... then
2:    a1 = ...
3: else
4:    a2 = ...
5: end if
6: a3 = Φ(a1,a2)
7: b = a3.map(...)
```

We explain how Mitos tracks the control flow and thus how Φ-functions choose between their inputs at run time in Section 3.4.

By relying on SSA, we abstract away from specific control flow constructs, and thus handle all control flow uniformly: Control flow constructs are translated into basic blocks and conditional jumps at the end of basic blocks. For instance, an execution path can be specified as a sequence of basic blocks.

3.3.3 Translating an Imperative Program to a Single Dataflow Job

After simplifying an imperative program and putting it into our intermediate representation, the final step to build a dataflow job is now simple: We create a single dataflow node from each assignment statement and a single dataflow edge from each variable reference. For example, from $c = a$ *join* b, we create a *join* node, whose two input edges come from the nodes of the a and b variables. Currently, each bag operator has at most one output bag[4].

To better illustrate this final translation step, we use our Visit Count running example program (Section 3.1). Figure 3.2a shows the program's intermediate representation, with the basic blocks as dotted rectangles, and Figure 3.2b shows the corresponding Mitos dataflow. Note that the join with the page types is not included for simplicity.

Scalars. As explained in Section 3.3.1, we wrap non-bag variables in one-element bags. We show the extra code for this in *italic* in Figure 3.2a. The corresponding nodes in Figure 3.2b have thin borders.

Φ-functions. We also create the nodes with the black background from assignments whose right-hand sides are Φ-functions (Lines 4–5). Unlike other nodes, the origins of their inputs depend on the execution path that the program has taken so far: In the first iteration step, they get their values from outside the loop (Lines 1 & 2), but then from the previous iteration step (Lines 18 & 19). This choice is represented by Φ-functions of the SSA form.

Conditions. The blue node corresponds to the *ifCond* variable (Line 10), and the brown node to the loop exit condition (Line 20). These *condition nodes* determine the control flow path. Edges with corresponding colors are *conditional edges*. A condition node determines whether a conditional edge with the same color transmits data in a certain iteration step, as we explain in the following section.

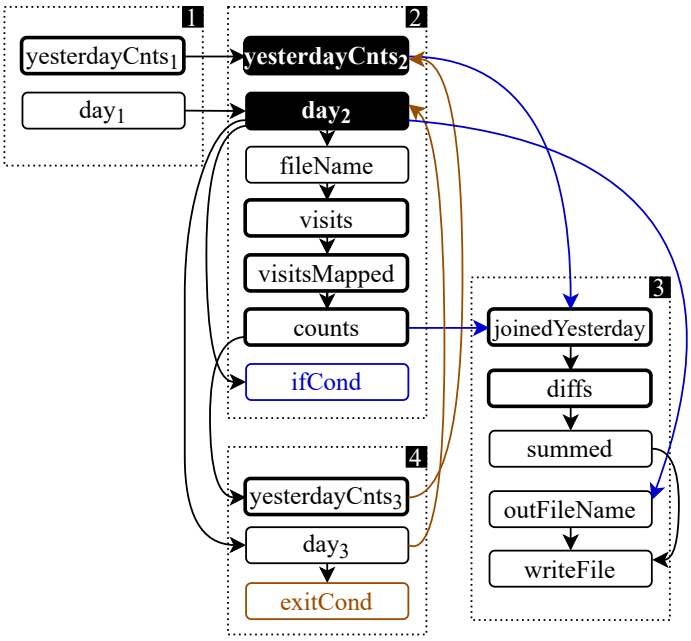
3.4 Control Flow Coordination

Once a job is submitted for execution in an underlying dataflow system, Mitos has to coordinate the distributed execution of control flow statements. It communicates control flow decisions between worker machines, gives appropriate input bags to operators for processing, and handles conditional edges. We achieve these via two components: the *control flow manager* and the *bag operator host*. The control flow manager communicates control flow decisions among machines. Thus, there is one instance per machine. Next, each operator is wrapped inside a bag operator host, which performs the coordination logic from the operators' side. This way, the coordination

[4]We could add support for multi-output bag operators by adding tuples of bags at the language level so that an assignment statement could have multiple bag variables on its left-hand side, and modifying the bag operator interface (see Section 3.4.3) to allow for emitting multiple outputs at the runtime level.

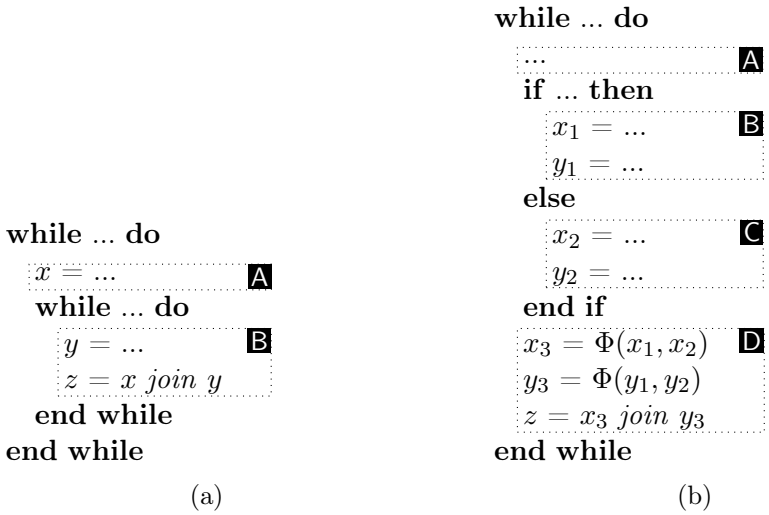
```
1:  yesterdayCnts1 = EmptyBag                                          1
2:  day1 = newBag(1)
3:  do
4:    yesterdayCnts2 = Φ(yesterdayCnts1,yesterdayCnts3)                2
5:    day2 = Φ(day1,day3)
6:    fileName = day2.map(x => "pageVisitLog" + x)
7:    visits = readFile(fileName)
8:    visitsMapped = visits.map(x => (x,1))
9:    counts = visitsMapped.reduceByKey(_ + _)
10:   ifCond = day2.map(x => x != 1)
11:   if ifCond then
12:     joinedYesterday = counts join yesterdayCnts2                   3
13:     diffs = joinedYesterday.map(...)
14:     summed = diffs.reduce(_ + _)
15:     outFileName = day2.map(x => "diff" + x)
16:     summed.writeFile(outFileName)
17:   end if
18:   yesterdayCnts3 = counts                                          4
19:   day3 = day2.map(x => x + 1)
20:   exitCond = day3.map(x => x ≤ 365)
21: while exitCond
```

(a)



(b)

Figure 3.2: (a) SSA representation of Visit Count and (b) its Mitos dataflow: The basic blocks are marked with dotted rectangles; The small rectangles are dataflow nodes, corresponding to variables in SSA; The variables corresponding to the thick-bordered nodes are bags; The colored nodes make control flow decisions and influence the same-colored edges.



Listing 3.2: Programs with non-trivial control flow structures.

logic is separated from the operator semantics. We refer to these two components together as the *Mitos runtime* (runtime, for short), and we detail them in the following.

Before diving into the runtime, we first give some required preliminaries. We will use the terms "logical" and "physical" to refer to parallelization: A dataflow system parallelizes a dataflow graph (job) by creating multiple *physical* instances of each *logical* operator. A *logical edge* between two logical operators is also multiplied into *physical edges*. Note that if an operator requires a shuffle (e.g., a grouped aggregation), then one logical edge is multiplied into $p \times p$ physical edges, i.e., one physical instance of the operator has p physical input edges corresponding to one logical input edge, where p is the degree of parallelization.

3.4.1 Challenges for the Runtime

Devising an algorithm for coordinating the distributed execution of control flow statements is challenging for three main reasons:

Challenge 1: Input elements from different bags can get mixed. Mitos aims at pipelining loop execution for efficiency reasons. This means that different iteration steps can potentially overlap. That is, different operators or different physical instances of the same operator may be processing different bags that belong to different iteration steps. An example is the Visit Count program's file reading: When any instance of the file-reading operator is done reading the file of the current iteration step, the instance can start working on the file that belongs to the next step. The difficulty is that the output from these different instances get mixed when the next operator is connected by a shuffle. This is because in case of a shuffle, each instance of the next operator receives input from all instances of the previous operator. This means that the runtime has to separate input elements that belong to different steps, so that appropriate inputs are used for computing an output bag.

Challenge 2: The matching of input bags of binary operators is not always one-to-one. In the case of binary operators (e.g., join), the runtime gives a pair of bags to an operator

at a time. To form a pair, we have to match bags arriving on one logical input edge to bags arriving on the other logical input edge. This matching is not always one-to-one, e.g., sometimes one bag has to be used several times, each time matching it with a different bag. The example program in Listing 3.2a demonstrates such a case. Input x of the join is from outside the loop, while input y is from inside the loop. This means that when the runtime provides the join with pairs of input bags, it has to use a bag from x several times, matching it with different bags from y each time. Therefore, the runtime has to save the bags coming from x.

Challenge 3: First-come-first-served does not work for choosing the input bags to process. Even when the matching of bags between the two logical input edges is one-to-one, the following naive algorithm for matching them up does not work: Assume we order bags in the same order as their first elements arrive. In this case, we could match bags from each of the inputs in the order they arrived, i.e., match the first bag from one input with the first bag from the other input, then match the second bags from both inputs, and so on. However, doing so might lead to errors. Suppose that the control flow in Listing 3.2b reaches the basic blocks in the following order: $ABDACD$. It is then possible that, due to irregular processing delays, the operator of x_3 gets data from x_1 first and then from x_2, while the operator of y_3 gets data from y_2 first and then from y_1. This can happen because the operators in the different *if* branches are not synchronized, i.e., they do not agree on a global order in which to process bags. This would clearly lead to an incorrect result: The operator of z has to match the bag that originates from x_1 with the bag that originates from y_1, and match the bag that originates from x_2 with the bag that originates from y_2. Note that this issue can arise only if we perform loop pipelining. Otherwise, all operators would finish the processing of one step before any operator starts the next step. This means that it would not happen that y_3 gets data from y_2 first and then from y_1.

3.4.2 Coordination Based on Bag Identifiers

The high-level structure of our solution to the above challenges is the following. We introduce a *bag identifier* (Section 3.4.2.1), which is straightforward to define from a hypothetical, non-parallel execution. Then, we use these identifiers as a specification of what should happen during the parallel execution. That is, we will show how to make sure that bags with the same IDs are created during the distributed execution as in a non-parallel execution, and the bags that each bag are computed from are also the same. Specifically, we will show how physical operator instances can determine during a distributed execution

- the identifier of the output bag that it should compute next (Section 3.4.2.2);
- the identifier of the input bags that it should use to compute a particular output bag (Section 3.4.2.3); and
- on which conditional output edge it should send a particular output bag (Section 3.4.2.4).

Note that the Mitos runtime is designed for allowing operators to start computing an output bag as soon as its inputs start to arrive. The runtime achieves loop pipelining via this feature, i.e., an operator can start a later step while some other operators are still working on a previous step.

3.4.2.1 Bag Identifiers with Execution Paths

A bag identifier encapsulates both the identifier of the logical operator that created the bag and the execution path of the program up to the creation of the bag. The execution path is a sequence of basic blocks that the execution reached. In a distributed execution, the execution path is determined by the condition nodes. A condition node appends a basic block[5] to the path when it evaluates its condition. Condition nodes let all other operators know about these decisions through the control flow manager. The local control flow manager broadcasts[6] the decision to all remote control flow managers through TCP connections (which are independent from dataflow edges). This way every physical instance of every operator knows how the execution path evolves. The bag identifiers are also used to separate elements that belong to different bags (Challenge 1): we tag each element with the bag identifier that it belongs to.

The execution path can grow arbitrarily large, which could conceivably introduce a bottleneck, if the coordination is implemented naively. For example, if the current execution path is repeatedly sent over the network as a list of basic blocks attached to every bag (as part of the bag ID), then we would require $\mathcal{O}(n^2)$ amount of network communication over the complete program execution, where n is the length of the complete execution path. Instead, we want to make sure that the actual implementation performs only $\mathcal{O}(1)$ amortized work for every new basic block that is appended to the execution path, and therefore requires only $\mathcal{O}(n)$ work over the complete program execution. For this,

- condition nodes should broadcast only the currently added basic block;
- when we send a bag ID over the network, it is enough to only send the length of the execution path in the ID, since every operator knows the current full list of basic blocks from the broadcasts of the condition nodes, and the execution paths in the bag IDs are always prefixes of the full list;
- the procedures described in the next sections should not repeatedly scan the entire path, but incrementally keep track of any relevant information as the path evolves.

3.4.2.2 Choosing Output Bags

By watching how the execution path evolves, operators can choose the identifiers of output bags to be computed: When the path reaches the basic block of the operator, the operator starts to compute the bag whose bag identifier contains the current path. For example, in Challenge 3, this means that the physical operator instances of both x_3 and y_3 choose to compute the output bag with path ABD in its identifier first, and then $ABDACD$.

The execution paths in the bag IDs give rise to a natural ordering between two arbitrary bags b_1 and b_2: let $b_1 < b_2$ if and only if the execution path of b_1 is a prefix of the execution path of b_2. Note that the above procedure means that a logical operator will process bags in a monotonically increasing order. However, this is only a local property of a logical operator, and is not true globally between all operators: if there does not exist a (direct or indirect) dependency between the bags computed by two operators, then their instances might process some bags in an order that is not consistent with the above ordering.

[5] Or several blocks when its target basic block has only one successor block.

[6] As we will see in Section 3.6.5, this is orders of magnitude faster than the overhead of launching a new dataflow job.

3.4.2.3 Choosing Input Bags

When an operator O_2 decides to produce a particular output bag g_2 next, it also needs to choose input bags for it (Challenges 2 & 3). This choice is made independently for each logical input.

In a non-parallel execution, the operator would use the latest bag that was written to the variable that the particular input refers to. We mirror this behavior in the distributed execution, by examining the execution path while keeping in mind the operator's and input's basic blocks. More specifically, for a logical input i of O_2, let O_1 be the operator whose output is connected to i, b_1 and b_2 be the basic blocks of O_1 and O_2, and c be the execution path in the identifier of g_2. To determine the identifier of a bag coming from i to compute an output bag g_2, we consider all the prefixes of c. Among these prefixes, we choose the longest one such that it ends with b_1. For example, in Listing 3.2a when we are computing z and choosing an input bag from x, we always choose the bag that the latest run of the outer loop computed. Concretely, if we are computing the bag with the path $ABBABBB$, then the prefix we choose is $ABBA$.

Recall that Φ-nodes need to choose between their inputs at each run. We, thus, specially treat Φ-nodes: For each particular output bag, a Φ-node reads a bag from only one input[7]. Therefore, we adapt the above procedure to choose between the inputs by looking at the above-mentioned prefixes for each input, and choosing the longer one.

It is worth noting that in some cases we need to materialize input bags. This happens in two cases: First, when an arriving input bag is not the bag that is currently being processed; Second, when the operator might need the same input bag later (for example, see Challenge 2 in Section 3.4.1). In both of these cases, the bag operator host saves the arriving input elements and provides them (possibly multiple times) to the bag operator at an appropriate time. Note that Mitos saves the elements in a serialized form to reduce the pressure on the Java garbage collector. (Spilling to disk could also be easily implemented.) It discards such saved input bags when they are not needed anymore. This happens when the execution path reaches a block b_3, such that b_1 dominates[8] b_2 from b_3. This is because in that case, the variable of O_1 will necessarily have a new value before O_2 would want to read it. Also note that a saved bag is always needed at least once. This is because conditional edges transmit data only when they know that the target operator will run (which they can determine from the information that they receive from the condition nodes).

3.4.2.4 Choosing Conditional Outputs

Operators look at how the execution path evolves after a particular output bag and send the bag on such conditional output edges whose target is reached by the path before the next output bag is computed. Specifically, let O_1 be an operator that is computing output bag g, e be a conditional output edge of O_1, O_2 be the operator that is the target of e, b_1 be the basic block of O_1, b_2 be the basic block of O_2, and c be the execution path of the identifier of g. Note that the last element of c is b_1. O_1 should examine each new basic block appended to the execution path and send g to O_2 when the path reaches b_2 for the first time after c but before it reaches b_1 again. This means that instances of O_1 can discard their partitions of g once the execution path

[7] This is different from TensorFlow, where not taken branches receive and send dummy values ("dead tensors") [186].

[8] On the control flow graph, a node d is said to *dominate* [148] a node n from node s, when all paths from s to n go through d. The *control flow graph's* [6] nodes are the basic blocks and its edges are the possible control flow transitions between the blocks.

reaches such a basic block from which every path to b_2 on the control flow graph goes through b_1. (This can also be seen by considering the non-SSA form of the program: the variable of O_1 will be overwritten before it is read by the variable of O_2.)

In the case when O_2 is a Φ-function, then we also need to consider the basic blocks of the other inputs of O_2. This is because if we consider the non-SSA form of the program, all the inputs of O_2 will be the same variable in this case, and therefore these assignment statements might overwrite the results of each other. Therefore, in case of O_2 being a Φ-function, we add the following condition to the above procedure. Before sending g to O_2, we check for every other input of O_2 (let the currently checked input operator be O_3) whether the execution path has reached the basic block of O_3 between c and b_2. If it did, then we do not send the output bag, since O_3 will send the output bag that O_2 should forward. This additional condition makes sure that the operators implementing Φ-functions do not actually need to have a logic for choosing among their inputs, because a bag is only sent to an input edge of a Φ-function if it is appropriate to emit the bag based on how the control flow proceeded during the program execution.

3.4.3 Bag Operator Host

To separate the above coordination logic from the semantics of bag operators (i.e., performing a join, aggregation, etc.), we introduced the *bag operator host*. This provides a standard, push-based interface for implementing the logic of bag operators: First, the operator's *open* method is called by the bag operator host so that the operator can initialize its state; Then, the operator is given input elements by *pushInElement* method calls; Finally, the operator is *close*d by the bag operator host, at which point it can emit its final output, e.g., all the results of a per-group aggregation. In other words, each bag operator instance is wrapped by a bag operator host, which performs the coordination logic described in the previous subsection on behalf of the bag operator: It provides the bag operator with appropriate input bags, separates input elements belonging to different input bags, and so forth. This way, it bridges the gap between the underlying dataflow system's operator interface, which does not know about bags, and Mitos' bag operator interface.

In more detail, the bag operator host provides the following guarantees to bag operators:

1. For each output bag that the bag operator should compute, the runtime first calls `OpenOutBag` on each physical instance of the bag operator. The runtime calls `OpenOutBag` only when the bag operator instance is in a *closed* state, i.e., either as the runtime's very first function call to the bag operator instance during the program execution, or as the runtime's first call to the bag operator instance after the bag operator instance called `Close` on its output collector.

2. After calling `OpenOutBag`, the runtime calls `PushInElement` any number of times. *The elements provided through these calls all belong to those input bags that should take part in the computation of the current output bag.* (Input elements for computing later output bags are buffered up by the system, and provided to the bag operator at the appropriate time.)

3. When there are no more `PushInElement` calls for a particular input bag, the runtime calls `CloseInBag` on the bag operator.

3.4.4 Fault Tolerance

Mitos comes with its own fault-tolerance mechanism as it cannot directly use Flink's Asynchronous Barrier Snapshotting algorithm [39]. This is because the communication among control flow managers happens independently of the dataflow edges that Flink knows about.

Note that Flink's asynchronous algorithm was designed for a streaming setting (i.e., continuous queries), where the latency increase caused by synchronously taking a snapshot would be unacceptable. In contrast, Mitos is targeting batch computations, and thus synchronous snapshotting would be acceptable. Therefore, a simple alternative to Flink's asynchronous algorithm would be to take each snapshot by simply synchronously pausing the entire program and saving all program state. However, this is not the algorithm that we implemented because of three issues: 1) Even in a batch setting, asynchronous snapshots can be a bit faster, since they can overlap the I/O intensive snapshot-writing with the normal program flow that might be compute-intensive; 2) We would need to make changes inside Flink's network stack because we would need to drain all network buffers before a snapshot; 3) Snapshots would include operators' internal states (not just bags), which we can mostly avoid, as we will see below.

Mitos provides an *asynchronous snapshotting* mechanism that is tied to basic blocks in the execution path. A snapshot contains the values of all the variables of a program at a certain point in the execution path, e.g. after every 10th basic block. In detail, Mitos takes snapshots as follows. First, it designates one control flow manager to be the *coordinator*. The coordinator selects the points in the execution path where snapshots should be taken and broadcasts these decisions. Each operator can then individually determine when it reaches such a snapshot point and write its latest output bag[9] into the appropriate snapshot. Once it is done, it sends a *'done'* message to the coordinator. When the coordinator receives all the *'done'* messages, it writes its state (the execution path) into the snapshot and marks the snapshot as complete. Note that this is an asynchronous algorithm, because different operators can reach a certain snapshot point at different wall-clock times. To restore from a snapshot, first the control flow managers read the execution path and tell it to the operators. Then all operators read their output bags from the snapshot and send these on their appropriate output edges. Normal execution then resumes.

3.4.5 Integration with the Underlying Dataflow System

We rely on Flink's streaming API because it allows us to add any arbitrary cycle to the dataflow graph. Note that we do not use any other streaming-specific features. As mentioned before, we aimed for minimal changes in Flink, so that Mitos is as general as possible to be able to sit on top of any dataflow system. We made only one non-trivial change in Flink to enable operators to flush output network buffers at will, which is needed at the end of output bags.

A 7-byte overhead per bag element is introduced by the multiplexing of control events (such as end-of-bag) with bag elements, and by adding some technical meta-information to each bag element (such as keeping track of Bag IDs). These could be avoided by a tighter integration with Flink, which would however require deeper changes in Flink itself, e.g., in its network stack.

[9] If the latest output bag is still the same as when the previous snapshot was taken, then we could avoid writing the bag again, and just write a reference that points to the bag that is in the previous snapshot. Currently, this optimization is not implemented.

3.4.6 External Side Effects

Mitos currently assumes that the order of side-effects does not matter. For example, the user should not read/write the same file multiple times during a program execution, nor should externally rely on output files being created or completed in a certain order.

We could serialize external side effects by automatically adding extra dataflow edges between side-effecting operators (similarly to TensorFlow[10]). More specifically, we would add output edges from a side effecting operator O_1 to all other such side-effecting operators that can be reached from O_1 without first reaching any other side-effecting operator. (In simple cases, this can be a loop edge, i.e., the edge might go from O_1 to O_1. For example, this can happen when there is only one side-effecting operator in a loop body.) O_1 would send a signal through this output edge once it has completed all side-effects for a particular run of the operator. The next side-effecting operator, which is on the receiving end of such a signal, would not perform any side effects (such as reading from a file that was written to by O_1) until it receives the signal.

3.5 Optimizations

In this section, we show how to incorporate several classic loop optimizations into Mitos' control flow handling. Note that we do not discuss the loop pipelining optimization here, as it is just a natural consequence of Mitos' control flow coordination algorithm shown in Section 3.4: An operator instance can start working on a certain output bag as soon as it is finished with the previous output bag and input elements for computing the new output bag start arriving.

3.5.1 Loop-Invariant Hoisting

We now show how to incorporate loop-invariant hoisting into our dataflows. That is, we show how to improve performance when a loop involves a loop-invariant (static) dataset, which is reused without updates during subsequent iteration steps. We can see an example of this in our running example in Section 3.1: The *pageTypes* dataset is read from a file outside the loop and is used in a join inside the loop. Another example is any iterative graph algorithm that performs a join with a static dataset containing the edges of the graph.

It is a common optimization to pull those parts of a loop body that depend on only static datasets outside of the loop, and thus execute them only once [34,60,65]. However, launching new dataflow jobs for every iteration step prevents this optimization in the case of binary operators where only one input is static. For example, if a static dataset is used as the build-side of a hash join, then the system should not rebuild the hash table at every iteration step. Mitos operators can keep such a hash table in their internal states among iteration steps. We make this possible by having a single cyclic dataflow job, where the lifetime of operators spans all the steps.

We now show how to incorporate this optimization into Mitos. Normally, the bag operators drop the state that they have built up during the computation of a specific output bag. However, to perform loop-invariant hoisting, the runtime lets the bag operators know when to keep their state that they build up for an input (e.g., the hash table of a hash join). Assume, without loss of generality, that the first input of the bag operator is the one that does not always change between output bags, and the second input changes for every output bag. Between two output bags, the runtime tells the operator whether the next bag coming from the first input changes

[10] `https://www.tensorflow.org/api_docs/python/tf/control_dependencies`

for the next output bag. If it changes, the operator should drop the state built-up for the first input. Otherwise, the operator implementation should assume that the first input is the same bag as before. For our example in Listing 3.2a, the first input bag changes at every step of the outer loop, but not between steps of the inner loop. To know whether the input bag is changing, the runtime can use the same mechanism that we described at the end of Section 3.4.2.3 for keeping track of when to drop the buffered-up inputs.

3.5.2 Incremental Loops

Many iterative algorithms perform incremental updates on a dataset, i.e., they update only a part of the dataset during each iteration step. For systems that launch separate dataflow jobs for each iteration step, even such partially updated datasets have to be scanned in their entirety at each step, to copy the unchanged elements. On the other hand, if we have a single dataflow job that executes all iteration steps (as in Mitos), we can keep incrementally changing datasets inside the state of our long-running operators (similarly to keeping the hash tables in the loop-invariant optimization). Eliminating the copying of unchanged elements makes the run time of each iteration step proportional to the number of changed elements instead of the size of the whole dataset. This can yield huge performance benefits, as shown by Ewen et al. [65] and Murray et al. [132].

Common examples are graph algorithms expressed in a Pregel-like model [117], where vertices send messages to neighboring vertices, and change their states based on received messages. In these algorithms, the fraction of vertices whose state is still changing often drops significantly as the loop progresses.

To allow Mitos to optimize incremental updates to a dataset, we add the type `MutableBag[A,K]` to our API. Here, `A` is the type of the elements, and `K` is the key type, used by joins and updates. The design of `MutableBag` was inspired by the operators of the incremental loops [65] of Flink. However, in Flink every incremental loop has exactly one mutable bag tied to it (which is called the *solution set*). By having a separate type for mutable bags, we decouple this concept from the loop API and provide more flexibility to the user (similarly to Alexandrov et al. [12]). For example, in Mitos one can use multiple mutable bags in a loop or use a mutable bag across nested loops. We provide the following primitives for manipulating `MutableBag`s:

- `ToMutable(b: Bag[A], keyF: A->K)`:
  Creates a `MutableBag[A,K]` from a regular bag. `keyF` is a UDF, which is called for every element of the bag to determine its key.

- `Join(m: MutableBag[A,K], b: Bag[B], k: B->K)`:
  Performs an inner join between `MutableBag` m and bag b, and returns a `Bag[(K,A,B)]`, which contains the matched elements with their join keys. The keys are extracted from `b` by the key function `k`. The run time of this operation is proportional to the size of `b`, since `m` is kept in a data structure that allows pointwise lookups based on the key that was specified when `m` was created. In our implementation this data structure is a hash table.

- `Update(m: MutableBag[A,K], up: Bag[A], k: B->K)`:
  For every record u in `up`, looks for a record in `m` that has the same key as u, and either replaces the found record with u, or adds u as a new record if a matching element was not found. Similarly to `Join`, the run time is proportional to the size of `updates`.

- `toBag(m: MutableBag[A,K])`:
  Converts a `MutableBag` back to a regular bag.

We translate operations on these mutable bags in the following way. For every variable m that holds a mutable bag, we create a special node $n(m)$ in the dataflow graph, which takes care of executing all the operations on m. To be able to implement the operations efficiently, $n(m)$ holds the contents of m in its state in a hash table, which it updates in-place for every call to `Update`. The edges going into $n(m)$ are determined by the operations involving m: for every variable v that appears on the right-hand side of any of these operations, we connect the node of v to $n(m)$, so that $n(m)$ receives all the input for all the operations. On the other hand, the outgoing edges of $n(m)$ are determined by the left-hand sides of all the operations involving m: for every variable t on the left-hand sides, we connect a conditional output from $n(m)$ to the node of t.

To perform the operations on m in the order in which they would appear in a sequential program, $n(m)$ can rely on the execution paths in the IDs of the input bags, and use a similar procedure as we showed earlier for choosing the next output bag to compute (Section 3.4.2.2).

In the case of mutable bags, our fault tolerance algorithm cannot avoid snapshotting the internal state of the operator that is holding the mutable bag. However, we incrementalize these snapshots by recording only the changed elements, and referring back to a previous snapshot for the other elements.

3.5.3 Speculative Execution Opportunity

Having such a general support for control flow in Mitos would make it relatively easy to implement a form of speculative execution for control flow branches. This is similar to speculative execution in CPUs: before making a particular control flow decision, the system predicts the target of the branch, and starts processing on the predicted branch target. If the prediction later turns out to be wrong, then the system throws away the speculatively computed results. The benefit of this optimization is that it enables loop pipelining in such cases where currently the wait for the loop exit condition prevents Mitos from starting the next iteration step.

We could implement this optimization in Mitos as follows. When the system adds a basic block to the execution path, it also makes a prediction about what will be the next basic block to be added, and also broadcasts this prediction. Operators treat the predicted basic block as if it were appended to the execution path, but mark the work performed for the predicted basic block as tentative. If the prediction later turns out to be wrong, then they discard the tentative work. We leave the implementation of this optimization for future work.

3.6 Evaluation

We implemented Mitos on OpenJDK 8 and Scala 2.11 and used Flink 1.6 as the underlying dataflow system. We evaluate Mitos with the following main questions in mind:

(i) How well does Mitos perform vis-a-vis state-of-the-art systems? (Section 3.6.2)

(ii) Can one efficiently bring the ease-of-use of Spark to Flink without Mitos? (Section 3.6.3)

(iii) How well does Mitos scale with respect to the input dataset size? (Section 3.6.4)

(iv) What is Mitos' iteration step overhead? (Section 3.6.5)

(v) How effective are Mitos' loop optimizations? (Section 3.6.6)

(vi) How much overhead does Mitos' fault tolerance has? (Section 3.6.7)

3.6.1 Setup

Hardware. We ran our experiments on a cluster of 26 machines, each with 2 × 8-core AMD Opteron 6128 CPUs, 32 GB memory, 4×1 TB disks, a 1 Gb network card, and Ubuntu Linux 18.04.

Tasks and Datasets. We used the Visit Count example introduced in Section 4.1, where we compare visit counts of subsequent days. We used two versions: one with and one without the join of the *pageTypes* dataset. We also used the per-day PageRank task, i.e., we inserted PageRank into the Visit Count example in place of the *reduceByKey* in Line 3. This resulted in nested loops, as explained in Section 4.1. For Visit Count, we have generated random inputs, with the visits uniformly distributed. The page types filter's selectivity is 0.5. For PageRank and Connected Components, we used a real graph[11] with 118,142,155 nodes and 1,019,903,190 edges (17.23 GB in CSV format), downloaded from the graph dataset collection of the University of Milan [27]. To be able to run many different PageRank computations, we randomly sampled its edges many times. We have also performed microbenchmarks to isolate the iteration step overhead.

Baselines. We performed most of our experiments against Spark 3.0 and Flink 1.6, with both running on OpenJDK 8. We stored input data on HDFS 2.7.1. We also performed microbenchmarks against Naiad [132] and TensorFlow [186].

System Configuration. Driver memory was set to 8GB, and worker memory was set to 24GB. In the case of Flink worker processes, this meant `-Xmx` and `-Xms` set to 20GB, plus 4GB off-heap memory was allocated, which stored network buffers. In the case of Spark, only `-Xmx` was set.

Repeatability. We report numbers for the average of three runs. We also provide the code for Mitos[12].

3.6.2 Strong Scaling

We start by evaluating how well Mitos scales with respect to the number of worker machines as well as how well it performs vis-a-vis two state-of-the-art dataflow systems: Spark and Flink.

3.6.2.1 Visit Count

Figure 3.3 shows the results for the Visit Count task. The size of the input for one day is 21 MB, and there are 365 days, i.e., the total input size is 7.6 GB. We observe that Mitos scales gracefully with the number of machines. However, Spark and Flink show a surprising *increase* in execution time as we give more machines to the system. This is because of their overhead in each iteration step increases with the number of machines, and thereby becoming a dominant factor in the execution time. We study this iteration step overhead in Section 3.6.5. In particular,

[11] `http://law.di.unimi.it/webdata/webbase-2001/`

[12] `https://github.com/ggevay/mitos`

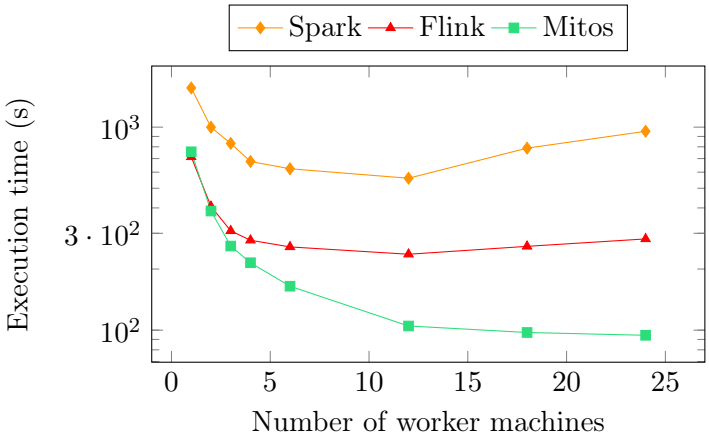


Figure 3.3: Strong scaling for Visit Count.

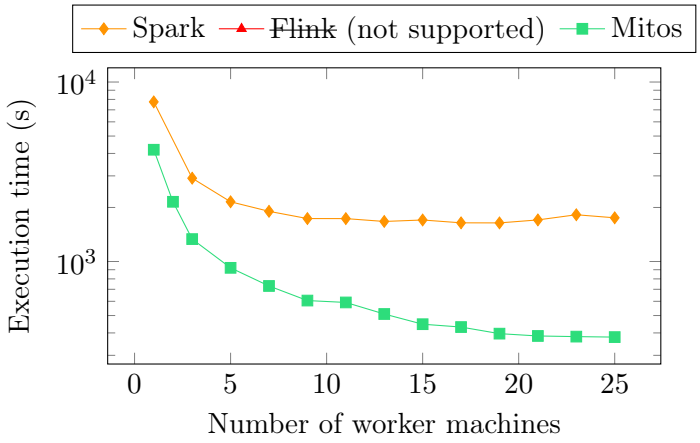


Figure 3.4: Strong scaling for per-day PageRank.

we observe that with the maximum number of machines, Mitos is 10× faster than Spark and 3× faster than Flink. The latter is an interesting result as Flink provides native control flow support. Our system improves over Flink because it performs loop pipelining.

3.6.2.2 PageRank

Figure 3.4 shows the results for PageRank. Note that Flink does not support this task with its native loop API due to the nested loops. We observe that Mitos scales gracefully, while Spark stops getting faster beyond 9 machines. Our system reaches an improvement factor of 4.6× over Spark with 25 machines.

Mitos performs and scales better than Spark and Flink. It achieves speedups of 4.6–10× compared to Spark while matching Spark's ease-of-use, and 3× compared to Flink while being easier to use than Flink.

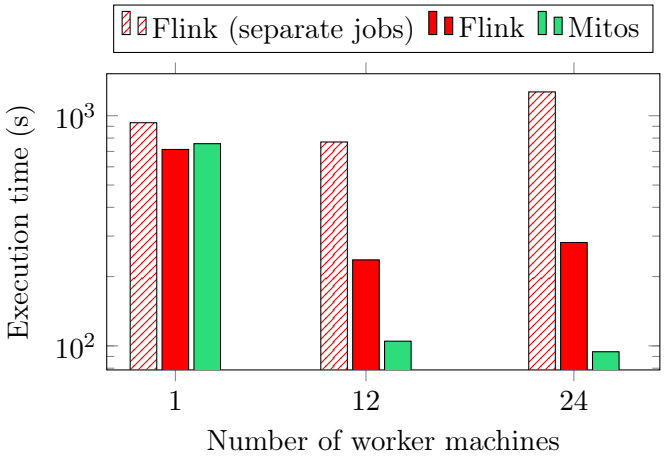


Figure 3.5: Easy-to-use Flink workaround.

3.6.3 Ease-of-Use vs. Performance in Flink

It is worth noting that implementing Visit Count using Flink's native loops was quite challenging. This is because Flink does not have built-in support for file I/O or if statements inside native loops. It took us almost 10 hours to implement such a task on Flink compared to less than 1 hour for its Spark counterpart. Thus, Flink users (including expert users) would typically resort to the workaround of an imperative loop in the driver program (similarly as in Spark), which launches a separate job per iteration step. However, this comes at the price of poor performance. We implemented Visit Count using this workaround, Flink (separate jobs), to show this problem.

Figure 3.5 shows the results. Note that, as a reference, we also show the numbers for Mitos and Flink (native loops) from Figure 3.3. We observe that launching separate Flink jobs from the driver program results in a big performance hit. For 24 machines, this approach is 4.5× slower than the Flink native loop, and 13.5× slower than Mitos. We also observe that the performance of this approach gets worse as we increase the number of machines due to its inherent job launch overhead. This result shows the high effectiveness and efficiency of our system: it allows users to write control flow imperatively, i.e., it matches the ease-of-use of this approach (as well as of Spark), while still achieving 13.5× better performance.

When users resort to an easy-to-use workaround in Flink due to the limitations of Flink's functional API, Mitos outperforms this approach by more than one order of magnitude.

3.6.4 Scalability With Respect to Input Size

Our goal is now to analyze how well Mitos performs with different input dataset sizes for Visit Count. Figure 3.6 shows the results of this experiment. We observe that our system significantly outperforms Spark, and the performance gap increases with the dataset size: it goes from 23× to more than two orders of magnitude. This is because of the loop-invariant hoisting optimization (see Section 3.6.6.1 for a detailed evaluation). Mitos outperforms also Flink, by 3.1–10.5×, while being easier to use due to its imperative control flow interface. The surprisingly large improvement factor over Flink for small data sizes is due to Flink's native loop having a large

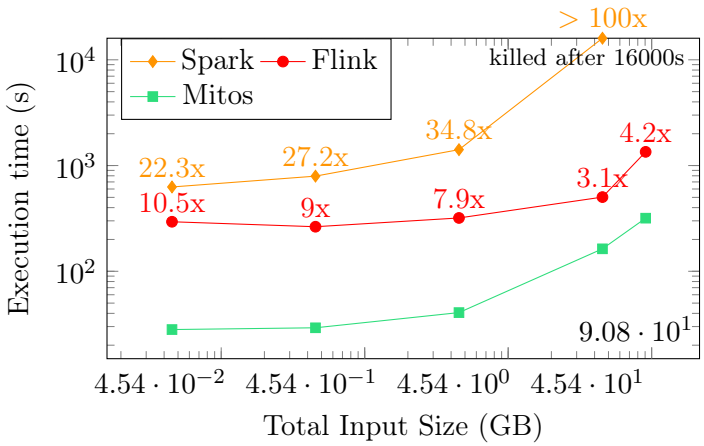


Figure 3.6: Visit Count (with the *pageTypes* dataset) when varying the input size. The factors are relative to Mitos.

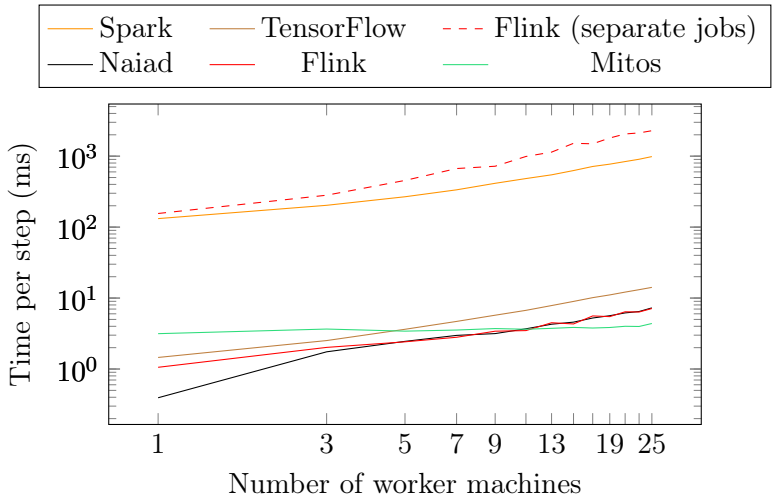


Figure 3.7: Log-log plot for the per-step overhead.

per-step overhead due to a technical issue[13].

Mitos can achieve more than two orders of magnitude speedup compared to Spark for large input datasets.

3.6.5 Iteration Step Overhead

We now dive into studying the step overhead. First, we isolate the step overhead from the actual data processing in a microbenchmark: a simple loop with minimal actual data processing in each step. More specifically, we create a small bag with enough elements to give at least one element to each partition, and then we `map` this bag at each iteration step. In TensorFlow, we manually create as many operators as there are `map` operator instances in the other systems.

[13]https://issues.apache.org/jira/browse/FLINK-3322

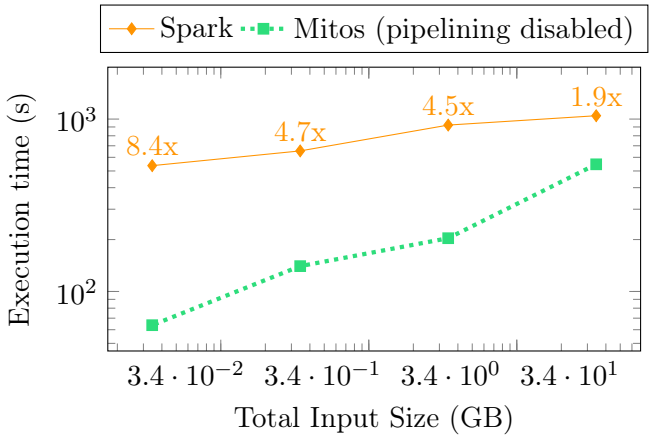


Figure 3.8: Visit Count (w/o *pageTypes*) when varying input size.

In this experiment, we also consider TensorFlow and Naiad as baselines to better evaluate the efficiency of Mitos. Figure 3.7 shows the results. We observe that the native loop of Mitos is about two orders of magnitude faster than launching new jobs for each step, i.e., Spark and Flink (separate jobs). It is interesting to note that the job launch overhead increases linearly with the number of machines. Importantly, this means that scaling out to more machines makes the step overhead problem of Spark worse. Furthermore, we can also see that Mitos matches the performance of other systems with native loops, i.e., Flink, TensorFlow, and Naiad, despite being able to handle more general control flow. Note that even systems with native control flow have some step overhead (∼1–10 ms). This is because they need to 1) broadcast control flow decisions, and 2) track progress, i.e., determine when operator input for a certain step is complete.

We now investigate the composition of Spark's step overhead. Since in typical cases each step launches a new dataflow job, we have considered so far Spark's step overhead to be the job launch overhead (task-launch overhead included). However, if a loop body does not contain an action (which is an uncommon case), then Spark can execute the entire loop in a single dataflow job (*unrolled*). One might think that this eliminates Spark's step overhead. However, the number of tasks per step is still the same. Therefore, we have to focus on the task-launch overhead (including the initiation of shuffle-reads) to know the step overhead in this case. We ran a microbenchmark that compares a loop with an action to the same loop without an action, but with the same number of tasks. In our experiments, we observed only a 10% speedup from removing the action. Therefore, we can conclude that most of Spark's step overhead actually comes from launching tasks. In other words, Mitos' performance advantage would not significantly diminish even in the case of a loop with no action.

We now examine how much effect the iteration step overhead has on a real program. As this depends on the amount of actual data processing per step, we ran an experiment where we varied the input size of the Visit Count program. In this experiment, we isolated the effect of removing the job launch overhead from Mitos' other optimizations: The join with the *pageTypes* dataset is not present in the program, and thus Mitos' loop-invariant hoisting optimization is not applicable. Furthermore, we disabled the loop pipelining optimization of Mitos. Figure 3.8 shows the result. We observe that increasing the input dataset size decreases the effect of the job launch overhead, and thereby the improvement factor of Mitos over Spark (in case when the

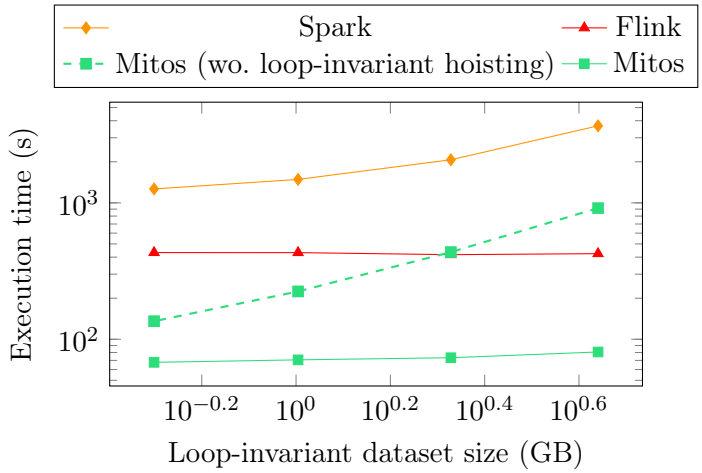


Figure 3.9: Varying the loop-invariant dataset size.

other Mitos optimizations are not applicable or turned off). For a 34 MB input, Mitos is 8.4× faster than Spark. However, even for a 34 GB input, Mitos is still 1.9× faster than Spark. In practice, many real datasets fall into this size range [145].

The overhead of Mitos is two orders of magnitude less than launching separate dataflow jobs per step, which, in real programs, can result in a 1.9–4.5× speedup over Spark, even when Mitos' other optimizations are disabled.

3.6.6 Optimizations

We proceed to evaluate Mitos' loop optimizations: loop-invariant hoisting, loop pipelining, and incremental loops.

3.6.6.1 Loop-Invariant Hoisting

We start by evaluating the loop-invariant hoisting optimization in Mitos. For this, we used the version of the Visit Count example that has the join with the *pageTypes* dataset at every iteration step. The *pageTypes* dataset does not change between steps, and therefore the loop-invariant hoisting optimization can improve performance. Figure 3.9 shows the results when varying the size of the loop-invariant dataset, while keeping the other part of the input constant (13 GB). We observe that increasing the loop-invariant dataset size has very little effect on Mitos and Flink. This is because they perform the loop-invariant hoisting optimizations i.e., they build the hash table for the join only once and then just probe the hash table at every iteration step. Still, Mitos is 5–6× faster than Flink.

On the other hand, the execution time of Spark (and the speedup of Mitos over Spark) linearly increases because Spark does not perform this loop-invariant hoisting optimization. Note that, in our Spark implementation, we manually inserted a repartitioning of the *pageTypes* dataset once before the loop. This way, the join does not need to repartition at every iteration step. However, this does not eliminate all redundancy: (1) Matching partitions might still be on different machines, and thus network transfer still happens redundantly at each step; (2) The

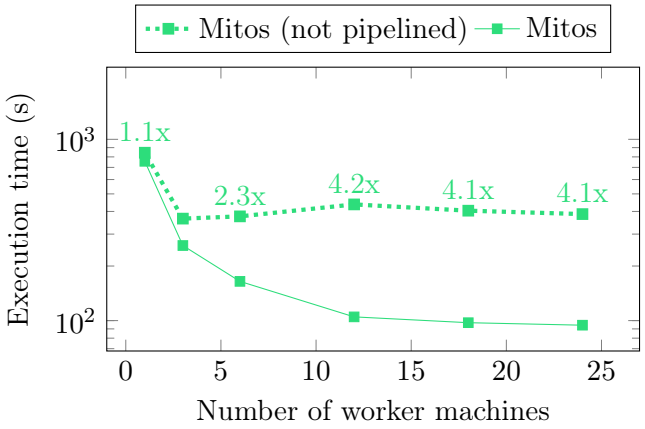


Figure 3.10: Loop pipelining with varying worker machine count.

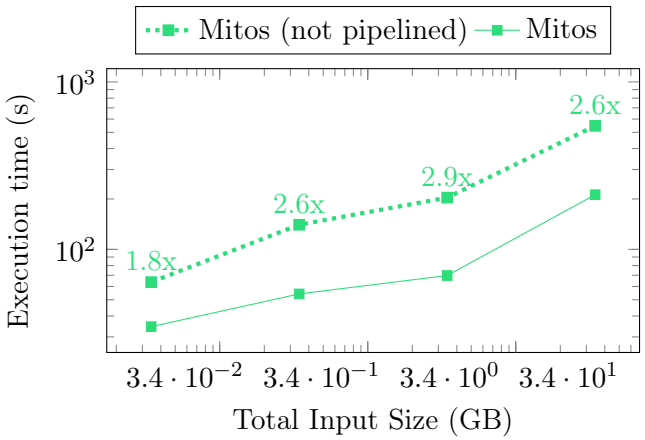


Figure 3.11: Effect of loop pipelining when varying the input size.

join's hash table building also still happens redundantly. As a result, Mitos is up to 45× faster than Spark.

To isolate the effect of loop-invariant hoisting from other differences between Spark and Mitos, we also ran Mitos with loop-invariant hoisting switched off. In this case, its execution time increases linearly with the size of the loop-invariant dataset, similarly to Spark. Therefore, Mitos is up to 11× faster than Mitos without loop-invariant hoisting.

Mitos performs loop-invariant hoisting, which improves its performance by up to 45× compared to Spark.

3.6.6.2 Loop Pipelining

We now analyze the loop pipelining feature of Mitos, which allows it to outperform Flink. Recall that, even though Flink also provides native loop support, our system is up to 3× faster in Figure 3.3, 3.1–10.5× faster in Figure 3.6, and 5–6× faster in Figure 3.9. As one might think that this performance difference could come from other factors, we ran an experiment to better isolate the effect of loop pipelining. We ran Visit Count (without the *pageTypes* dataset) in

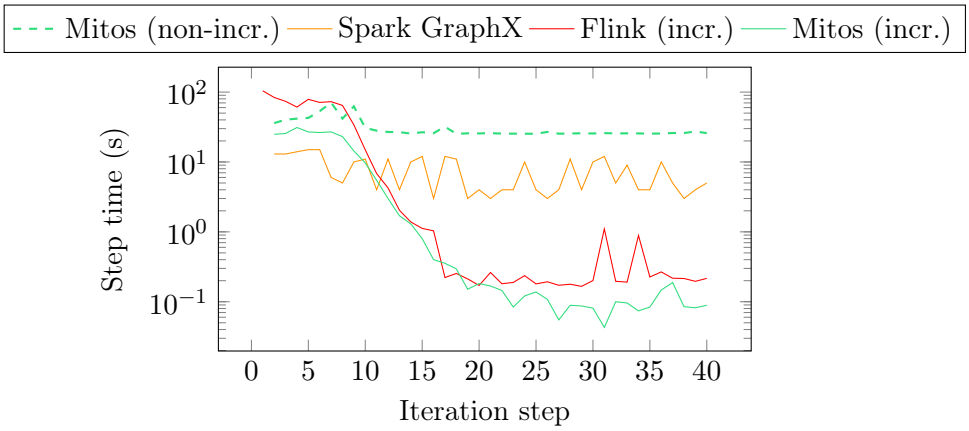


Figure 3.12: Time of the first 40 iteration steps of incremental and non-incremental versions of computing connected components of a large graph. The rest of the 744 steps are similar to the last ten shown here.

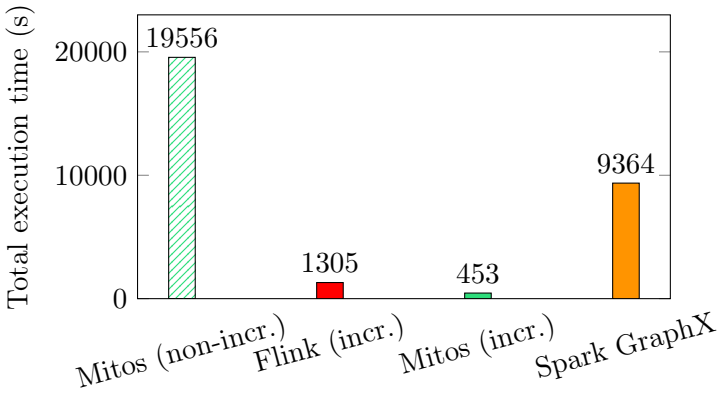


Figure 3.13: Total time of incremental and non-incremental versions of computing connected components.

Mitos with and without the loop pipelining optimization. Figure 3.10–3.11 show the results. Overall, we clearly observe the benefits of loop pipelining: Our system can be up to 4× faster with than without loop pipelining, which is made possible by our control flow coordination mechanism. Varying the input size does not have a significant effect on the speedup achievable by loop pipelining.

The control flow coordination algorithm of Mitos allows for loop pipelining, which results in speedups of up to 4×.

3.6.6.3 Incremental Loops

To assess Mitos' incremental loops, we perform experiments with the widely used label propagation algorithm [65,98] for computing the connected components of a graph. We run incremental and non-incremental implementations of the algorithm in different systems. Note that all variants have the loop-invariant hoisting optimization, i.e., we keep the build-side of the hash-join

that involves the edges of the graph. The graph that we used (see Section 3.6.1) has a large diameter, and therefore computing the connected components requires 744 iteration steps until convergence, with the majority of vertices getting their final values in the first 10 iteration steps.

In Figure 3.12 we plot the time taken for the first 40 iteration steps of several implementations. We observe that the run time of the incremental versions (including Mitos) indeed decreases significantly after the 10th iteration step. The algorithm converges after 744 iteration steps, but the rest of the steps (after the 40th) are similar to the last 10 steps of this plot. We show the total execution time in Figure 3.13.

After most of the vertex values have converged, Mitos is the fastest of the measured implementations. Mitos is faster than Flink by about a factor of three, because of the Flink issue mentioned in Section 3.6.4.

In the first 10 iteration steps the Spark GraphX [77] implementation is the fastest one, because GraphX has specific optimizations for graph computations, including a physical data representation optimized for graphs. However, after the 10th iteration step it falls behind the incremental Flink version and also Mitos, since the graph-specific optimizations cannot overcome the limitation of scheduling new dataflow jobs at every iteration step. Note that the graph-specific optimizations of GraphX could be applied together with the optimizations of Mitos, which would probably close the gap in the first 10 iteration steps.

We have encountered an additional hurdle with the GraphX implementation: calling the GraphX connected components library function using the default options resulted in long RDD lineages, which eventually caused either an out-of-memory error, or a stack overflow during the serialization of some internal data structure that holds the lineages. There is a configuration option called `checkpointInterval`, which causes GraphX to take a checkpoint at every Nth iteration step to durable storage (e.g., HDFS), and thus break the long lineage chains. The interval can be easily set to a large enough value that the I/O overhead of performing the checkpoint is negligible. However, we observed a roughly 2-times slowdown in all subsequent iteration steps after taking the first checkpoint. Unfortunately, we could not trace down the reason for this phenomenon.

The GraphX plot was obtained after extensive tuning of configuration parameters. Parallelism was set to 400. A larger parallelism resulted in more scheduling overhead and more garbage collection overhead. A smaller parallelism increased the computation time in the first 10 steps and also after the slowdown caused by the first checkpoint. Setting the checkpoint interval smaller adds to the execution time because of the HDFS I/O needed to perform the checkpoints, and setting it larger adds more garbage collection pressure because of the long RDD lineages, resulting in a slowdown at the ends of the intervals between checkpoints.

In loops having many iteration steps, but with most data points converging already in the first few steps, Mitos' incremental loops can result in more than an order of magnitude speedup.

3.6.7 Fault Tolerance

To test Mitos' snapshotting mechanism (see Section 3.4.4), we used the Visit Count program (without the *pageTypes* dataset) with an input data size of 34.4 GB. We configured Mitos to snapshot every 10th iteration step. We observed that the execution without Mitos' snapshotting is 205s, while with Mitos' snapshotting is 222s. This represents an overhead of 8.3%, which shows the high efficiency of Mitos' snapshotting algorithm.

Chapter 4

Nested Parallelism in Dataflow Systems

In this chapter, we discuss how our system, Matryoshka, flattens nested parallel programs, so that they can be executed on a standard dataflow engine.

4.1 Motivating Examples

We detail four common examples where nested parallelism is essential. We then distill three desiderata for proper nested parallelism support.

4.1.1 Bounce Rate

When analyzing website traffic data, a commonly used metric is the *bounce rate* [101,158], 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).

```
def bounceRate(visits: Bag[Visit]): Double = {
  val countsPerIP = visits.map((_, 1)).reduceByKey(_+_)
  val numBounces = countsPerIP.filter(_._2 == 1).count()
  val numTotalVisitors = visits.distinct().count()
  numBounces / numTotalVisitors
}
```

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 `bounceRate` 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 `bounceRate` 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 `bounceRate` 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 different invocations of the `bounceRate` function in parallel with each other, as well as parallelizing the individual operations that are inside the `bounceRate` function. 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.

4.1.2 Partitioned Graph Analytics

We now discuss an example that highlights how proper nested parallelism support enhances the composability of different algorithms, i.e., makes it easier for the user to build complex pipelines out of smaller building blocks. Graph partitioning (e.g., connected components) is an important building block in many graph processing algorithms [36, 103]. As an example, consider the task of computing the average distances between all pairs of nodes in each connected component of an input graph. For the first part of this task, current dataflow engines [77] typically[1] provide a connected components function similar to the following:

```
connectedComps: Graph => Bag[(VertexID, CompID)]
```

The above function takes a graph as input and produces a collection of vertices where each vertex is tagged with the identifier of the component it belongs to. Assume that for the second part of our task (i.e., computing the average distances between all node pairs in a connected component) we already have a library function:

```
avgDistances: Graph => Double
```

Thus, we would need to call `avgDistances` on each component produced by `connectedComps` to perform our task. However, `avgDistances` expects each connected component as one graph as input while the output of `connectedComps` is a single collection for all the connected components it identifies. Again, one would have to use one of the two common workarounds: outer- or inner-parallel. Besides the performance problems we identified before, these workarounds require users to modify the library functions or write cumbersome glue code between them.

We ideally want a nested collection as `connectedComps`' output:
`connectedComps: Graph => Bag[Graph]`, where `Graph` is itself represented by the collections of its vertices and edges: `(Bag[VertexID], Bag[(VertexID, VertexID)])`. This allows us to combine `connectedComps` and `avgDistances` in a natural way:

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

The above line returns a `Bag[Double]` containing the average distances in each of the connected components. Note that `avgDistances` is an iterative computation and hence existing systems, such as TraNCE [166, 167] and MRQL/DIQL [66, 67], cannot flatten it.

[1] Flink Gelly:
https://github.com/apache/flink/blob/2d10acf8189309edc42d57d603887a3431a2ae18/flink-libraries/flink-gelly/src/main/java/org/apache/flink/graph/library/ConnectedComponents.java#L45,
Spark GraphX:
https://github.com/apache/spark/blob/branch-3.0/graphx/src/main/scala/org/apache/spark/graphx/lib/ConnectedComponents.scala#L34

4.1.3 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 [17]. 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. This is shown in the following code (with a `maxBy` instead of a `map`):

```
1  // Training examples:
2  val items = readFile(...)
3  // Create a collection of parameter values to try:
4  val params: Bag[Double] = ...
5  // Try each parameter value to find the best one:
6  params.maxBy {(p: Double) =>
7    // K-fold cross-validation for hyperparameter set to p:
8    new Bag(Seq(1,K)).map((rndSeed: Int) =>
9      trainingSet = items.sample(ratio, rndSeed)
10     validationSet = items minus trainingSet
11     model = train(p, trainingSet)
12     validate(model, validationSet)
13   ).avg // Take the average of the validation runs
14 }
```

Here, `params.maxBy(f)` (Line 6) calls `f` on each element of bag `params` and returns that element of `params` for which `f` gave the largest value. This task has three nesting levels: The outermost (non-nested) level, `maxBy` introduces the second level, and the `map` (Line 8) for the cross-validation introduces the third level. Matryoshka allows dataflow engines to parallelize operations at all these three levels (and more) and thus avoids the issues of the workarounds.

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 [66,67,166] do not support loops at inner nesting levels. Therefore, they are not suitable for this kind of tasks.

Note that some iterative hyperparameter optimization algorithms determine the hyperparameter values to try next based on the results of earlier values, which hinders parallelization. Still, these algorithms often try many parameter values in one iteration step [53, 90], and thus there are still parallelization opportunities. Also, sampling-based techniques often dynamically vary the sample size [105]. Thus, it is important to efficiently handle both a large number of

small samples and a small number of large samples. A main feature of our system is exactly this flexibility, as we experimentally demonstrate in Section 4.8.2.

4.1.4 Matrices as Nested Collections

We adapt an example task from Boehm et al. [26], and show how we can express the solution in our system. We are given a matrix, where each column is treated as a separate vector, and the task is to compute the correlations between all pairs of column vectors (which is representative for more complex bivariate statistics). Suppose that we already have a library function for computing the correlation between two vectors, where a vector is (sparsely) represented as a collection of `(index, value)` pairs:

```
correlation: (Bag[(Int, Double)], Bag[(Int, Double)]) => Double
```

Suppose that a matrix is represented as a collection of `(rowInd, columnInd, value)` tuples. Using a `groupBy`, we can transform this representation into a collection of column vectors. Then we can use a cartesian product (`cross`) to get all pairs of columns, and then use a `map` to call the above `correlation` library function for each pair:

```
// Matrix originally as (rowInd, colInd, value) tuples:
val m: Bag[(Int, Int, Double)] = ...
// Transform into Bag[(colInd, Bag[(rowInd, value)])]:
val cols: Bag[(Int, Bag[(Int, Double)])] =
  m.groupBy(_.colInd)
   .map(...) // project out colInd from the inner bags
// Compute the correlations:
(cols cross cols).map {((i, coli), (j, colj)) =>
  (i, j, correlation(coli, colj))
}
```

4.1.5 Other Examples

Many other tasks in their natural specification require multiple levels of parallelism. For example, many machine learning algorithms would benefit from nested parallelism [26], e.g., ensemble learning [150] and building a multi-class classifier from a binary classifier using the one-vs-rest approach [130]. Moreover, parallel dataflow engines often require broadcast variables for accessing an originally parallel collection inside a UDF as a non-parallel collection [23]. In this case, nested parallelism is beneficial as well because then the program is scalable in the size of the collection that would have originally been broadcasted[2].

4.1.6 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 1.2). 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

[2] `https://issues.apache.org/jira/browse/SPARK-18731`

practical: Already for simple cases, such as the Bounce Rate example (Listing 4.1), it is hard to devise a flattened version (Listing 4.3). 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 Section 4.1.2.

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 three core desiderata for proper nested parallelism support. The system should allow

1. scalable operations both inside and outside their UDFs;
2. nested collections;
3. loops at inner nesting levels.

4.2 Matryoshka Overview

In the following, we introduce Matryoshka, a system that *flattens* [22, 66, 175, 176] 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.

Figure 4.1 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. As mentioned before, we use Emma [9, 11, 12] as our query language, which is an *embedded* DSL in Scala. This means that Emma is expressed in a general-purpose programming language (similarly to Spark and Flink). For example, Listing 4.1 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 [136], SQL+nested data [146], MRQL/DIQL [66, 67].

Given a nested-parallel Emma program as input (the user's program), Matryoshka removes any nesting from (i.e., flattens) the program so that it can be executed on a standard dataflow engine (without resorting to the inner-parallel or outer-parallel workarounds). We propose performing 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

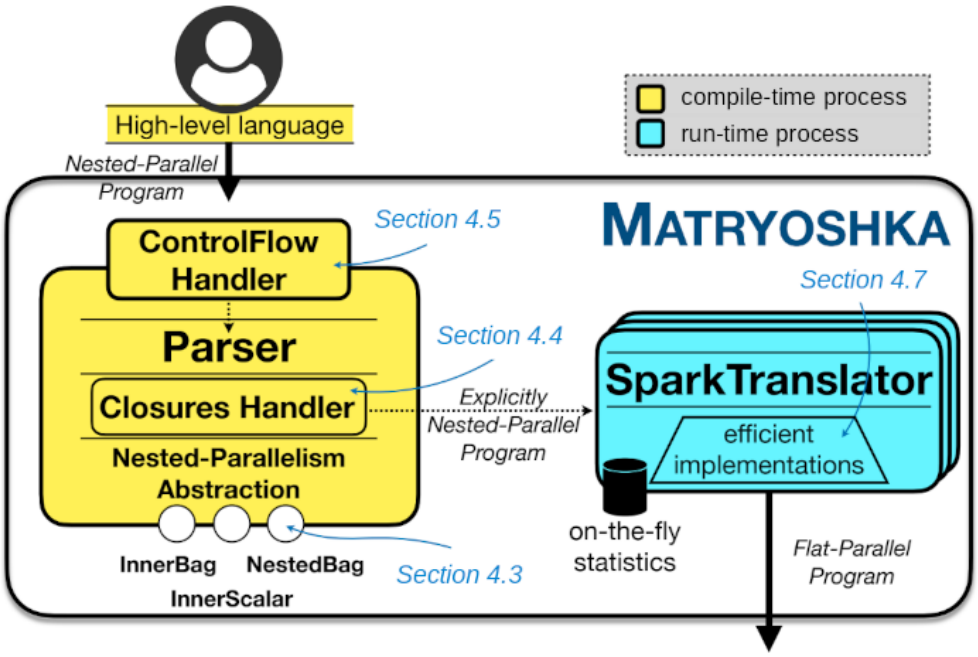


Figure 4.1: Matryoshka architecture.

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

Listing 4.1: Bounce rate program (Section 4.1.1) using nested bags and nested parallel operations. The brown parts are not supported in current dataflow engines.

and executes it. This lowering phase happens at run time 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 run time by exploiting the program structure highlighted by our nesting primitives.

Note that the above splitting of the compilation into two phases also has the advantage that we can minimize the amount of metaprogramming. Since we have an embedded DSL, flattening necessarily involves some metaprogramming, i.e., changing the user's code at the abstract syntax tree level (see Section 4.3.1.1). However, we would like to minimize the metaprogramming, because creating all the flat code with metaprogramming would be quite laborious. Therefore, our first phase, which does the metaprogramming, just performs small transformations, such as introducing the nesting primitives `InnerScalar`, `InnerBag`, and `NestedBag`, which are then resolved to flat implementations in the second phase, at run time.

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

4.3 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 run time) phases. We then present the core concept of *lifting* [20, 175], 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. Finally, we show how to handle such bag operations that appear in a UDF of an operation other than `map`.

4.3.1 Two-Phase Flattening

We perform the flattening of a nested-parallel program in two phases so that we can enable further optimizations (Section 4.6). 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*).

4.3.1.1 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*[3] operations and control flow operations into staged computations.[4] 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 run time. This distinction allows us to represent all `Bag`s with flat `Bag`s.

Let us illustrate this phase through the Bounce Rate example in Listing 4.1. The parsing phase takes as input this program and outputs the explicitly nested-parallel program in Listing 4.2 by performing the following main changes (highlighted in brown). First, it wraps scalars that are inside UDFs into `InnerScalars` (the lowering phase will need to turn these into `Bag`s). For example, in Listing 4.1, `numBounces` and `numTotalVisitors` in Lines 5–6 are scalars (integers), while in Listing 4.2 they are `InnerScalar`s. Second, it turns `Bags` inside UDFs into `InnerBags`. An example is the `group` variable in Line 3 of Listing 4.1 and 4.2. Third, it turns nested bags, i.e., `Bag[(A,Bag[B])]`, into `NestedBag[A,B]`, e.g., `visitsPerDay` in Line 2.

[3] We use the term *scalar* for any non-`Bag` type, even tuple types, such as `(A,B)`.

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

Listing 4.2: Explicitly nested-parallel bounce rate program.

```
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 4.3: Flat-parallel bounce rate program, i.e., only using flat bags and one level of parallel operations. Since there is no nesting anymore, this program can be executed on a standard parallel dataflow engine, e.g., on Spark by just exchanging `Bag`s with `RDD`s. (The key of operations such as `reduceByKey` and `join` is the first element of the pairs.)

4.3.1.2 Lowering Phase

This phase receives an explicitly nested-parallel program (Listing 4.2) and outputs a flat-parallel program (Listing 4.3). 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 flat-parallel program is both equivalent to the initial input nested-parallel program (Listing 4.1) and executable on any standard parallel dataflow engine. For example, in our system architecture (Figure 4.1), *SparkTranslator* performs this phase for Spark. There could be more translators added for other parallel dataflow engines, e.g., for Flink.

4.3.2 Lifting UDFs

Let us start by 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. (There are other operations that have UDFs besides `map`, but here we explain the lifting of only `map` UDFs. Section 4.3.6 discusses how to reduce other cases to lifting `map` UDFs.)

Lifting an operation does not only move it but also changes the operation. 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 Section 4.3.3. 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 Section 4.3.4).

For example, the operations in Listing 4.1 Line 5 have lifted versions in Listing 4.3 Line 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 Section 4.6 can be performed only at run time: 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.

4.3.3 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.

Parsing Phase – Introducing `InnerScalars`. We propose to tackle this challenge by compile-time metaprogramming [37], 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 4.2 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.

Lowering Phase – Translating `InnerScalars` to standard flat bags. 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 4.1, 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.

Translating operations on `InnerScalars` to operations on flat bags. 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. Implementing `binaryScalarOp` is a little more complex 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 4.3 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 `InnerScalars`, 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 4.2 Line 7 is Listing 4.3 Line 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` (Section 4.3.5), we then propagate the tags from the `NestedBag`; if `mapWithLiftedUDF` runs on a non-nested `Bag`, we create the tags using `zipWithUniqueId` (a standard operation in dataflow systems), 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 Section 4.6.

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.

4.3.4 InnerBag

Similarly to scalars, we must lift each `Bag` operation that is inside a UDF when lifting the UDF. 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 run time. We rely on our two-phase flattening process to overcome this difficulty.

Parsing Phase – Introducing `InnerBag`s. 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 4.1 `countsPerIP` is a different bag in each invocation of the original `map` UDF. In the parsing phase, it is replaced by an `InnerBag` (Listing 4.2 Line 4), 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.

Lowering Phase – Translating `InnerBag`s to standard 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 4.2 to Listing 4.3 `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)}*.

Translating operations on `InnerBag`s to operations on flat bags. `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 4.1 and 4.2, Line 4) 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.

In our current implementation, these rekeying operations destroy information about the partitioning of the `RDD`, i.e., even if the `RDD` representing the `InnerBag` was already partitioned by the tag before an `InnerBag.reduceByKey` call, a redundant repartitioning is later performed by Spark again when there is a later operation needing a partitioning by the tag again. A careful

implementation could avoid this. For example, `InnerBag.reduceByKey` could be implemented as follows for the case when the input `RDD` is already partitioned by tag: 1) Instead of `map`, we would use `mapPartitions`, which has an optional argument for telling the system that the UDF preserves the partitioning, and 2) Instead of calling `reduceByKey`, we would manually implement the `reduceByKey` by using `mapPartitions`, where we would only perform the reduce side, without a shuffle. A shuffle is not needed, because the rekeying before the `reduceByKey` is injective.

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 the same for all `InnerBag`s in a UDF. (See also `LiftingContext` in Section 4.6.)

4.3.5 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` that is outside a UDF. This is the case for Listing 4.1 Line 2, where a nested `Bag` comes from a `groupBy`, and is translated into a `NestedBag` in Listing 4.2 Line 2 Recall that `NestedBag`s are a typical case in nested-parallel programs (Section 4.1.1–4.1.2).

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`. Formally, we 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)}*.

We further illustrate `NestedBag` by explaining the grouping of a flat bag, which is a common case of `NestedBag`s. Specifically, let `groupByKey` be the operation that takes a bag of key-value pairs `Bag[(K,V)]` and produces a `Bag[(K,Bag[V])]`, where each of the inner bags contains all the elements that have a specific key. The parsing phase changes a `groupByKey` to a `groupByKeyIntoNestedBag`, which instead produces a `NestedBag[K,V]`. For example, grouping the flat bag *{(fruit,apple),(fruit,orange),(animal,dog),(animal,cat)}* would result into the nested bag already shown above.

4.3.6 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, for `filter` (using `flatMap` in the translation), and for `maxBy` (see Section 4.1.3).

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`.

The only operations whose UDFs we cannot lift are aggregations, e.g., `reduce`. `Reduce`'s UDF is of the type `(T, T) => T`, and it uses this UDF to repeatedly combine elements of the input bag. However, we could not think of any non-contrived example program where lifting such a UDF would be actually needed, i.e., when `reduce` would be called on a nested bag and thus the UDF would contain bag operations. Therefore, we believe that this is not a severe limitation in practice. Note that this limitation is unrelated to lifting a `reduce` itself (i.e., not its UDF) that is inside a UDF. This is a situation that we can handle, as explained in Section 4.3.4.

4.4 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.

4.4.1 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 Section 4.3.3). 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.

4.4.2 Lifted UDF Case

Consider our hyperparameter optimization in Section 4.1.3, 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 code performs 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)))}
```

4.5 Control Flow at Inner Nesting Levels

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 iteration steps, 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 (Section 4.5.1). In the lowering phase, we then lift `while` loops and `if` statements (Section 4.5.2).

4.5.1 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

```
1  val allEdges: Bag[(Date, PageID, PageID)] = readFile(...)
2  val PRsPerDay = allEdges.groupBy(_.date).map {edges =>
3    val pages = edges.flatMap((a,b) => Seq(a,b)).distinct
4    var PR = pages.map(x => (x, initWeight))
5    do {
6      val newPR = ... // Compute one PageRank step (use PR from prev. step)
7      val delta: Double = PR.join(newPR)
8        .map{(k,(oldval, newval)) => abs(oldval-newval)}.sum
9      PR = newPR
10   } while (delta > epsilon)
11   return PR
12 }
```

Listing 4.4: PageRank – imperative control flow.

the function calls to the lifted versions. We, thus, encapsulate the lifted versions inside functions[5], 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.

For example, Listing 4.4 shows an iterative program for computing PageRanks [140] per day, which has control flow statements inside a UDF. This program is similar to our Bounce Rate example in Section 4.1.1: it reads lines from a log file, groups these lines by date, and computes a PageRank for each group. Listing 4.5 shows this program after we change the imperative while loop to a higher-order function call (Lines 5–11). Then, the change to `InnerBag`s, `InnerScalar`s, and `NestedBag`s happens, and then the lowering phase finalizes the lifting to generate the flat-parallel program.

4.5.2 Lifting Loops

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

For example, consider the code in Listing 4.5. Before the lifting, the UDF of the `map` in Line 2 is executed once for each element of the grouped bag (i.e., each group), i.e., there is a separate loop for each group. However, recall that a lifted UDF is executed just once. This

[5]Alternatively, we could directly complete the lifting of the control flow statements during the parsing phase. However, we want to minimize the amount of code generated by compile-time metaprogramming, because it is much more tedious than writing code the normal way.

```
1  val allEdges: Bag[(Date, PageID, PageID)] = readFile(...)
2  val PRsPerDay = allEdges.groupBy(_.date).map {edges =>
3    val pages = edges.flatMap((a,b) => Seq(a,b)).distinct
4    val initialPR = pages.map(x => (x, initWeight))
5    val finalPR = doWhile(initialPR, PR => {
6      val newPR = // Use PR to compute one PageRank step
7      val delta: Double = PR.join(newPR)
8        .map{(k,(oldval, newval)) => abs(oldval-newval)}.sum
9      val cond: Boolean = delta > epsilon
10     return (newPR, cond)
11   })}
12   return finalPR
13 }
```

Listing 4.5: PageRank – loop as a higher-order function.

means that when the lifted UDF calls the lifted version of `doWhile`, the function should execute all the iteration steps of all the loops.

We tackle this challenge by relying on the abstractions for lifted operations introduced in Section 4.3. 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 iteration steps into `InnerBag`s and/or `InnerScalar`s. For example, in Listing 4.5, the variables `initialPR`, `PR`, and `NewPR` are turned into `InnerBag`s by the parsing phase.

Still, we must manage data that enters or leaves the body at each iteration step and lift the loop control logic. Specifically, we need to:

(P1) discard those parts of `InnerBag`s and `InnerScalar`s from the iteration steps whose original loops have finished;

(P2) save the result of the discarded parts as soon as 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. For instance, Line 9 in Listing 4.5 will be turned into such an `InnerScalar`. We leverage this bag to achieve the above P1-P3 as follows (see Listing 4.6):

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);

```
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 4.6: Lifted while loop. For ease of exposition, this code passes only one bag between iteration steps, i.e., `bodyFunc` is an `InnerBag[T,A]=>(InnerBag[T,A],InnerScalar[T,Bool])` function, such as in Listing 4.5

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 (Line 7–8);

Impl. of (P3) If (P1) did not let through any element, then we exit the lifted loop. (Line 9)

4.5.3 Lifting If Statements

As mentioned before, the higher-order function version of an if statement is a function that takes the condition as a boolean value, and takes two branch functions. As part of transforming the code to this higher-order function representation, variables that are referenced from inside the if statement but defined before the if statement are made explicit by adding them as parameters to the branch functions. Similarly, variables that are assigned inside one of the branches and then used later outside the if statement are made explicit by adding them as return values to the branch functions.

A lifted if statement needs to do the work of many executions of the original if statement that the original code would have executed in different UDF invocations. The challenge is that some of the executions of the original if statement would have executed the *then* branch, while some the *else* branch. Therefore, a lifted if statement executes both branches, but lets into each of the branch functions only the values for those tags for which the if condition had the appropriate value. (This is a somewhat similar idea to predication [28].)

In more detail, a lifted if statement takes an `InnerScalar[T,Boolean]`, which specifies for which of the tags does the if condition hold. Recall how loop exit conditions are joined on tags with the loop variables to allow for filtering out tags for which the loop continues. A lifted if statement can be implemented in a similar way: We join the `InnerScalar` of the condition on the tags with the `InnerBag`s and `InnerScalar`s that are passed into the branches, and then filter out those values where the condition has the appropriate value for the branch. This ensures that when we then execute both branch functions, each of them only gets data belonging to those tags where the condition had the appropriate value. Then we union the results of the branch functions to get the final result.

4.5.4 Implementation

Here, we discuss some performance considerations when implementing the unioning of all the partial result RDDs (called `finished` in Listing 4.6) in the lifted while loop. For ease of exposition, Listing 4.6 Line 8 simply performs a binary (two-input) union at every iteration step: it unions the RDD that stores the results of all earlier iteration steps (`result`) with the RDD that stores the results coming from the current iteration step (`finished`). However, our actual implementation is different, as explained in the next paragraphs.

First, we should not actually perform a binary union at every iteration step, since that approach has an $O(SN)$ worst-case total run time across all iteration steps, where S is the final result size, and N is the number of iteration steps. This is because a binary union scans both input RDDs, and therefore result elements from early iteration steps would get scanned again and again in every remaining iteration step. Fortunately, Spark offers an n-ary union operator. Therefore, instead of a cascade of binary unions, we just save all the `finished` RDDs in a list, and apply an n-ary union after the loop is finished.

Second, we need to checkpoint (and manually force) the `finished` RDD at every iteration step. This is needed to avoid referring back to RDDs from older iteration steps after the loop is finished. Referring to the old RDDs would either trigger recomputations or considerably increase the memory footprint of the program.

4.6 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 run time, based on different data characteristics. Most of the optimizations depend on the sizes of the bags representing the InnerScalars. Fortunately, the structure of the program, which is visible at the logical plan level, gives vital information about the sizes of InnerScalars to the optimizer. In the remainder of this section, we first discuss how we track the sizes of InnerScalars and then discuss optimizations based on these sizes.

InnerScalar Sizes. We exploit an important observation to track the sizes of InnerScalars: *All InnerScalars inside a lifted UDF have the same size.* Recall that the bags representing InnerScalars 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 InnerScalars 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.

Computing and Tracking InnerScalar Sizes. Each lifted UDF has an associated metadata object, which we call `LiftingContext`, which stores the `InnerScalar` size of that UDF. 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`. 1) If it is a flat bag, then the size will be simply the size of this input bag; 2) If it is a `NestedBag`, then the size is determined at the point where the `NestedBag` is created: if it was created inside a lifted UDF, then we simply take the size information from that UDF's

`LiftingContext`; if the `NestedBag` is created by a grouping, then we compute the number of distinct keys.

4.6.1 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 run time. 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, because we know their sizes upfront, as explained above.

4.6.2 Joins between InnerBags and InnerScalars

Recall from Section 4.4 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 (Line 5 in Listing 4.6).

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 always choose a broadcast join when there are not enough elements in the `InnerScalar` to give work to all CPU cores. Above that, our current threshold for switching to a repartitiong join is 10000 elements in the `InnerScalar`.

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 [14]: 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 [14]: Instead of directly making the join algorithm choice, we could give the above information as hints to the engine optimizer. Currently we do not perform cost-based optimization, which is out of scope for this thesis, but integration with a mature system optimizer could enable that.

4.6.3 Half-lifted MapWithClosure

Recall from Section 4.4, that there exist half-lifted operations in lifted UDFs, where only one of the inputs comes from inside the UDF, 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, then we broadcast it. This is quick to check, and it is also the common case due to the optimization in Section 4.6.1. Otherwise, we use Spark's `SizeEstimator` to compare the sizes of the two inputs and broadcast the smaller one.

4.7 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 arise mainly when employing different variations of the inner-parallel workaround, which is not needed when using Matryoshka. As mentioned in Section 4.3.6, we also assume that the UDFs of bag *aggregations*, such as `reduce`, do not contain bag operations, which is an uncommon case in practice. Also, we do not support recursive functions. This is because we handle function calls by simply inlining them (through Emma's compiler infrastructure).

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 and without recursive functions.*

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 (Section 4.3.6), eliminates closures (Section 4.4), and transforms control flow statements into a functional representation (Section 4.5.1). Then, the parsing phase traverses the code statement-by-statement (compound statements are broken down into atomic statements, similarly as in Section 3.3.1), and makes local changes to 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`[6];

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`;

[6]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.

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 Sections 4.3.3 and 4.3.4. It also turns control flow operations into their lifted equivalents (Section 4.5.2). 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 briefly discuss the case of handling more than two levels of parallelism. In this case, 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. 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[7] 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[8] 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. □

[7]The word "map" is used here in the mathematical sense, as opposed to the bag operation `map` in other parts of the thesis.

[8]By *output operation* we mean writing a bag to a distributed filesystem, such as HDFS [164].

4.8 Evaluation

We implemented the experiments for Matryoshka on Spark 3.0 with OpenJDK 14 and Scala 2.12. As a distributed filesystem, we used HDFS 2.7.1.

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 also compare to other systems supporting nested parallelism, namely SystemDS [24, 26] and DIQL [67]. We designed the 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?

(v) compare to DIQL [67] and SystemDS' parfor construct [26]?

4.8.1 Setup

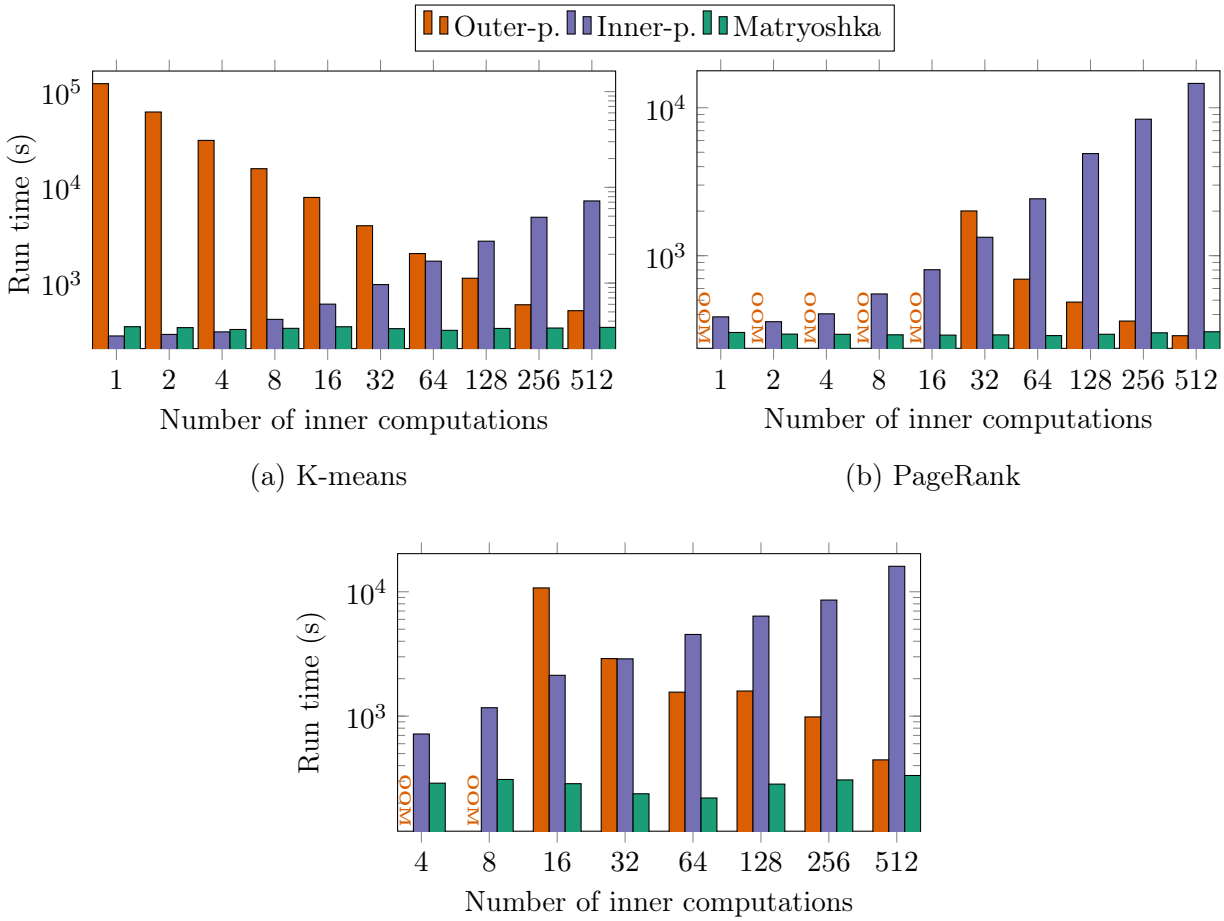
Hardware. We ran most of our experiments on a cluster of 25 machines: each with two 8-core AMD Opteron 6128 processors, 32 GB main memory, 4×1 TB hard disks, and 1 Gb network, and 64-bit Ubuntu Linux 18.04. We also experimented with a large cluster, see Section 4.8.8.

System Configuration. We configured Spark and HDFS with one node as the main/name node and the others as workers/data nodes. We dedicated 22 GB memory per machine to Spark processes and set the Spark default parallelism to 3× the total number of cores.[9]

Tasks and Datasets. We considered data analytics tasks from different areas: namely *PageRank* (graph analytics), *Average Distances* (graph analytics), *K-means* (machine learning), *Bounce Rate* (web analytics), and *Pairwise Correlations* (statistics). While Bounce Rate, Average Distances, and Pairwise Correlations are explained in Sections 4.1.1, 4.1.2, and 4.1.4, 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 [82] and BlockRank [97]. Listings 4.1 and 4.4 show the code for Bounce Rate and PageRank, respectively. Note that K-means, PageRank, and Bounce Rate have two levels of parallelism, while Average Distances has three levels. Note that Bounce Rate and Pairwise Correlations have no control flow statements while the other three do. We generated datasets for each task, with sizes varying between 2 GB – 384 GB. Grouping keys are generated from a uniform distribution, except in the data skew experiment where we used a Zipf distribution.

Baselines. We considered the inner- and outer-parallel workarounds as well as DIQL [67] and SystemDS [26] as baselines.

[9] The Spark documentation suggests setting the parallelism to 2-3× the total number of cores: `https://spark.apache.org/docs/3.0.0/tuning.html#level-of-parallelism`



(a) K-means

(b) PageRank

(c) Average Distances

Figure 4.2: Scalability in the number and sizes of inner computations.

Repeatability. All the numbers we report are the median of three runs. We provide the code of all our experiment programs[10].

4.8.2 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 run time to stay nearly constant.

Figure 4.2 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. Matryoshka is up to two orders of

[10] `https://github.com/ggevay/matryoshka`

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. Surprisingly, our system is a bit faster than inner-parallel even for a small number of inner computations in case of PageRank and Average Distances. This is due to the inner-parallel workaround performing an extra `groupBy` when the nested bag is produced by a `groupBy`, which is the case for PageRank and AverageDistances but not for K-means. Our technique avoids this `groupBy`, as we work on the flat representation. 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.

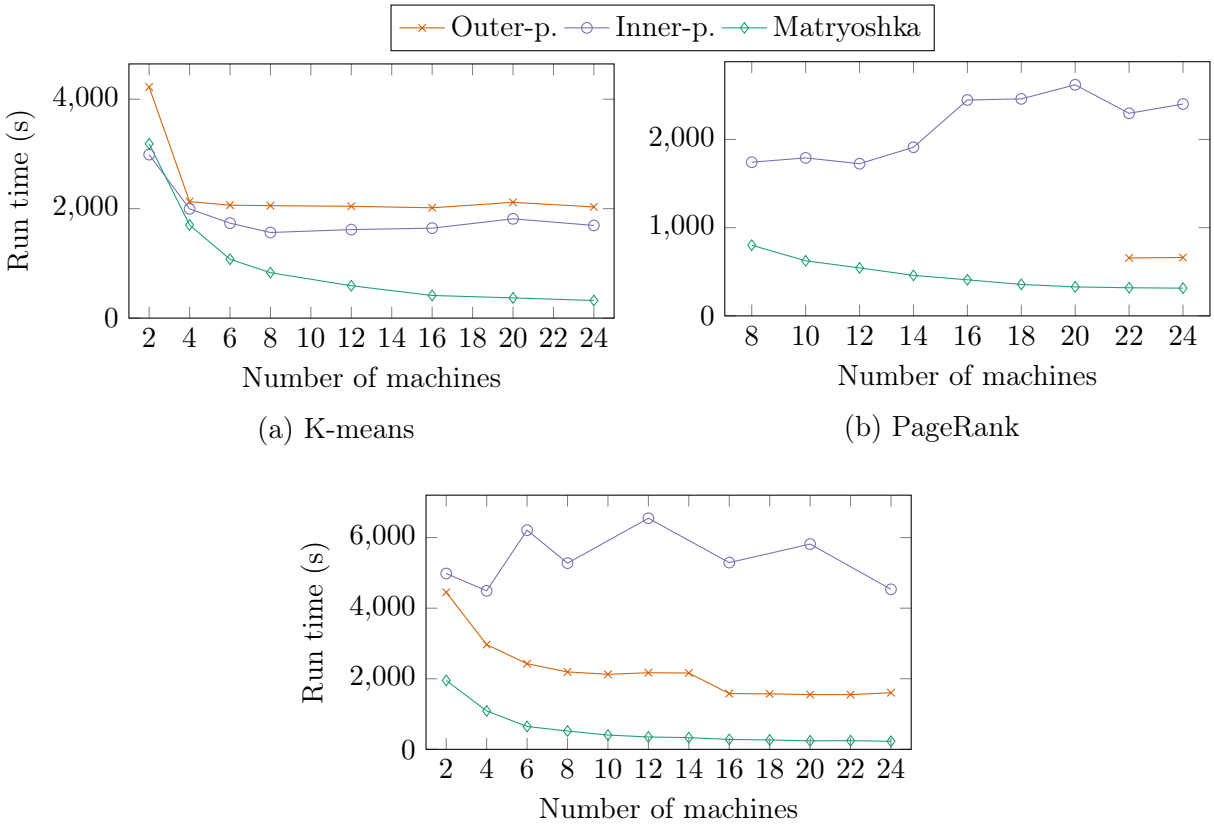
Matryoshka scales in both inner and outer collection sizes, reaching orders of magnitude better performance than baselines.

4.8.3 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 4.3 shows the results of this experiment. Overall, we observe that our system scales gracefully, while the workarounds do not: when adding more machines, Matryoshka consistently gets faster, while the two workarounds' run times remain constant in many cases, or sometimes even worsen. With the maximum number of machines, our system is 5–20× faster than inner-parallel, and 2–7× faster than outer-parallel. The performance advantage of our system comes from the same reasons as in the previous experiment: namely 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 (see also Figure 3.7): more partitions mean more (i) scheduling and (ii) task-launch overheads [138]. Matryoshka does not suffer from any of these problems.

When increasing the number of machines, Matryoshka scales better than the inner- and outer-parallel workarounds.



(a) K-means

(b) PageRank

(c) Average Distances

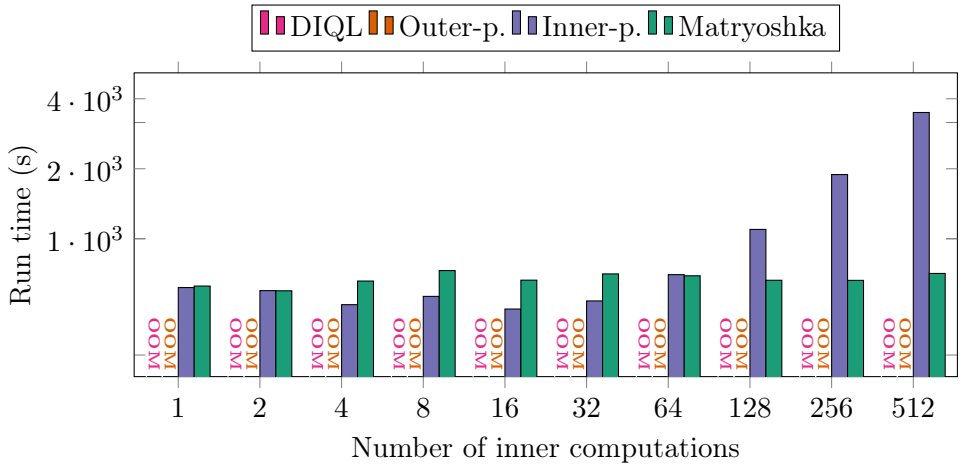
Figure 4.3: Scalability in the number of machines.

4.8.4 Performance Without Control Flow – Comparison with DIQL

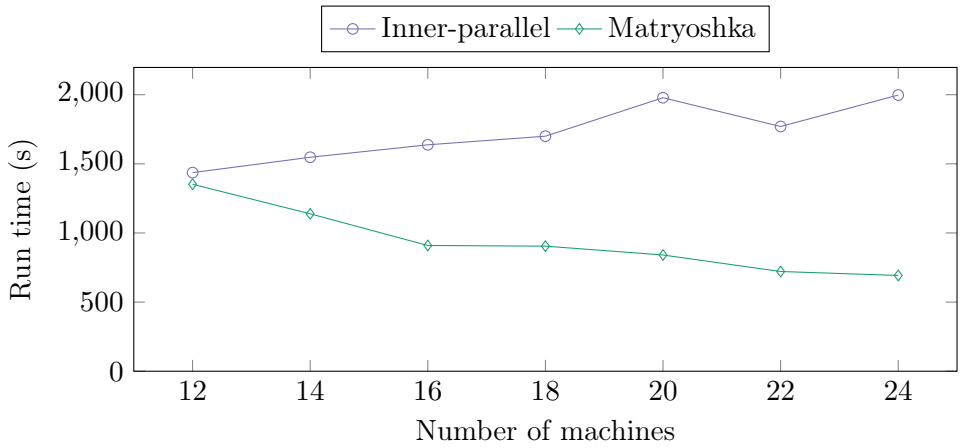
We now evaluate the performance of Matryoshka when a task has no control flow statements. We repeated the experiments from Section 4.8.2 and Section 4.8.3 but using the Bounce Rate task, which has no control flow statements. In addition to the inner-parallel and outer-parallel workarounds, we also considered DIQL as a baseline. (DIQL does not support control flow statements in the inner levels, and therefore we did not use it for the previous experiments.) We used 256 inner computations for the scale-out experiment.

Figure 4.4 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. In contrast, inner-parallel processes it in smaller chunks per job thereby avoiding spilling.

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 4.5 shows the results. We observe



(a) Weak scaling.



(b) Scaling out. Outer-parallel ran out of memory in all cases here.

Figure 4.4: Bounce Rate (no control flow). Input size is 48 GB.

that Matryoshka is faster than DIQL in all cases, by up to 6.6×.

Matryoshka is faster than baselines also when programs do not involve control flow statements.

4.8.5 Comparison with SystemDS' Parallel For Loop

We now compare Matryoshka with SystemDS' parallel loop construct. Boehm et al. [26] introduced a *parallel for loop* (or *parfor*) in Apache SystemML/SystemDS [24,25,74], which allows for adding outer levels of task parallelism on top of the data parallelism of linear algebra operators. SystemDS uses a sophisticated optimizer to choose between different execution strategies for each level: sequential execution; parallel on multiple cores of one machine (termed *local parallelism*); or in a MapReduce/Spark[11] job (termed *remote parallelism*). These execution strategies have

[11]Originally, Boehm et al. [26] described *parfor* execution on MapReduce, but SystemDS (which was formerly SystemML) currently uses Spark.

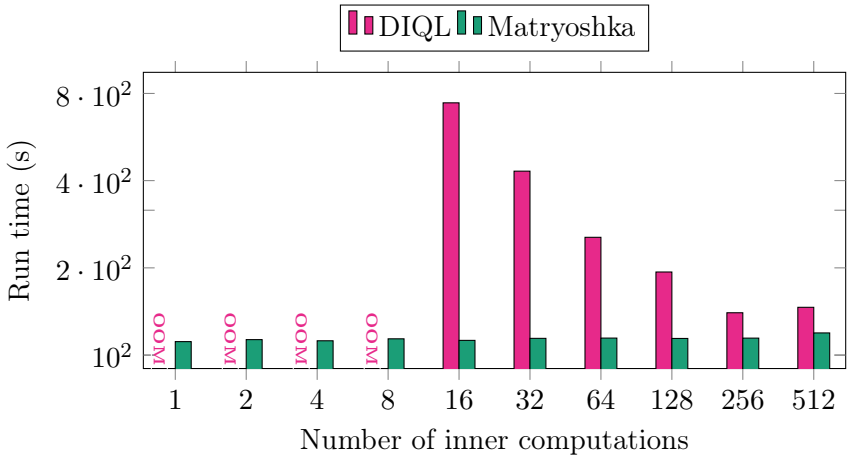


Figure 4.5: Performance against DIQL for Bounce Rate. Input size is 12 GB.

further details to be chosen by the optimizer, such as how many parallel worker threads to use, how to partition the input data of the parallel loop and how to merge the results.

Still, as SystemDS does not employ flattening, it cannot parallelize multiple levels using the same remote-parallel Spark job. Therefore, we expect Matryoshka to have an edge over SystemDS in certain cases: First, SystemDS might run into the usual problems of the outer- or inner-parallel workarounds (such as the outer level not providing enough parallelism, or launching too many Spark jobs for the inner executions). Second, deciding between the many possible execution strategies for a complex program seems to be a very hard problem, and therefore the optimizer might make wrong choices sometimes.

For this experiment, we used the pairwise correlation example of Boehm et al. [26], which computes correlations between all pairs of columns of a matrix. In SystemDS, this task involves three levels to be possibly parallelized: two nested *parfor* loops plus the vector operations at the innermost level. How to express this task in our system is described in Section 4.1.4. In our case, the nested *parfor* loops are substituted with a cartesian product, and therefore we have only two levels to be parallelized. Matryoshka flattens these two levels, and therefore it always parallelizes at all levels, using Spark.

We generate random matrices, and we vary the number of columns in powers of 2. For each 2-fold increase in the number of columns, the total amount of work to be performed should increase 4-fold. We ran the experiment with two kinds of matrices (they are ∼3.6 GB with 128 columns):

1. dense with 10^6 rows (Figure 4.6)

2. a sparsity of 0.01 with 10^8 rows (Figure 4.7).

SystemDS uses an optimized matrix representation, which makes matrix/vector operations much faster than Matryoshka in the dense case. In the sparse case, this advantage is greatly reduced. SystemsDS's Spark configuration (default parallelism, memory, version, etc.) is the same as Matryoshka's, except that we used JDK 11.0.2 instead of JDK 14 to work around an obscure classloader issue.

We ran SystemDS in four configurations:

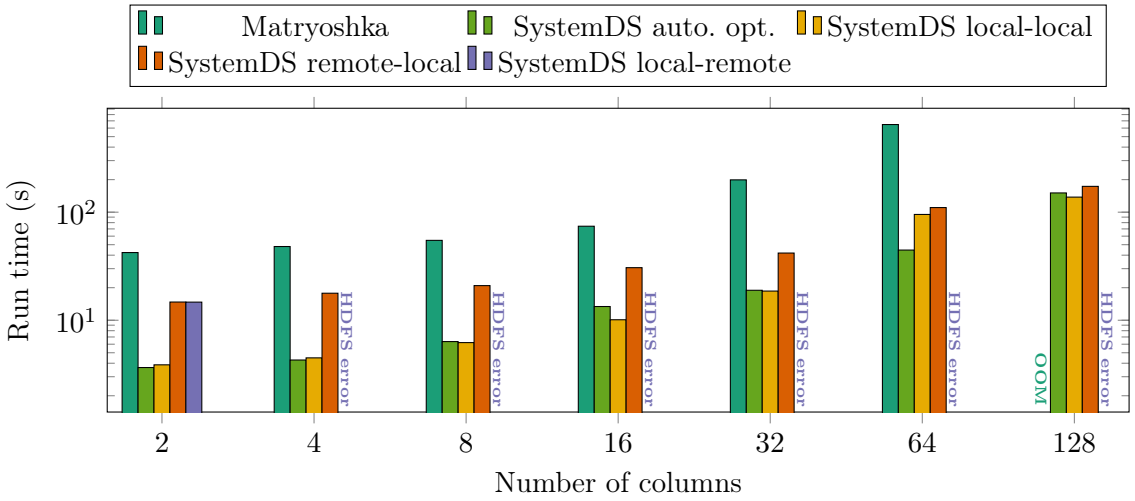


Figure 4.6: Pairwise Correlations task – Comparison with SystemDS – dense input matrix

- *optimized automatically*: We let the optimizer freely choose execution strategies for all levels.
- *local-local*: We constrain the optimizer to using local parallelism for both the outer *parfor* and the inner *parfor*. This execution strategy uses only a single machine.
- *remote-local*: We constrain the optimizer to using a Spark job for the outer *parfor*, and local parallelism for the inner *parfor*.
- *local-remote*: We constrain the optimizer to using local parallelism for the outer *parfor*, and Spark jobs for the inner *parfor*.

Figure 4.6 shows the results for the dense matrix case. SystemDS' optimized matrix representation makes it faster than Matryoshka in all cases. Matryoshka runs out of memory with 128 columns, while SystemDS is still able to finish. (SystemDS in the local-remote configuration fails with a HDFS error while creating a temporary file. This might be due to a bug in SystemDS or HDFS, or some issue in our HDFS infrastructure.) Notably, the optimizer always chooses a local-local execution strategy, which is indeed the best choice in all cases. (The difference between local-local and auto. opt. with 64 columns is only due to a large variance with that configuration in the run times of different runs.)

Figure 4.7 shows the results for the sparse matrix case, which shows a more nuanced picture, because here SystemDS' matrix representation has less advantage. Even though with a few matrix columns SystemDS' local-local strategy is still faster than Matryoshka, with more columns Matryoshka becomes faster. This is because having more columns allows for more parallelization, which Matryoshka can make better use of than SystemDS' local-local strategy.

It is interesting to examine whether the optimizer can choose the best execution strategy also in the sparse matrix case. The optimizer always chooses the local-local execution strategy, similarly as in the dense matrix case. However, in the sparse matrix case, the remote-local and local-remote strategies are better than the local-local strategy with many columns (by more than an order of magnitude with 128 columns), and are able to keep up with or outperform Matryoshka. However, the optimizer always chooses local-local, i.e., it is not able to find the best

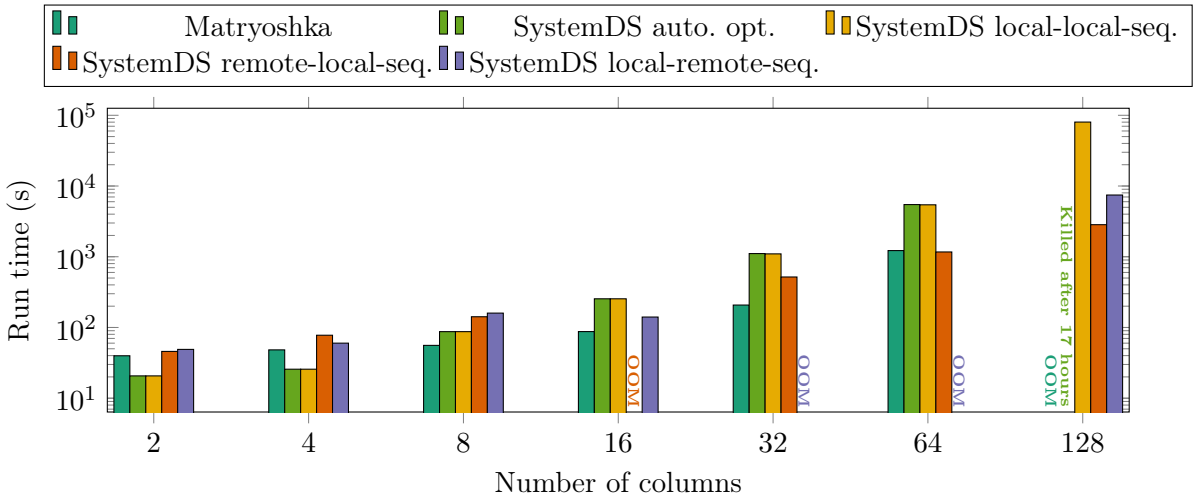


Figure 4.7: Pairwise Correlations task – Comparison with SystemDS – sparse input matrix. We killed auto. opt. after 17 hours, but the execution plan was the same as local-local, and therefore in the results analysis we assume a similar run time.

choice among SystemDS' different execution strategies. Matryoshka does not have an optimizer, but instead flattens the nested parallelism, and thus parallelizes all levels using the same Spark job.

Matryoshka runs out of memory with 128 columns, but neither of SystemDS' four execution strategies have this problem. This is because SystemDS' optimizer can make a smart choice on how many parallel instances (parameter k) of the *parfor* loop body can be running in parallel to fit in memory. This parameter is essentially infinite in case of Matryoshka: all instances of the inner computation run in the same Spark job. However, we could incorporate a similar optimization as SystemDS: We could have an execution strategy that is a hybrid between the flattened and the inner parallel workarounds: The inner parallel workaround runs one instance of the inner computation per one Spark job, but we could instead run k instances in the same Spark job, running the lifted version of the inner computation n/k times, where n is the number of inner computations. This way, we would avoid running out of memory in some cases, similarly to SystemDS. Also, this would avoid disk spilling in cases such as the one mentioned in Section 4.8.4 (4–32 inner computations in Figure 4.4a).

SystemDS' specialized matrix representation provides a big advantage over Matryoshka in case of dense matrices. However, with sparse matrices, Matryoshka's flattened execution strategy can make use of more parallelization opportunities than the execution strategy that SystemDS' optimizer chooses.

4.8.6 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 4.8 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-

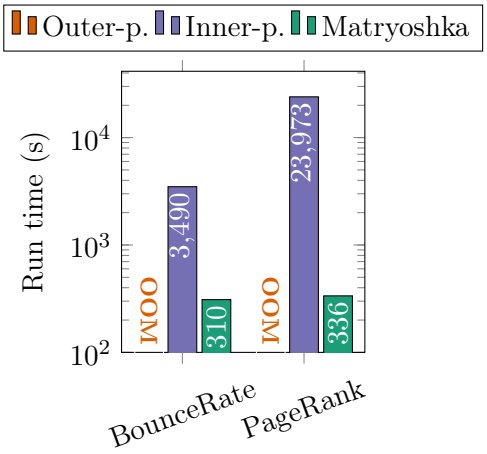


Figure 4.8: Skew handling.

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.

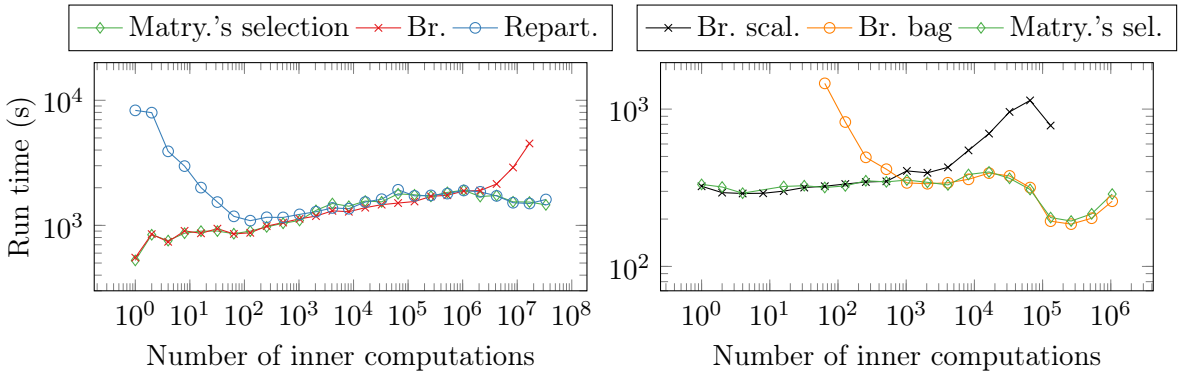
Matryoshka is not affected significantly (∼15%) by data skew, achieving 71× better performance than baselines.

4.8.7 Optimizations

We now study the efficiency of our optimizations discussed in Section 4.6.

4.8.7.1 InnerBag-InnerScalar Joins

We performed an experiment with PageRank to evaluate the effectiveness of Matryoshka to select the right join algorithm (broadcast vs. repartition, see Section 4.6.2) when varying the number of inner computations. Figure 4.9a shows the results. We observe that Matryoshka's optimizer is highly effective in selecting the right algorithm at any number of inner computations. It selects the broadcast join when having a small number of inner computations and selects the repartition join when having a very big 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 join when the number of inner computations is small. The reason for this is that a repartition join gives elements to only that many partitions as there are distinct keys, which is exactly the number of inner computations. Thus, for a small number of inner computations, there is not enough work for all CPU cores. In contrast, the broadcast join can be up to 3× slower than the repartition 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.



(a) InnerBag-InnerScalar join strategies. (b) Impl. strat. for half-lifted MapWithClosure

Figure 4.9: Optimization experiments.

4.8.7.2 Half-lifted MapWithClosure

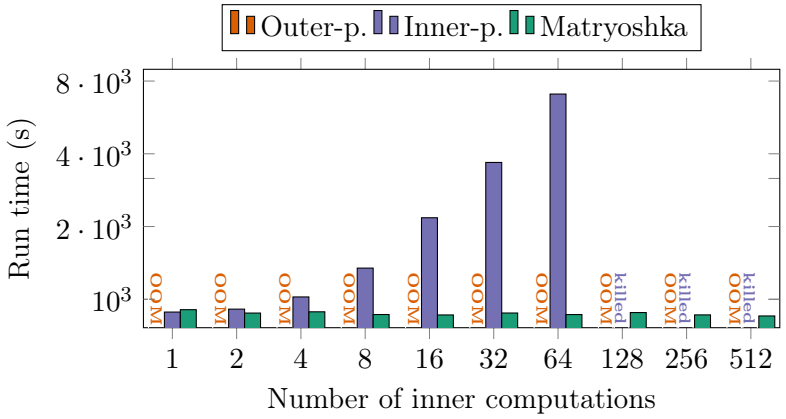
We performed an experiment with K-means where we tried the different strategies for half-lifted `mapWithClosure` (see Section 4.6.3). We can see in Figure 4.9b that our optimizer always makes the optimal choice, which prevents Matryoshka to crash or to fall into big performance degradations (up to 4.6×).

Matryoshka selects the best operator implementation in most cases, preventing a program from crashing or from being more than an order of magnitude slower.

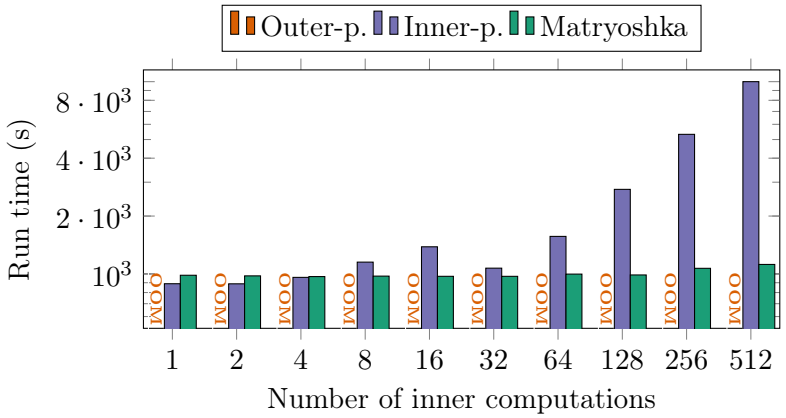
4.8.8 Larger Datasets

We also used a larger cluster to run the weak scaling experiment (Section 4.8.2) with 8× larger input sizes than in the previous experiments. 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 4.10 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.

Matryoshka scales to hundreds of GBs of input data size on 1,440 CPU cores, achieving more than one order of magnitude speedup over the inner- and outer-parallel baselines.



(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 4.10: Larger total input size.

Chapter 5

Related Work

We will first survey the research literature in control flow handling in distributed dataflow systems (Section 5.1) and then in nested parallelism (Section 5.2).

5.1 Control Flow Handling in Dataflow Systems

We will primarily focus on DDS, the approaches they take when handling loops and other control flow statements, and examine these approaches from both the efficiency and ease-of-use perspectives. Among the classes of systems discussed are:

- MapReduce extensions that add support for loops [34, 60, 61, 63, 193];
- DDS with operators beyond MapReduce[1] (e.g., Spark, Flink, Naiad, TensorFlow);
- Systems with high-level programming models that compile to systems belonging to the classes mentioned above (e.g., AutoGraph [126], Emma [9, 11, 12], Janus [92, 93], SystemML/SystemDS [24, 25, 74]);
- We take a slight detour to systems specialized for a particular data model, namely graphs (e.g., GraphLab [116], Pregel [117]) and arrays (SciDB [31, 170]). We believe it is good to know about these specialized systems when designing general dataflow systems, since the specializations were motivated partly by insufficient loop support in the more general systems.

MPI (Message Passing Interface) [169] is a message passing standard designed for parallel computing architectures. It employs a classic distributed programming model, where it is possible to implement iterative algorithms. However, MPI is different in that its programming model is at a lower level relative to the programming models employed in DDS considered in this thesis. Although MPI can be highly efficient, using it requires a larger programming effort and expertise in contrast to DDS.

Section 5.1.3.4.4 discusses the closest related works to Mitos: compiling imperative APIs to in-graph control flow. Section 5.1.1 gives a brief overview of the most important programming models for expressing loops. Section 5.1.2 discusses the most important design choices for control flow support in DDS. Section 5.1.3 introduces various programming models used for control flow

[1] We treat MapReduce also as a "dataflow system," albeit where the structure of the dataflow graph is fixed.

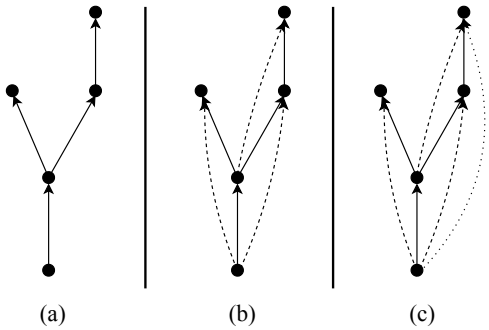


Figure 5.1: An illustration of the transitive closure of a digraph, where (a) depicts the initial graph, (b) shows new edges (as dashed lines) at iteration step one, and (c) reflects yet another new edge (as a dotted line) at iteration step two, after which the computation is finished.

in DDS. Lastly, Section 5.1.4 discusses optimizations that DDS commonly employ in handling control flow.

5.1.1 An Overview of the Programming Models

In this section, we give a brief overview of the approaches for expressing control flow adopted across several programming models. To ease our upcoming discussion about programming models in this section, let us introduce an example: computing the Transitive Closure (TC) of a digraph, $G = (V, E)$, which yields a new digraph $G^* = (V, E^*)$, where $E^* = \{(i, j) :$ a path exists from vertex i to vertex j in $G\}$. Figure 5.1 illustrates the steps taken when computing the TC of a digraph.

There are numerous algorithms to compute the TC of a digraph [42]. A simple algorithm would start from the set of edges of a graph G and then repeatedly calculate a new set of edges by performing an equi-join between the current edge set and the original edge set. For example, after the first step, the current edge set includes such (u, v) vertex pairs where v is reachable from u by a path of length at most 2. After the subsequent step, we would include vertex pairs reachable by a path of length 3, and so on. The algorithm terminates once the equi-join is no longer able to add any new vertex pairs. Figure 5.2 depicts a representative dataflow job for this simple TC algorithm.

Next, we highlight the implementation of the TC algorithm across four different programming models: Datalog, SQL, functional control flow, and imperative control flow. We will transition from programming models that are more declarative to those that are less so. To ease comparison of the implementations across the four models, program statements that correspond to a common operation will be colored in red, blue, and green, respectively.

5.1.1.1 Datalog

Datalog is a declarative logic programming language. An implementation of the TC algorithm in Datalog [42] is simply:

1: $Closure(x, y) \leftarrow Edges(x, y)$
2: $Closure(x, y) \leftarrow Closure(x, z)$ AND $Edges(z, y)$.

Line 1 represents initialization: if an edge exists between two vertices x and y, then the TC will

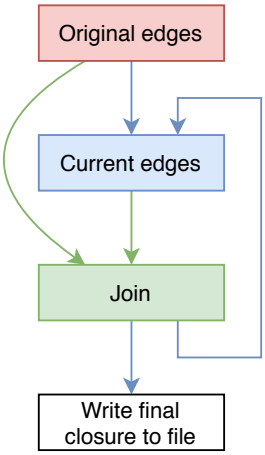


Figure 5.2: A dataflow job for the transitive closure algorithm. The colors map to the colored lines in the example codes. After the last iteration step, the final closure is written to a file (not shown in the code listings).

include the (x, y) edge. Line 2 is a join step: if an (x, z) edge is part of the closure, and there is also a (z, y) edge in the original graph, then the (x, y) edge should also be part of the closure. In this manner, the $x \to z$ path is extended by one edge. Note that Datalog has set semantics, which ensures that an edge can be added only once to the closure.

Since Datalog is declarative, there are varying execution strategies for a program. A common execution strategy is the so-called *bottom-up evaluation* [42]. In the case of our example above, line 1 is executed and then the join specified in line 2 is executed iteratively until no additional edges are added to the closure. This execution strategy makes Datalog relevant for this survey: Although there is no explicit loop statement in Datalog, loops often appear in system implementations.

For more details about Datalog, see Section 5.1.3.1.2.

5.1.1.2 SQL

An implementation of the TC algorithm as a recursive SQL query:

```
1: WITH RECURSIVE Closure(from, to) AS
2:     (SELECT from, to FROM Edges)
3:   UNION
4:     (SELECT R.from, E.to
5:      FROM Closure R, edges E
6:      WHERE R.to = E.from)
7: SELECT * FROM Closure.
```

This approach involves more boilerplate than the Datalog version, but essentially specifies the transitive closure the same way. The "with recursive" clause specifies that the definition of the *Closure* relation will involve the *Closure* relation itself. The "union" keyword specifies that the results of line 2 and lines 4-6 must be unioned (with set semantics, i.e., duplicates eliminated). Finally, line 7 is a required part of every recursive query in SQL, and could be used to select a subset of the *Closure* relation, but here we just select the entire relation. In contrast to SQL,

Datalog does not require an explicit keyword for marking recursive queries. Furthermore, a keyword for union is also not required in Datalog. Instead, the left-hand side of lines 1 and 2 both being *Closure* automatically implies union. For further details about recursive SQL queries, see Section 5.1.3.1.2.

5.1.1.3 Functional Control Flow APIs

Like in Flink, Naiad, and TensorFlow, but unlike Spark, the implementation of the TC algorithm below uses a higher-order function[2] to specify the loop:

```
1: Edges.fixpointIterate(curClosure => {
2:   curClosure.join(Edges)
3:     .where(_.to).equalTo(_.from) {(left,right)=>
4:       (left.from, right.to)
5:     }
6:     .distinct()
7: }).
```

Here, the argument to the higher-order function *fixpointIterate* is the loop body expressed as a λ-expression (i.e., an anonymous function given inline). Lines 2-5 perform a join between the current state of the *curClosure* dataset (corresponding to the current set of edges) and the *Edges* dataset (corresponding to the original edges), whereas line 6 filters out duplicates.

The fixed point of a function f is a value c for which $f(c) = c$. It is for this reason that the term "fixpoint" appears in *fixpointIterate*, since the final state of the closure dataset is a fixpoint of the body function: $body(curClosure) = curClosure$. Note that both Datalog and SQL also continue to evaluate the query until reaching a fixpoint. For further details about expressing loops via higher-order functions, see Section 5.1.3.3.

5.1.1.4 Imperative Control Flow

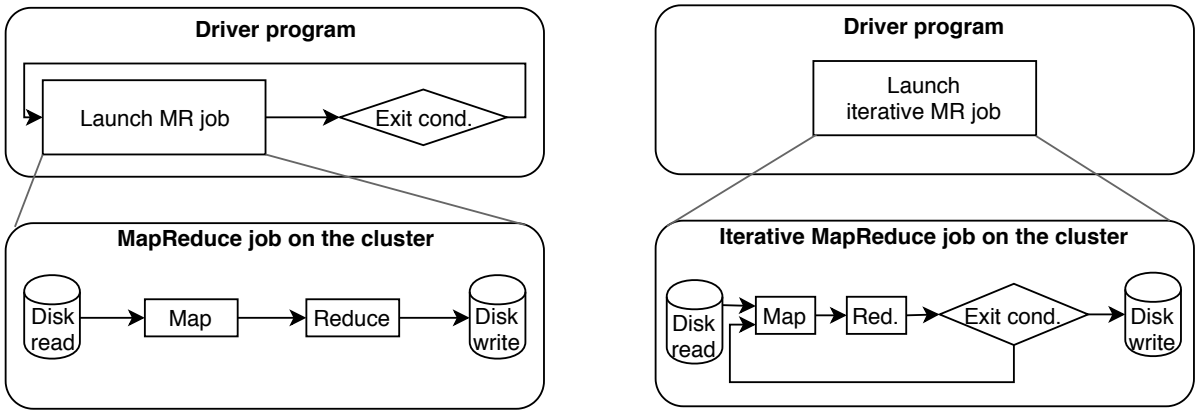
We now show a TC implementation with an imperative control flow API, i.e., a standard *while* loop, similar to Spark, Emma [9, 11, 12], and Mitos.

```
1: Closure = Edges
2: do
3:   OldCount = Closure.count()
4:   Closure = Closure.join(Edges)
5:     .where(_.to).equalTo(_.from) {(left,right)=>
6:       (left.from, right.to)
7:     }
8:     .distinct()
9: while Closure.count() ≠ OldCount.
```

Such an imperative API is more natural for many users than the previous functional API. This is because data scientists are used to the standard, imperative control flow constructs from languages such as MATLAB, Python, and R. However, as we will later see, programs written in an imperative API are harder for the system to execute efficiently.

We should mention that in line 9 (the loop exit condition), we exit the loop once the number of elements in the closure no longer grows. This is in contrast to the previous three examples,

[2] In this context, a higher-order function is a function that takes another function as an argument.



(a) (b)

Figure 5.3: The difference between standard MapReduce and iterative MapReduce systems. In standard MapReduce, a driver program launches jobs individually for each iteration step. On the other hand, iterative MapReduce systems, such as Twister [58,60], need only launch a single job, which includes all of the iteration steps. Note that these two approaches exist also in later DDS, where the map and reduce steps are replaced by a larger dataflow graph. Also note that there might be disk I/O also between the map and reduce steps. Figure adapted from Bu et al. [34].

which exit the loop when a fixpoint is reached. Instead, since the loop body is monotone (i.e., once an edge appears in the closure, it will always be there in subsequent iteration steps), it is sufficient to evaluate the number of elements in the closure as the loop exit condition.

5.1.2 Key Design Choices

In this section, we will discuss the key design choices that affect control flow support in DDS:

- control flow execution approach,
- expressivity of loop APIs, including support for global state, and
- fault tolerance.

These choices influence both the performance and the usability of a system. Table 5.1 provides an overview of the key characteristics of systems, whereas Table 5.2 highlights the expressivity of loop APIs by DDS.

5.1.2.1 Control Flow Execution Approach

The simplest approach to implement control flow, such as loop, is to launch a separate dataflow job for each iteration step [187]. This approach can be employed as a workaround even when a dataflow system does not explicitly provide any control flow support: the user can write a driver program to execute the control flow (e.g., by writing a while loop in Java) and submit a series of dataflow jobs one by one (denoted as *separate jobs* in Table 5.1). However, this approach has several performance issues: since the dataflow system is unaware of the control flow, some optimizations cannot be performed. Furthermore, there will be a job launch overhead at each iteration step.

A more efficient approach is to integrate a loop into a single dataflow graph (denoted as *In-Graph Loops* in Table 5.1), such as in Mitos. In this approach, a loop is executed entirely in a single dataflow job without involving the driver program during the loop execution. Since there is no job launch overhead at each iteration step, the per-step overhead can be 1-2 orders of magnitude less than the separate jobs approach[3] (see Figure 3.7), which can lead to an overall speedup of several times [72, 118] (see Figure 3.8). A further advantage of in-graph control flow is that it enables loop optimizations, such as loop-invariant hoisting and loop pipelining. We will come back to loop optimizations in Section 5.1.4.

The third approach [61] is a hybrid of the previous two approaches (denoted as *Reuse Tasks But Involve the Driver* in Table 5.1): In this case, there is only one dataflow job, which executes all steps, but control is returned to the driver program after each iteration step. At these times, the driver program launches the next iteration step (if the exit condition has not been met), for which it reuses the job and its constituent tasks (which are already present on the machines of the cluster).

The fourth approach extends a running dataflow job with additional dataflow nodes and edges for each iteration step [133] (denoted as *Extend Job* in Table 5.1). Although this avoids the job launch overhead at every iteration step, there is still an associated overhead due to the creation of new tasks. Note that some papers [41, 120] use the terminology of "extending a job" (or dataflow graph) differently: They also use it for the situation where we would say that a later job consumes cached datasets of earlier jobs, e.g., for the case when in a Spark program there is an action in every iteration step. We avoid this usage since Spark's web UI clearly shows that, in this case, later iteration steps that consume a cached dataset from an earlier step are new jobs. Furthermore, one could use the terminology "extending a job" for lazily adding operations to a not-yet-executed dataflow job. We avoid also this usage to prevent ambiguity with the mechanism of extending a dataflow job that has already run (at least partially), which is a very different mechanism from the lazy job building before job execution starts.

Thus far we have discussed approaches for implementing loops. Next, we will discuss how these approaches apply also for conditionals (i.e., *if* statements). Like loops, conditionals can also be implemented as multiple dataflow jobs: The first job would evaluate a condition, and then a subsequent job would be launched for either the *then* branch or the *else* branch. However, this does not work if a conditional is within a loop and the loop is implemented using in-graph control flow. In this case, the entire conditional must be implemented in the same dataflow graph as the loop. We show whether a system supports this in the *In-Graph Cond.* column in Table 5.1. If a system does not provide support for this, then a simple, but wasteful workaround is to execute both branches in the same dataflow job, and have a node that chooses between the results.

5.1.2.2 Expressivity of Loop APIs

For our purposes, the *expressivity* of a language (and the associated system) is the ease with which users are able to express mechanisms that arise in large-scale data analytics. We discuss the following mechanisms:

- fixpoint loops,

[3] However, there are several optimization methods available to reduce the job launch overhead [118, 139], which will be discussed in Section 5.1.4.1.

Table 5.1: Key design choices of the surveyed systems. Note that systems with a functional model also allow for imperative control flow, simply by launching separate dataflow jobs per step (as a workaround for expressivity problems). However, this negatively affects the performance.

Control Flow Model	System(s)	DSL Design Approach	Control Flow Execution					Fault Tolerance	Open Src.
			Separate Jobs	Extend Job	Reuse Tasks But Involve the Driver	In-Graph Loops	In-Graph Cond.		
Recurs. Queries	BigDatalog/RaSQL [79, 163]	External	Yes	Yes	Some	No	No	Spark	Yes
	Cog [87]	External	Yes	No	No	Yes	No	Flink	Yes
Driver (Imperative)	Spark [189]	Emb. (type)	Yes	No	No	No	No	RDDs	Yes
	FlumeJava [45]	Emb. (type)	Yes	No	No	No	No	MapReduce	No
Imperative	Emma [11, 12]	Emb. (metapr.)	Yes	No	No	Some	No	Spark/Flink	Yes
	Mitos (Emma) [72]	Emb. (metapr.)	Yes	No	No	Yes	Yes	Async. Checkpoints	Yes
	AutoGraph (TensorFlow 2.0) [126]	Emb. (metapr.)	Yes	No	No	Yes	Yes	TensorFlow	Yes
	Janus [92, 93]	Emb. (metapr.)	Yes	No	No	Yes	Yes	TensorFlow	No
	Swift for TensorFlow	Emb. (metapr.)	Yes	No	No	Yes	Yes	TensorFlow	Yes
	Myria [80, 178]	External	Yes	No	No	Yes	No	Save shuffle prod. side + opt.	Yes
	SystemML/SystemDS [24, 25, 74]	External	Yes	No	No	No	No	MR/Spark	Yes
	Musketeer [75]	External	Yes	Yes (IR)	No	Some	No	Varies based on backend	Yes
Functional	Flink [10, 65]	Emb. (type)	Yes	No	No	Yes	No	Async. Checkpoints [184]	Yes
	Naiad [120, 132]	Emb. (type)	Yes	No	No	Yes	No	Sync. Checkpoints	Yes
	DryadLINQ [187]	Emb. (type)	Yes	No	No	No	No	Dryad (re-execute vertices)	Yes
	Dandelion [154]	Emb. (mixed)	Yes	No	No	Yes	No	Async. Checkpoints	No
	Optimus [102]	Emb. (type)	Yes	Yes	No	No	No	Dynamic replication subgr.	No
	TensorFlow (w/o AutoGr.) [186]	Emb. (type)	Yes	No	Yes	Yes	Yes	Async. Checkpoints	Yes
	MXNet Symbol, Gluon [47]	Emb. (type)	Yes	No	No	Yes	Yes	Checkpoints	Yes
	Theano [8]	Emb. (type)	Yes	No	No	Yes	Yes	N/A (single machine)	Yes
	Wayang/Rheem [2, 3, 4, 107]	Emb. (type)	Yes	No	No	Yes	No	Varies based on backend	Yes
	Gilbert [149]	External	Yes	No	No	Yes	No	Flink	Yes
	MRQL [66]	External	Yes	No	No	Yes	No	Spark/Flink	Yes
	DIQL [67]	Emb. (metapr.)	Yes	No	No	Yes	No	Spark/Flink	Yes
Iterative MapReduce	CGL-MapReduce [61]	Emb.	Yes	No	Yes	No	No	NaradaBrokering [141]	No
	Twister [58, 60]	Emb.	Yes	No	Yes	No	No	Checkpoints	Yes
	HaLoop [34, 35]	Emb.	Yes	Yes	No	No	No	Hadoop+Checkpoints	Yes
Other	CIEL [133]	Emb.	Yes	Yes	No	No	No	Lineage	Yes

Table 5.2: Expressivity of the loop APIs. For systems that provide a functional control flow API we consider the expressivity of using this functional API. Note that in these systems one can also write imperative control flow using the separate jobs workaround. In this case expressivity improves, but there is a performance penalty.

Control Flow Model	System(s)	DSL Design Approach	Arbitrary Loop Cond.	Nested Loops	Multiple Loop Variables	Scalars in Loops
Recurs. Queries	BigDatalog/RaSQL [79, 163]	External	Only fixp.	Yes	Yes	No
	Cog [87]	External	Only fixp.	No	No	No
Driver (Imperative)	Spark [189]	Emb. (type)	Yes	Yes	Yes	Yes
	FlumeJava [45]	Emb. (type)	Yes	Yes	Yes	Yes
Imperative	Emma [11, 12]	Emb. (metapr.)	Yes	Yes	Yes	Yes
	Mitos (Emma) [72]	Emb. (metapr.)	Yes	Yes	Yes	Yes
	AutoGraph (TensorFlow 2.0) [126]	Emb. (metapr.)	Yes	Yes	Yes	Yes
	Janus [92, 93]	Emb. (metapr.)	Yes	Yes	Yes	Yes
	Swift for TensorFlow	Emb. (metapr.)	Yes	Yes	Yes	Yes
	Myria [80, 178]	External	Yes	No	Yes	No
	SystemML/SystemDS [24, 25, 74]	External	Yes	Yes	Yes	Yes
	Musketeer [75]	External	Yes	Yes	Yes	Yes
Functional	Flink [10, 65]	Emb. (type)	Yes	No	No	No
	Naiad [120, 132]	Emb. (type)	Yes	Yes	No	No
	DryadLINQ [187]	Emb. (type)	Yes	Yes	No	No
	Dandelion [154]	Emb. (mixed)	Yes	?	No	No
	Optimus [102]	Emb. (type)	Yes	?	No	No
	TensorFlow (w/o AutoGr.) [186]	Emb. (type)	Yes	Yes	Yes	Yes
	MXNet Symbol, Gluon [47]	Emb. (type)	Yes	Yes	Yes	Yes
	Theano [8]	Emb. (type)	Yes	Yes	Yes	Yes
	Wayang/Rheem [2, 3, 4, 107]	Emb. (type)	Yes	Yes	No	No
	Gilbert [149]	External	Yes	No	Yes	Yes
	MRQL [66]	External	Yes	Yes	No	No
	DIQL [67]	Emb. (metapr.)	Yes	Yes	Yes	Yes
Iterative MapReduce	CGL-MapReduce [61]	Emb.	Yes	No	No	No
	Twister [60]	Emb.	Yes	No	No	No
	HaLoop [34, 35]	Emb.	Approx. fixp. or num. steps	No	No	No

- arbitrary loop conditions,
- nested loops,
- passing more than one dataset (or scalar) between iteration steps,
- interleaving non-system code with control flow, and
- scalar values in loops.

5.1.2.2.1 Fixpoint Loops

Many algorithms fall under a Fixpoint Loop (FL) scheme, where a function f is applied repeatedly until convergence, as depicted below.

1: $xs = ...$ // initialization
2: **do**
3: $\quad xs = f(xs)$
4: **while** xs changed in this step.

The support for FL varies across systems. Recursive queries in SQL and (classic) Datalog only support monotonic FI. In monotonic FI, f is a monotonic function, and thus the set xs is monotonically increasing (i.e., elements are only added, but never removed). For example, the TC algorithm has this property. In contrast, the well-known PageRank algorithm does not: page ranks evolve over iteration steps, and consequently records (which contain the ranks) are replaced. There are many other similar, non-monotonic graph algorithms, where vertex labels continue to change, and thus cannot easily[4] be implemented as a recursive SQL query or Datalog program.

5.1.2.2.2 Arbitrary Loop Conditions

In many systems, users are free to write loops using arbitrary exit conditions. Besides FL, which is a special case of arbitrary loop conditions, another common termination condition is evaluating the delta between successive collections. For example, this is useful in PageRank, because the ranks are floating point values that may change by very small amounts across a great many iteration steps. In this case, the standard fixpoint exit condition would not work well, because it would lead to far too many iteration steps with insignificant differences. We can solve this problem if arbitrary exit conditions are allowed by a system: we specify a distance function and exit when it is below a threshold.

The use of distance functions as described above can sometimes be mirrored in the FL case: When updating the rank of a particular page, we can check whether the update is too small. If it is indeed too small, then we do not actually perform the update. By modifying the algorithm in this manner, the standard fixpoint exit condition is now sufficient to implement PageRank.

5.1.2.2.3 Nested Loops

Some algorithms require two or more nested loops. For example, a common algorithm for computing strongly connected components requires two nested loops [120]. Another example

[4]There is a workaround: we can add the iteration step number to the records as an extra component, as shown in the appendix of [34] via an implementation of k-means clustering, and in [32] via a translation of the Pregel model to Datalog. Seo et al. [159] implement PageRank in this way.

```
1: G = read(...); // Read graph as a matrix
2: authorities = round(G);
3: hubs = authorities;
4: maxiter = ...
5: toler = ... // Convergence tolerance
6: converge = False
7: iter = 0
8: while !converge do
9:   hubs_old = hubs
10:  hubs = G %*% authorities
11:  authorities_old = authorities
12:  authorities = t(G) %*% hubs
13:  hubs = hubs/max(hubs)
14:  authorities = authorities/max(authorities)
15:  delta_hubs = sum((hubs - hubs_old)^2)
16:  delta_authorities = sum((authorities - authorities_old)^2)
17:  converge = delta_hubs < toler and delta_authorities < toler or iter > maxiter
18:  print(delta_hubs)
19:  print(delta_authorities)
20:  iter = iter + 1
21: end while
```

Figure 5.4: An implementation of HITS [106], taken from Apache SystemDS[5]. t(G) denotes the transpose of the matrix G, and %*% denotes matrix multiplication.

is Li et al.'s [112] implementation of SimRank, which uses three nested loops. Furthermore, training a machine learning model also involves nested loops. Typically, there is an inner loop to find good model parameters and an outer loop to find good hyperparameters. Moreover, a third loop may be required when using k-fold cross-validation. As a last example, some training algorithms (e.g., k-means clustering) are sensitive to the starting values, in which case multiple starting values are typically tried in a loop.

Although Datalog does not have explicit loop statements, we wrote *yes* under *Nested Loops* in Table 5.2. This is because the dataflow graphs that result from certain Datalog programs can be similar in structure to the dataflow graphs resulting from nested loops in other control flow models.

5.1.2.2.4 Passing More Than One Loop Variable Between Steps

In some programming models, such as SQL, passing a dataset between iteration steps is implicit, and restrictive: one can only pass a single dataset. However, passing more than one dataset is useful, for example, in algorithms such as Gaussian non-negative matrix factorization [74] or HITS [106]. Figure 5.4 shows an example of HITS, where both the *hubs* and *authorities* matrices are passed between iteration steps. It can also be useful for keeping track of meta-information about the loop, such as the learning rate.

[5] https://github.com/apache/systemds/blob/v2.0.0-rc3/src/test/scripts/applications/hits/HITS.dml

5.1.2.2.5 Interleaving Non-System Code with Control Flow

It is straightforward to execute non-system code (i.e., code that is not using the system's API) at every iteration step in systems that execute loops as separate dataflow jobs per step. For example, for testing and debugging purposes, we would often like to collect various meta-information about a loop, such as the number of steps, the time of each step, the value of a loss function at each step, etc. This is trivial to achieve in a standard, imperative *while* loop that launches a new dataflow job for every step: we can just add a few extra lines of code that run in the driver program, outside the dataflow job. These extra lines run in each iteration step and collect or print the desired information. Figure 5.4 shows an example where two values that determine convergence are printed at each step. This information can help the user understand the algorithm's convergence behavior.

On the other hand, for an in-graph loop, all aspects of the program have to be built into the dataflow job, since all of the steps are in a single dataflow job and the driver program is not involved at every step. This is an issue in APIs where a loop is built by higher-order functions, such as Flink. In this case, we have to rely on built-in system functionality to collect meta-information[6].

5.1.2.2.6 Scalar Values in Loops

There are times when it is convenient to use a simple scalar (non-collection) variable, which takes different values at each iteration step. Examples of this include loop counters, learning rates during training in machine learning, and the comparison of the loss function values between consecutive iteration steps. Variables of this sort are trivially supported in systems with imperative control flow, such as Spark or SystemML. Figure 5.4 illustrates an example with several scalar values, which determine when to exit a loop.

However, scalars can be more difficult to work with in more rigid programming models or loop APIs that only support collections of values. In such a programming model, we can emulate scalars by wrapping them in 1-element collections. Such a wrapping is performed automatically by Mitos: The user's program can have scalars, and Mitos' compilation procedure turns them into 1-element collections Section 3.3.1.

5.1.2.3 Fault Tolerance

Fault tolerance is vital in DDSs, since many machines are employed and thus there is a high chance that some machines will fail. Thus, we now turn our attention to the interplay between loops and fault tolerance. For an overview of the fault tolerance mechanisms employed in the varying DDSs, these are listed in Table 5.1. It is worth mentioning that some systems rely on other systems in their execution layers (i.e., their backends). In these systems, the fault tolerance will just refer to the backend system. For example, Emma can compile programs to either Flink or Spark, and therefore Emma's fault tolerance is provided by Flink or Spark, respectively.

5.1.2.3.1 Separate Dataflow Jobs

If we execute a loop in separate dataflow jobs, then we can just rely on the system's standard mechanisms for fault tolerance. For example, if each iteration step is launched as a separate

[6]For example, `https://issues.apache.org/jira/browse/FLINK-1759`

MapReduce (MR) job, then MR provides fault tolerance: each job is itself fault tolerant, and intermediate results between jobs are materialized in a fault tolerant distributed file system, such as HDFS. However, performance can be further improved if we take advantage of the fact that we have an iterative program, which we discuss next.

5.1.2.3.2 Twister

Twister [58, 60] is an iterative MR system, with a programming model that repeatedly executes MR computations. Ekanayake et al. [58, 60] employ checkpoints between iteration steps, and restart the current iteration step in the event of a failure. Note that this fault tolerance approach differs from the fault tolerance that we get if we simply repeatedly run traditional (non-iterative) MR jobs (e.g., Google's or Hadoop's MR implementation). Traditional MR provides more fine-grained fault tolerance at the cost of more disk I/O, which would negate a lot of the performance gains of Twister.

5.1.2.3.3 Spark – Resilient Distributed Datasets

Spark's distributed collection type is called RDD (Resilient Distributed Dataset) [188]. In Spark, fault tolerance is based on RDD *lineage*, *shuffle files*, and *checkpointing*. By default, Spark tracks RDD lineage, i.e., how each RDD was computed. When a machine fails while computing a certain RDD, Spark restarts the computation of those partitions of the RDD that were being computed on the failed machine. If an RDD is computed from another RDD (*parent RDD*), this will trigger the re-computation of some of the parent RDD's partitions as well, since RDDs are ephemeral[7] by default. In turn, this might trigger the recomputation of further parent RDD partitions, and so on. This re-computation process can go back all the way to the source RDDs (e.g., that were read from a reliable distributed filesystem, such as HDFS).

In many cases, Spark does not need to re-do the entire computation, but only a small part, which is proportional to the work of the failed machine only. First, if all of the involved RDDs only have narrow dependencies, i.e., each RDD partition has only one (or a few) parent partitions, then the above re-computation process is confined to only a few partitions of each of the involved RDDs. Second, for RDDs with wide dependencies, i.e., where each partition depends on *all* of the partitions of a parent RDD, fault tolerance is handled by saving shuffle files[8] (on the workers' local disks). This ensures that one partition of a shuffle's consumer side (i.e., the side after the network transfer) can be recomputed by a limited amount of work: reading a small part of each of the shuffle files from the producer side[9], plus recomputing those shuffle files which were on the failed machine.

Third, Spark also has a checkpointing[10] mechanism to further limit recomputations. A user

[7] By ephemeral, we mean that they are not saved anywhere, not even in local memory (just forwarded to some other RDD's computation, which is requesting to read it).

[8] *Shuffling* is the repartitioning of data, which is necessary for joins, grouped aggregations, etc. Shuffle files contain the data just before a shuffle's network transfer.

[9] Shuffle files are sorted (or hashed), and therefore the computation of one partition at the consumer side needs to read just a small interval of each shuffle file.

[10] *Caching* an RDD is different from *checkpointing*. From a fault tolerance perspective, the main difference is that *caching* is local (by default), and therefore RDD caching usually does not help much for fault tolerance: Typically, all cached RDDs lose some partitions upon a machine failure. These lost partitions are often exactly the ones that would be needed for a recovery, because of the scheduler placing partitions on the same machine with their dependencies.

can call the *checkpoint* method of an RDD, which causes Spark to save the RDD reliably in a distributed file system, such as HDFS. This means that Spark can forget the lineage of that RDD, since it can always reconstruct that RDD from the checkpoint instead of re-computing it from its parent RDDs.

Checkpointing RDDs is especially important in the case of iterative computations. The problem is that lineage can become excessively long: An RDD can refer back to RDDs created at earlier iteration steps, which also refer back to RDDs from even earlier steps, and so on. Spark cannot handle such long lineage chains (e.g., due to stack overflows when recursively traversing them[11]). Therefore, users have to insert checkpoints to make Spark forget the lineages from earlier steps. In cases where loops are hidden from the user behind higher-level APIs (e.g., GraphX or machine learning), this checkpointing can be performed automatically.

Checkpointing an RDD is an expensive operation, given that it involves writing it to disk, typically on three machines. Hence, it is worth trading-off checkpointing time with recovery time, i.e., checkpointing every nth (e.g., 10th) step, rather than at every step. The optimal n depends on both the mean time to failure (MTTF) and the step time, which can both depend on the size of the compute cluster. For a study on automatic checkpointing policies see Sharma et al.'s paper [160].

5.1.2.3.4 Myria

Myria's overall fault tolerance mechanism [179] is similar to Spark: It saves shuffle data at the producer side, which allows it to re-execute all the work of only the failed machine from the iterative program's beginning. However, Myria introduces two optimizations for iterative Datalog evaluation: recovered tuples are prioritized [192], and for Datalog programs with aggregations, saved shuffle data is aggregated across iteration steps (similar to a MR combiner).

5.1.2.3.5 Checkpointing Incremental Loops

In order to not introduce a significant overhead by checkpointing, we would like to checkpoint asynchronously, where checkpointing happens in the background, without suspending normal program execution. This is straightforward to achieve, for example, in Spark [188], since RDDs are immutable. Their immutability means that there is no danger that program execution would change them during taking a checkpoint.

However, Flink and Naiad support incremental loops, which they achieve by keeping mutable state across iteration steps (see Section 5.1.4.5). This means that taking a checkpoint of this state cannot be arbitrarily overlapped with program execution. The problem is that if the mutable state currently being checkpointed changes, then the checkpoint would become inconsistent.

To address the problem of checkpointing incremental loops, Xu et al. [184] explore varying approaches. In their experiments, they report that *head checkpointing*, i.e., performing a checkpoint at the beginning of an iteration step, is a good strategy. Since the loop's mutable state typically changes near the end of an iteration step, there is often enough time to perform the checkpoint from the beginning of the step to the point when the state starts to mutate. In this way, suspending the program execution can be avoided in many cases.

[11] `https://issues.apache.org/jira/browse/SPARK-5484`

5.1.2.3.6 Optimistic Recovery

Schelter et al. [157] and Dudoladov et al. [54] take a different approach to fault tolerance for iterative computations, one that does not involve taking checkpoints. Instead, they employ so-called *compensation functions*, which are provided by the user, and invoked when a failure happens. A compensation function should achieve a consistent system state that leads to the correct result at the end of the program execution, despite the system state not being identical to the state prior to the failure. This is possible due to the convergence properties of many iterative data analytics: they converge to the same end state from many different intermediate states.

5.1.2.3.7 TensorFlow

TensorFlow [1] is a machine learning system, which also explicitly represents global state (in addition to computations) in its dataflow graphs (e.g., for building parameter server architectures). To increase performance, TensorFlow supports asynchronous updating of parameters (in the microstep sense, see Section 5.1.4.3) with relaxed consistency. This means that the fault tolerance also does not need strong consistency: When a checkpoint is taken in parallel with a training step, the checkpoint might only include a part of the updates that the training step makes.

5.1.3 Programming Models

Iterative computation approaches and control flow constructs vary widely across programming models. In this section, we will delve into four classes of programming models. First, we examine two traditional programming models: SQL and Datalog. Second, we turn our attention to iterative MapReduce, such as CGL-MapReduce [61], Twister [60], HaLoop [34,35], iHadoop [63], and iMapReduce [193]. Third, we transition to programming models that incorporate control flow into dataflow jobs using higher-order functions in DDS, such as Flink, Naiad, TensorFlow, MXNet, Theano, MRQL, and DIQL. Fourth, we address programming models with imperative control flow constructs, such as Emma, Spark, and SystemML. Lastly, we shortly discuss CIEL.

5.1.3.1 SQL and Datalog

Now, we dive into the approaches taken in both SQL and Datalog to support the implementation of iterative algorithms.

5.1.3.1.1 Imperative Control Flow in SQL

Prior to the release of SQL:1999 (or SQL 3) support for iterative algorithms was unavailable. For example, as showed in Libkin et al. [114], the transitive closure of a graph could not be implemented in *classic* SQL. Today, however, modern SQL dialects and extensions offer several ways to implement iterative algorithms.

In one line of work, procedural extensions to SQL have appeared. Among these extensions are PL/SQL [69] (Procedural Language for SQL) and SQL/PSM [56] (SQL/Persistent Stored Modules). These solutions offer traditional imperative control flow constructs and mutable variables, which enable SQL statements to be embedded in *while* loops. Grust et al. [55, 85, 86]

compile PL/SQL queries to standard SQL (recursive) queries, in order to avoid the interpretive overhead of PL/SQL as well as context switches between PL/SQL and SQL.

Alternatively, we can employ a general-purpose programming language together with standard ways to embed SQL queries. For example, Java offers the Java Database Connectivity (JDBC) API, and C# offers LINQ [123]. With this approach, SQL statements can be embedded in imperative code using standard control flow constructs and executed repeatedly within these constructs (e.g., as depicted in Hellerstein et al.'s paper on the MADlib analytics library [83]). Likewise, modern DDS, such as Apache Flink and Apache Spark offer SQL APIs [14], which allow for a similar combination of standard imperative code and SQL queries.

5.1.3.1.2 Recursive Queries in SQL and Datalog

Recursive queries (RQ) are yet another approach to express iterative algorithms. In SQL, RQ are declared using the keywords *WITH RECURSIVE*. We can see an example in Section 5.1.1.2.

RQ were standardized in the SQL:1999 standard [57] on the basis of several research publications [5, 7, 15] and system-specific SQL extensions introduced in both IBM DB2[12] and Oracle. For further insight into the evolution of RQ in SQL:1999 through SQL:2008 and leading DBMS vendors, see the 2010 survey paper by Przymus et al. [147]. Today, there are ongoing efforts to add support for SQL RQ in both Spark[13] [79] and Hive[14].

RQ in standard SQL have several limitations. First, queries must be *monotonic*, i.e., a row appearing in a relation cannot make another row disappear. Thus, once a row r has been added to the result table, a subsequent execution of the recursive query cannot result in the disappearance of r. Second, the recursion must be *linear*, i.e., the query can refer to the result table only once. Since SQL is widely used, many researchers have proposed extensions to overcome these limitations and achieve greater expressivity [13, 124, 142, 194].

Like SQL, the Datalog [42] query language also supports recursion. To execute Datalog queries on large datasets, numerous scalable systems have arisen. In 2012, Shaw et al. [161] discussed how to compile linear Datalog programs into MapReduce jobs. Additionally, in 2012, Borkar et al. [30] described how to compile XY-stratified Datalog programs to Hyracks [29] (a distributed dataflow system). In 2015, Wang et al. [179] showed how to compile queries in yet another subset of Datalog into Myria [80, 178] (a distributed data management system) jobs. Shkapsky et al. [163] and Wu et al. [181] compile Datalog programs into Spark jobs, and Imran et al. [87, 88] compile a Datalog program into a Flink job, using Flink's in-graph loops [65].

Differential Dataflow [122] can also be the target of Datalog compilation. In this context, Ryzhyk et al. [155] emphasize the use case of embedded incremental deductive databases. Chothia et al. [49] focus on data provenance for iterative data analytics, and base their approach on Differential Dataflow. In addition, there are a number of unpublished, yet promising research efforts that build incremental Datalog evaluation systems based on Differential Dataflow's incremental nature[15]. By building on Differential Dataflow, these systems also have the potential to

[12] https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.5.0/com.ibm.db2.luw.apdv.porting.doc/doc/r0052877.html

[13] https://jira.apache.org/jira/browse/SPARK-24497

[14] https://issues.apache.org/jira/browse/HIVE-16725

[15] https://www.nikolasgoebel.com/2018/09/13/incremental-datalog.html
https://github.com/comnik/declarative-dataflow
https://github.com/TimelyDataflow/differential-dataflow/blob/23b04441b43ce695f4bbaa08305e8af43fcb7345/sosp2019-submission.pdf

support non-monotonic Datalog queries.

There are also standalone systems for large-scale Datalog evaluation [180,191], i.e., not building on an existing dataflow system. Yet another approach executes recursive SQL queries for machine learning in distributed databases [91].

5.1.3.2 Iterative MapReduce

MapReduce [51] is a popular programming model for large-scale data processing, which does not provide in-graph loop support. Although this limitation can be worked around by launching a separate MapReduce job for each iteration step, there are still performance problems, as previously mentioned. Therefore, MapReduce extensions have appeared that offer loop support, which we discuss next.

Systems that extend MapReduce to provide loop support include CGL-MapReduce [61], Twister [60], HaLoop [34, 35], iHadoop [63], and iMapReduce [193]. In these systems, a driver program submits a single job to the system, which then executes all iteration steps. The basic structure of these iterative MapReduce jobs is to execute the same map and reduce steps repeatedly until a termination condition is satisfied, as depicted in Figure 5.3. Executing all of the iteration steps in a single job enables these systems to perform several optimizations, which will be discussed in Section 5.1.4. In 2016, Lee et al. [110] performed an experimental comparison on four iterative algorithms and demonstrated that HaLoop, Twister, and iMapReduce outperform Hadoop.

Loop support across iterative MapReduce systems is exposed via user-facing APIs that closely follow the systems' internal architectures. To better understand an iterative MapReduce system's API, users need to dive into system internals. In the following, we examine five characteristics of iterative MapReduce system APIs that programmers should be aware of.

Rigid computation structure. Whereas CGL-MapReduce, and iHadoop only allow for one MapReduce computation per iteration step, both HaLoop [34, 35] and iMapReduce [193] allow for a sequence of MapReduce computations per iteration step. Further, iMapReduce allows for an "auxiliary" MapReduce computation to supplant built-in termination conditions with custom ones. However, unlike general dataflow systems (e.g., Flink and Spark), none of these systems allow for an arbitrary graph of operators.

Loop-invariant data handling. In Twister, loop-invariant input data can be read by mappers and reducers in a special "configure" phase, which happens only once, before the first iteration step. In HaLoop, a user can designate a dataset to be loop-invariant, which will cause it to be passed along to reducers in each iteration step as a separate argument to the reduce function with matching keys. That is, HaLoop will automatically perform an equi-join between the current reducer input and the loop-invariant dataset. (See also Section 5.1.4.2.)

Accessing data from previous iteration steps. In general, by default, the output of a MapReduce computation at a current step k serves as the input to a subsequent step $k + 1$. However, in HaLoop the input of a step can include the results of all the previous steps.

Termination conditions. Iterative MapReduce systems provide mechanisms that allow users to specify when a loop must terminate. The most common of these are fixpoints. However, they vary in how they handle approximate fixpoints (see Section 5.1.2.2.2).

Broadcasting data to all mappers/reducers. Many iterative algorithms need to broadcast shared information to all of the mappers or reducers in iterative MapReduce systems. For

example, in k-means clustering, the most recently computed clusters are broadcast to all of the mappers in the subsequent iteration step in iMapReduce [193]. To handle this additional communication, iterative MapReduce systems require custom API constructs.

5.1.3.3 Functional Control Flow APIs

Now we turn our attention to systems that offer higher-order functions for specifying control flow, such as Flink [10, 41, 64, 65], Naiad [120, 132], Dandelion [154], MRQL [66], DIQL [67], TensorFlow [1, 186], MXNet [47], and Theano [8]. They offer such a functional API instead of a standard imperative control flow API (such as Spark) to be able to have in-graph control flow, i.e., incorporate control flow into their dataflow jobs. This has several performance benefits by eliminating job launch overhead, and by enabling several optimizations, such as eliminating redundancy when dealing with loop-invariant data, enhancing data locality, and pipelining across iteration steps. We will discuss these optimizations in Section 5.1.4.

We can see an example of a functional control flow API in Section 5.1.1.3. Each control flow construct is expressed as a higher-order function, which takes user-defined function(s) that specify loop bodies, exit conditions, if branches, etc. Next, we discuss the programming models of the eight aforementioned DDS, each of which incorporates control flow into their dataflow jobs using higher-order functions, and conclude with some remarks about usability.

5.1.3.3.1 Flink

Apache Flink's [10, 41, 64, 65] collection-based API provides a loop operator with two variants. One is denoted as `iterate` for bulk loops and the other is denoted as *deltaIterate* for incremental loops. We will treat each one in turn.

Under the `iterate` variant, there are two types of termination conditions: one for a fixed number of iteration steps and one for a data-dependent number of iteration steps. In the first case, the syntax is

```
initialDataSet.iterate(numSteps, body: DataSet => DataSet)
```

where *body* is a user-defined function that builds the body of the loop as a dataflow, and *DataSet* is Flink's distributed collection type.

Note that basic collection transformations such as `map`, `filter`, `flatMap`, etc. are also higher-order functions, but they take very different kinds of functions as arguments. A function given to `map` takes just one element of a distributed collection and returns one element, while the body function of `iterate` takes a distributed collection and returns a distributed collection.

In the second case, the specification of a custom termination condition is as follows:

```
initialDataSet.iterWithTerm(body: DataSet => (DataSet, DataSet))
```

Here, the *body* function returns two collections, the second of which is the termination condition: exit the loop once this collection is empty.

Programming models are often heavily influenced by performance considerations, of which Flink's delta loop (or delta iteration) API [65] is a good example. In contrast with Flink's bulk loop shown above, the semantics of delta iterations are more complicated in order to make the programs using it more efficient. Delta loops are *incremental*, i.e., they exploit the fact that in many algorithms only a small part of the loop dataset changes per step. For example, in the

transitive closure example, it often happens that many steps add only a few new pairs per step. In this case, it is beneficial to not recreate the whole dataset at each step, but deal with only the change. Delta loops make this possible by allowing the body function to return only the new and/or changed elements of the main dataset. Therefore the execution time of a step in a delta loop can be proportional to the number of new and/or changed elements, rather than with the size of the whole dataset.

Although Flink offers several iterative constructs, there are two limitations. First, loops in Flink cannot be nested. Second, Flink does not allow multiple datasets to be passed between steps. However, both of these are just limitations of the current implementation.

5.1.3.3.2 Naiad, Differential Dataflow, and Timely Dataflow

Naiad [120, 132] is another DDS with a collection-based API. For loops, it has a programming model that looks similar to Flink's bulk loops: the user creates a fixpoint loop by calling a higher-order function and giving it the loop body as a UDF. However, Naiad executes this in a way that it automatically becomes an incremental loop, through the mechanism it calls *differential dataflow*, where each operator of the dataflow job can efficiently react to small changes in its inputs.

Naiad was originally written in C#, but a new implementation of Naiad's core ideas exists in Rust[16]. This implementation is modular: *Timely Dataflow* keeps track of iteration steps using a system of hierarchical timestamps, whereas *Differential Dataflow* adds a set of collection-oriented abstractions to the timely dataflow model.

5.1.3.3.3 DryadLINQ and Optimus

DryadLINQ [187] is a DDS with a collection-based API that leverages .NET LINQ. DryadLINQ programs are executed on Dryad [89]. Loops can be expressed using either the standard, imperative C# loop construct, or a higher-order function provided by DryadLINQ's API[17]. However, even when using the higher-order function, loop execution happens by launching a series of dataflow jobs. Via the API users can specify an unrolling factor U, which means that each dataflow job executes U steps of the loop [59]. We should note that the loop termination condition will only be evaluated at every Uth step.

Optimus [102] is a DDS built on DryadLINQ. It enables the run-time modification of dataflow graphs. This feature allows for dynamic loop unrolling, where the dataflow graph is repeatedly extended with copies of the loop body, for each iteration step. Since dataflow graph extensions happen at run time, all of the iteration steps can be unrolled dynamically without knowing the number of steps beforehand. In this manner, all of the iteration steps are in a single dataflow job, which simplifies both fault-tolerance and job monitoring, and eliminates the job launch overhead. However, the task launch overhead due to launching new operators is not eliminated, and it can be a significant part of the step overhead (see Section 3.6.5).

5.1.3.3.4 Dandelion

Dandelion [154] is a DDS with a similar API to DryadLINQ. However, it focuses on heterogeneous compute clusters, i.e., where there are also specialized compute cores, such as GPUs. It provides

[16] https://github.com/TimelyDataflow/

[17] `https://github.com/MicrosoftResearch/Dryad/blob/master/LinqToDryad/DryadLinqExtension.cs#L205`

a higher-order function for building loops, which enables in-graph loops, similarly to Flink and Naiad. In the context of GPUs, an in-graph loop means that Dandelion ships a loop to a GPU, and thus avoids context switching at every step.

In Table 5.1 we wrote *mixed* for Dandelion's DSL embedding. Dandelion uses cross-compilation for translating the .NET bytecode of UDFs to GPU code, which we classify as metaprogramming. However, for other parts of the code (including control flow constructs) it relies on LINQ types and their method calls, i.e., a type-based DSL embedding.

5.1.3.3.5 MRQL and DIQL

The MapReduce Query Language (MRQL [66]) is a SQL-like external DSL for large-scale data processing. It can compile queries to Hadoop MapReduce, Apache Hama, Spark, and Flink. It supports fixpoint loops using a special syntax. The termination condition is a user-specified Boolean value for each element in the collection that is passed to the next step, indicating whether an element has converged. The iterative process terminates when all of the elements have converged. The Data-Intensive Query Language (DIQL [67]) is a similar DSL, but embedded in Scala. Chlyah et al. [48] extend MRQL/DIQL's monoid algebra to perform loop-related optimizations, such as reordering operators and deferring shuffles to the end of loops.

5.1.3.3.6 TensorFlow, MXNet, and Theano

TensorFlow [1, 186], MXNet[18] [47], and Theano[19] [8] are three prominent dataflow systems for machine learning (ML). These three ML systems are not distributed in the same sense as Flink, Naiad, and Spark, given that distributed training in these systems requires another layer of programming, besides the single machine program (e.g., distributed training in a parameter server architecture, possibly with Keras[20]). However, they employ a dataflow programming model, which is suitable even for single-machine programs, which can be executed on GPUs. When executing dataflows on GPUs, similar design considerations apply to control flow support as in the case of distributed dataflow graphs (one dataflow job can execute an entire loop or just one iteration step, the control flow API can be imperative or functional, etc.).

Control flow is vital in ML systems. In 2018, Yu et al. [186] reported that within Google, about 65% of TensorFlow jobs contain conditional statements and about 5% contain loops, for such uses as processing sequential data in recurrent neural networks or agents performing a sequence of actions in reinforcement learning.

Like Flink and Naiad, control flow in TensorFlow, MXNet, and Theano is incorporated into the dataflow jobs via higher-order functions. Besides a function for building a *while* loop, TensorFlow and MXNet also offer a function for building conditionals, akin to an *if* statement.

The loop APIs of TensorFlow, MXNet, and Theano slightly differ from those of Flink and Naiad. For example, they support the passing of multiple loop variables between iteration steps, via the parameters of body functions, which allow tuples of variables (e.g., scalars, multidimensional arrays) as their data types. Another difference is that the loop exit conditions are determined by condition functions, independently of body functions. Condition functions take the same loop variables as the body function, but return a Boolean, which determines whether to exit a loop.

[18] `https://cwiki.apache.org/confluence/display/MXNET/Optimize+dynamic+neural+network+models+with+control+flow+operators`

[19] `http://deeplearning.net/software/theano/library/scan.html`

[20] `https://keras.io/`

Unfortunately, this design does not allow for the inspection of the values of loop variables from two successive iteration steps at the same time. However, this limitation can be overcome with additional loop variables to store the values from earlier steps. Lastly, another small variation is that both MXNet and Theano allow for returning the values of loop variables after each step (i.e., not just the value after the last step). Note that besides the above API, TensorFlow 2.0 also incorporates AutoGraph, which compiles from an imperative API to in-graph control flow, which we discuss in Section 5.1.3.4.4.

Yu et al. [186] argue that in-graph control flow support in dataflow jobs is advantageous in these systems because of the need for automatic differentiation, which is a commonly used feature when training neural networks. They argue that automatic differentiation can be better supported when a system can examine an entire program upfront, including control flow. However, it should also be noted that PyTorch can still perform automatic differentiation of loops [143], even though it is an *eagerly* evaluated machine learning framework, i.e., it executes control flow in the driver program instead of building it into dataflow graphs.

5.1.3.3.7 Usability

When expressing in-graph loops via higher-order function calls, it is important to keep in mind that body functions are called by the system

- exactly once (i.e., *not* for every iteration step),
- in the driver program,
- while building a dataflow job (i.e., *not* while the loop is running).

In other words, during the execution of a loop, the driver program is not involved, and therefore every aspect of the loop body must be built into the dataflow graph upfront. Consequently, the loop body can only contain operations that can be expressed using system-supported operators. As mentioned in Section 5.1.2.2.5, this can make it cumbersome to perform certain tasks which would be trivial if control flow is executed in the driver program instead of in-graph. Moreover, users have to be careful not to accidentally use an operation that is executed immediately, instead of as part of the dataflow job. These characteristics make these APIs unintuitive to new users, which is evidenced by dozens of questions on the *stackoverflow.com* website about `tf.while_loop`, `theano.scan`, and Flink's loops. Listing 3.1 shows an example comparing functional and imperative control flow APIs.

5.1.3.4 Imperative Control Flow in Dataflow Systems

Many DDSs have host languages that provide traditional imperative control flow constructs (e.g., *while* loop, *if* statement), such as Scala in Spark. In contrast to functional control flow APIs, which require greater effort to use, imperative control flow offers greater expressivity, and facilitates the representation of several concepts discussed in Section 5.1.2.2.

However, even though imperative control flow in DDSs offers several benefits for users, it is hard for the systems to efficiently execute imperatively written programs. We will present several approaches in this section. Table 5.3 shows an overview.

Table 5.3: Systems where control flow is expressed imperatively.

System(s)	Loop Unrolling	Separ. Jobs Explicitly	Separ. Jobs Implicitly	In-Graph	Tracing
Spark [189]	Yes	Yes	No	No	No
DryadLINQ [187]	Yes	Yes	No	No	No
FlumeJava [45]	Yes	Yes	No	No	No
Emma [9, 11, 12]	Yes	No	Yes	Some	No
Mitos (Emma) [72]	No	No	No	Yes	No
TensorFlow [126]	Yes	Yes	Yes	Yes	Yes
Janus [92, 93]	Yes	No	Yes	Yes	No
Myria [178]	No	No	No	Yes	No
SystemML/SystemDS [24, 25, 74]	Yes	No	Yes	No	No

5.1.3.4.1 Loop Unrolling

Recall that many of the distributed collection operations are *lazy* in the sense that they are not executed at once when the driver program calls them. Instead, they build up a dataflow job to be executed later, when an *action* is called. For example, creating a new distributed collection from an existing one is typically a lazy operation.

Lazy operations allow for the *unrolling* of a loop: If the loop body does not involve any actions, then the dataflow job will continue to grow at each iteration step, but not be executed while the loop is running in the driver program. Such a dataflow job will have many similar copies of the loop body chained one after the other.

Unfortunately, fully unrolling a loop is only possible in limited circumstances: specifically, when control flow does not depend on values computed in dataflow jobs. The problem is that most iterative algorithms require convergence checks and these depend on the results computed in the current iteration step. If we rely on the control flow constructs of the host language, then we need to get information from the distributed collections into the driver program (for example, summing up the values of a collection of changes). This would require actions to be called at every iteration step and thus dataflow jobs to be executed.

Also note that even unrolling a loop might not eliminate a signficant portion of the step overhead. This is because unrolling a loop can create a large number of operators, proportional to the number of iteration steps. Since each operator typically has an overhead from launching tasks, this means that a per-step overhead can still be present. We experimentally demonstrate this problem in Section 3.6.5.

There are instances when it is beneficial to partially unroll a loop: one job executes several, but not all iteration steps. This way, we can amortize some of the per-job overhead. Ekanayake et al. [59] explored partial loop unrolling in DryadLINQ, but found that the overhead is still large in comparison to frameworks that specifically optimize for iterative computations.

Nemeth et al. [135] compile machine learning model evaluations expressed in Julia [18] to parallel dataflows. They unroll control flow at compile time by "inlining" the input data into the model, i.e., they perform *partial evaluation* [94] of the model with the input data.

5.1.3.4.2 Explicitly Launching Separate Dataflow Jobs at Every Step

As mentioned before, a simple approach to loops in DDSs is to just let the driver program launch separate dataflow jobs for every iteration step[21]. However, if done naively, this approach can have a bad performance. One issue to keep in mind is that the distributed collections of DDSs are typically ephemeral by default. Collections are not saved (either in memory or disk) but recomputed if used in multiple jobs (which can happen due to various reasons in iterative programs, as we show later). Therefore, users should explicitly persist datasets between the dataflow jobs of different steps. However, if the system does not provide support for persistence within the system, then we have to write it to disk (in, e.g., HDFS).

Spark [188, 189] solves this problem by offering the user the option of keeping a dataset in the memory of the Spark worker processes between dataflow jobs. Specifically, Spark offers a method on RDDs (Spark's distributed collection type), called `cache`. Calling this method has the effect that Spark will save the elements of the collection when they get computed the next time, and then will use the cached elements if needed later. (Note that Spark also performs a form of caching automatically before shuffles.) A related method is `unpersist`, which removes a cached RDD. This can have several benefits: overriding Spark's default cache eviction scheme so that the right RDDs remain cached under tight resource constraints, reducing the work of the garbage collections by reducing the number of live JVM objects in the worker processes, etc. Note that paying attention to call `persist` and `unpersist` is extra user effort, which can be avoided with the alternative approaches in Section 5.1.3.4.3–5.1.3.4.4. Also note that, in some cases, Spark can automatically unpersist RDDs that went out of scope in the driver. However, this mechanism relies on the finalizer of the RDD being called in the driver, which happens only when a garbage collection in the driver happens. Since the driver itself has little memory pressure in many Spark programs, the automatic unpersisting might happen later than ideal. In the next paragraphs, we show how to use these Spark methods to handle the passing of intermediate data between iteration steps and to handle loop-invariant datasets. Figure 5.5 shows an example.

Intermediate data between iteration steps. If the intermediate data is written to the disk at the end of each step, and then read back at the beginning of the next step, this incurs a considerable performance overhead. Early papers on DDSs did not present a solution to this problem (DryadLINQ [187], FlumeJava [45], Hyracks [29]). In Spark, however, the user can call `cache` on those RDDs which will be used in the next iteration step, and call `unpersist` on those that were cached before but are not needed anymore. Also note that Shinnar et al. [162] discuss keeping intermediate data in memory between jobs in the context of Hadoop.

Loop-invariant datasets. These are such datasets that are used throughout a loop without any changes. For example, in the transitive closure example, the collection of edges is the same in all iteration steps. We would like such datasets to be computed (or read from disk) only once. In Spark, the user can achieve this by calling `cache` on such datasets before the loop. We will discuss optimizations for loop-invariant datasets in Section 5.1.4.2.

TensorFlow allows for both in-graph control flow and for a variation of the above approach of executing control flow in the driver program: we can create and distribute a graph once, and then trigger its execution repeatedly from the driver program (with different inputs, such as new batches of training data). This variation has a lower overhead than launching a completely new

[21] The approach of launching separate jobs works for branching constructs as well (*if* statement, *switch-case*): we launch one job to compute the condition the *if* condition, and then the appropriate branch executes a subsequent job.

```
1:  Edges = readFile(...)
2:  Edges.cache()
3:  Closure = Edges
4:  Closure.cache()
5:  NewCount = Closure.count()
6:  do
7:    OldCount = NewCount
8:    OldClosure = Closure
9:    Closure = Closure.join(Edges)
10:     .where(_.to).equalTo(_.from) {(left, right) =>
11:       (left.from, right.to)
12:     }
13:     .distinct()
14:   Closure.cache()
15:   NewCount = Closure.count()
16:   OldClosure.unpersist()
17: while NewCount ≠ OldCount
```

Figure 5.5: The transitive closure example, extended with caching in the style of Spark. The blue lines are managing which datasets to keep in memory between dataflow jobs, The red lines are actions, which trigger job executions.

graph at every iteration step, but in-graph loops are still faster: Yu et al. [186] report an 5× difference in TensorFlow in a microbenchmark.

Note that most systems that provide support for in-graph control flow through a functional API typically also allow the user to fall back to expressing control flow imperatively in the driver program and executing each step as a separate dataflow job. However, in that case, the user obviously does not get the performance benefits of in-graph loops.

5.1.3.4.3 Leaving Execution Details to the System

Ideally, we would like systems to automatically handle low-level execution details, such as caching. However, there is no easy way to extend Spark with automatic caching of RDDs. The problem is that Spark's API is a type-based embedded DSL (see Section 2.3) that is only aware of method calls involving RDDs and unaware of other parts of a driver program, such as the imperative control flow constructs. Moreover, Spark does not directly recognize that there is a loop or that an RDD will be used again in a subsequent job. However, recent systems, such as Apache SystemML/SystemDS [24, 25, 74], and Emma [9, 11, 12] have a more holistic view of programs: they hide execution details from users, yet still provide imperative control flow constructs. Next, we discuss these systems.

SystemML [25, 74] is a large-scale machine learning system, whose programs are expressed in a declarative language called DML, a DSL with an R-like syntax. The system compiles DML programs to MapReduce or Spark. DML is an *external DSL* [70], i.e., it is not embedded in a general-purpose language, but rather is an independent language. SystemML has complete responsibility for the user programs written in this language, and therefore it is naturally aware of everything in the source code, including loops and other control flow. This allows it to

Table 5.4: Control flow handling approaches in DDS. In-graph loops allow for better performance than executing control flow in the driver program with launching separate dataflow jobs. Imperative control flow APIs are easier to use than the functional ones.

Ease of Use ↑ \ Performance →	Separate Jobs	One Job (In-Graph Loops)
Imperative Control Flow	Dryad/DryadLINQ [187], FlumeJava [45], Spark [189], SystemML/SystemDS [24, 25]	Mitos [72], Janus [92, 93], AutoGraph [126], Swift for TensorFlow
Functional Control Flow		Flink [65], Naiad [120], TensorFlow 1.x [186], MXNet

automatically handle low-level execution details, such as caching. SystemDS [24] evolved from SystemML, extending it with features for the end-to-end data science lifecycle.

Emma [9, 11, 12] is a metaprogramming-based Scala DSL (see Section 2.3 for an overview of DSL types) for scalable data analysis compiling to Flink and Spark. Emma has a holistic view of a user program, including its control flow. It achieves this via Scala's macro system [37]. Like SystemML, Emma takes care of low-level execution details, such as inserting appropriate cache calls in loops.

5.1.3.4.4 Compiling Imperative APIs to In-Graph Control Flow

Functional control flow APIs allow the system to execute a loop as a single dataflow job, enabling good performance. On the other hand, the imperative control flow APIs are more convenient to use, but older systems with imperative APIs typically execute loops as separate dataflow jobs per iteration step, which incurs a performance penalty. However, it is possible to combine the advantages of both of these API approaches: recent systems provide an imperative control flow API, but compile iterative programs into a single dataflow job. We can see an overview of these approaches in Table 5.4.

AutoGraph [126] and Janus [92, 93] are both Python DSLs for machine learning, compiling to TensorFlow. They allow for writing TensorFlow code in an imperative style with the standard Python control flow statements, which is convenient for users. They use metaprogramming to compile this code to a single TensorFlow dataflow job, which TensorFlow can then execute efficiently. Swift for TensorFlow[22] is a similar project aiming to compile from the Swift language to TensorFlow.

Emma [9, 11, 12] can compile the user's program into a single dataflow job only if the loop is compilable to Flink's in-graph loops, whose limitations we discussed in Section 5.1.3.3.1. Otherwise, it compiles to separate dataflow jobs per step.

Mitos can always compile to a single dataflow job. The structure of Mitos dataflows mirror SSA [148], which is a common intermediate representation of control flow in compilers of imperative languages. Because of relying on SSA end-to-end, it is easy to add any of the standard control flow constructs into Mitos (even unstructured ones, such as *goto*, *break*, *continue*, etc.): the only step for adding a new control flow construct is to compile it to SSA, from which point

[22] `https://github.com/tensorflow/swift/blob/f0d6c74ef5d016046afc1eac0b07a2f6b74b8fdf/docs/GraphProgramExtraction.md#adding-intraprocedural-within-a-function-control-flow`

Mitos can already handle it. Since SSA is quite common, translating the common control flow constructs to it is well-studied and straightforward.

Note that the above-mentioned AutoGraph [126], Janus [92, 93], and Swift for TensorFlow DSLs have to work with TensorFlow's control flow primitives as their compilation targets. Specifically, Janus compiles to TensorFlow's lower-level control flow primitives, where standard control flow operations have to be constructed from several primitive operations. AutoGraph and Swift compile to TensorFlow's higher-level control flow operations, where each of the standard control flow constructs is expressed as a single higher-order function call. These compilation approaches differ from Mitos' approach, where the target of the compilation is Mitos' own dataflows, whose representation of control flow is directly based on SSA. Mitos' approach is more general, as AutoGraph does not support *goto*. Also note that AutoGraph's compilation of *break* and *continue* involves introducing extra variables and conditionals, while Mitos can simply rely on the SSA translation of these constructs.

There are instances when a user wants to interleave non-system code (e.g., printing) and control flow (as discussed in Section 5.1.2.2.5). Systems with imperative APIs that execute loops as separate dataflow jobs trivially allow for running non-system code at every step. In these systems, non-system code runs in the driver program instead of being incorporated into dataflow jobs. If we instead compile imperative control flow to in-graph loops then the question arises, how to handle non-system code. To do so, we have to convert non-system code into system code, which is a hard problem due to the potential for external side effects (e.g., I/O): If an arbitrary interaction with the external world is allowed, the conversion might become invalid if it is not executed in the context of the driver program, but rather executed in one of the (cluster) worker machines. Unfortunately, current systems do not provide a general solution to this problem. Instead, they convert a select set of non-system code into system code. For example, AutoGraph handles an ordinary Python `print` statement by converting it to `tf.print`, which is TensorFlow's built-in printing function, and is thereby incorporated into the dataflow job. Janus [92] does an additional trick: it records global object mutation attempts, and defers them to the end of the job execution.

Myria [80] has an external DSL that includes a *while* loop construct. Myria compiles programs that use this construct to a cyclic dataflow and executes the loop as a single dataflow job. However, Myria does not allow for more general control flow, such as *if* statements or nested loops.

Fernandez et al. [68] compile imperative Java code to stateful distributed dataflow graphs. Their system has an imperative API not just in the control flow constructs, but also in working with data: Instead of using higher-order functions to transform immutable collections, the user writes imperative code to directly manipulate (annotated) data structures.

5.1.3.4.5 Trace-based Compilation

We turn our attention to a hybrid control flow handling approach, incorporated in TensorFlow and MXNet, which builds a graph from the trace of an imperative in-driver execution[23]. In TensorFlow 1.x, for example, let us assume that we apply the `tf.contrib.eager.defun` decorator to a Python function f, which has both tensor and non-tensor parameters. Each time f is called, TensorFlow will examine the non-tensor arguments to see whether the function was called before with these non-tensor arguments. If it is the first time for these arguments, the

[23] `https://www.tensorflow.org/api_docs/python/tf/function`

system will call the function in a special way called *tracing*. This entails calling the function with special placeholders for the tensor parameters and building a graph. This graph can then be used by TF for any values of the tensor parameters if the non-tensor arguments are the same.

The (somewhat complicated) consequences of the above mechanism on control flow handling is as follows. Assume that control flow is specified with imperative Python code. If the control flow depends on the non-tensor parameters of f, then TF will build just a single dataflow graph for each concrete value of these parameters. These graphs will not contain control flow, but instead loops will be unrolled, and only one branch of an *if* statement will be included in such a graph. Since the entire function is in a single dataflow job, this should be efficient, provided that the loop unrolling does not produce a very large graph. However, a drawback of tracing is that the control flow cannot depend on neither the tensor arguments, nor any values computed from tensors within the function, since this would require building the control flow into the dataflow job, which is not possible via tracing.

As of the time of this writing, the state-of-the-art execution mode in MXNet is tracing in the Gluon API[24]. However, in TensorFlow 2.0, the `defun` decorator uses AutoGraph by default (see Section 5.1.3.4.4) instead of tracing, which allows for control flow depending on tensor values, even when specifying control flow using Python's control flow primitives.

5.1.3.5 CIEL

CIEL [133] is a DDS with a directed acyclic graph of operators, similar to Spark or Flink. However, the crucial difference with regards to control flow is that operators in CIEL that are a part of a running dataflow job can extend the job by *spawning* new operators. This low-level mechanism enables the user to implement control flow constructs: an operator that evaluates an if-condition can spawn the dataflow fragment of the then-branch or the else-branch. Loops can be implemented in a manner akin to tail recursion: the dataflow fragment corresponding to the loop body either produces a final result or spawns itself to execute the subsequent step in the loop. Since this mechanism is too low-level, Murray et al. [131] created the Skywriting script language, which is more high-level. Musketeer [75] implements certain loops by similarly extending dataflow jobs. However, it performs these extensions at the level of its intermediate representation instead of the execution level, where it launches separate jobs.

The performance characteristics of the spawning approach in CIEL is similar to that of launching separate dataflow jobs for each step from a driver program: There is a per-step overhead associated with launching operators, and internal operator state cannot be shared between steps. However, the optimization of pipelining between iteration steps (which is otherwise typically possible only in systems with in-graph loops, see Section 5.1.4.4) is performed by CIEL.

5.1.3.6 Specialized Models

In this section, we breifly discuss computational models and systems that are specific to a particular data model, namely graph data and array data.

5.1.3.6.1 Graph Data

Many datasets can be represented as graphs and expressed as a collection of vertices and edges. Accordingly, several programming models and systems have arisen for the processing of graphs.

[24] `https://mxnet.incubator.apache.org/versions/1.8.0/api/python/docs/api/gluon/index.html`

In these models, graphs serve as input, and each vertex and edge can have an associated state of an arbitrary user-defined data type. This state can be modified during computation.

In the Pregel model [117], computation proceeds in a series of *supersteps* (inspired by the Bulk Synchronous Parallel model [177]). In each superstep, the system runs a user-defined *vertex function* in parallel (conceptually) for each active vertex of the graph. In one invocation of the vertex function, a single vertex receives all of the messages that were sent to it in the previous superstep, and can send messages to other vertices, which will receive them in the next superstep. Usually, messages are sent along graph edges, but they can also be sent to any vertex whose identifier is known.

In addition to receiving and sending messages, the vertex function can modify the state of the vertex and its outgoing edges, modify the graph structure, and *vote to halt* the program. If the vertex *votes to halt*, the vertex becomes *inactive*. In this case, the system will not run the vertex function for this vertex in subsequent supersteps. In the event the vertex later receives a message, it will become active once again. The program halts when all vertices are inactive, and there are no further messages to be delivered.

There are several variations on the above-mentioned "think like a vertex" idea, where a single invocation of a user-defined vertex function handles a single vertex and (possibly) its neighboring vertices. For example, GraphLab [116] provides a shared memory programming model, where a vertex function can directly read and write the states of its neighboring vertices, instead of sending and receiving messages. Gonzalez et al. [76] introduce the Gather-Apply-Scatter model, where the vertex function is split into three functions: the *gather* function aggregates information about the vertices and edges adjacent to the vertex, the *apply* function uses the aggregation result to update the vertex state, and the *scatter* function sends information out to neighboring vertices based on the updated vertex state.

Unlike Pregel, GraphLab supports microstep iteration (or asynchronous iteration, see Section 5.1.4.3), where a vertex function sees the results of the most recent updates, instead of results from the previous supersteps. Instead of the global synchronization barrier among all of the worker threads at the end of supersteps, GraphLab synchronizes only conflicting runs of the vertex update function, i.e., when affected vertex neighborhoods overlap.

Xie et al. [183] investigate the relative advantages of microstep and superstep models. In their study, they conclude that neither model outperforms the other in all circumstances. Therefore, they introduce Hsync, a hybrid graph computation system that dynamically switches between models automatically to optimize the performance.

GraphX [77] combines the benefits of a graph programming model and Spark, a general-purpose DDS. In GraphX, users can leverage both graph and general distributed collection operations in a single system. GraphX adds a number of optimizations on top of Spark, and thereby outperforms naive Spark implementations of graph algorithms. This way, GraphX achieves a performance that is comparable to specialized graph processing systems. Similarly to GraphX, Pregelix [33] has a graph programming model and executes programs on a DDS, Hyracks [29]. Gelly[25] executes graph programs on Flink.

For more detailed information about graph-based programming models and systems, have a look at these surveys: [16, 81, 95, 119, 185].

[25] `https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/libs/gelly/`

Table 5.5: Iteration support in streaming dataflow systems.

System(s)	Cyclic Dataflows for Streaming Data	Incremental Changes Across Both Streaming Epochs and Iteration Steps	Iterations Inside Micro-Batches
Flink	Yes	No	N/A
Naiad/Timely Dataflow	Yes	Yes	N/A
Spark	No	No	Yes

5.1.3.6.2 Array Data

SciDB [31] is a data management system for large-scale n-dimensional array data. Array data have inherent locality properties: data elements close in an array typically represent real-world objects also close together. For example, a 2-dimensional array can be the pixels of an image. A 3-dimensional array can be a series of images of the same location from different times. In these examples, close array elements represent close pixels, either in space or time.

Soroush et al. [170] extend SciDB with support for efficient iterative processing. Their approach introduces a programming model that is well-suited to exploiting the above-mentioned locality properties of array data. The programming model can be thought of as a constrained form of the Gather-Apply-Scatter model, with no scatter phase, and with the graph structure corresponding to the locality properties of the processed array. More specifically, the gather phase gathers data from a window in the array, such as a 3-by-3 block of pixels (instead of from the neighborhood of a vertex in an arbitrary graph). We discuss optimizations enabled by this model in Section 5.1.4.6.

5.1.3.7 Iterations in Streaming

In other parts of the thesis, we discuss batch computations, where the input data is finite and available in its entirety. In streaming computations, the input data is not finite but arrives continuously over a long time while the computation is running. Results should also be produced continuously, using only the input data that has arrived so far. There are several DDS that are also optimized for streaming computations, such as Naiad [120,132] and Flink [41]. In this section, we discuss iteration handling in streaming computations. Table 5.5 provides an overview.

Flink. In its streaming API, Flink allows for adding arbitrary cycles into its streaming dataflow jobs, but only using a low-level API. It does not provide certain basic services for building an actual iteration, such as separating records belonging to different iteration steps. However, there are research efforts to add support for structured iteration[26] [38].

Naiad uses an incrementalization approach to handle both incremental iterations and streaming computation. In Naiad, continuously arriving input data are divided into *epochs*. In each epoch, a new batch of input data arrives, which changes the input collections, for which the system computes the necessary changes in the output. Naiad has an elaborate mechanism to simultaneously handle incremental changes between iteration steps and between streaming epochs. Timely Dataflow (see Section 5.1.3.3.2) supports streaming similarly.

Spark handles streaming computation using the so-called micro-batch approach, where the system divides input data into small batches as it arrives, and starts separate batch computations for each batch [190]. (This is in contrast to epochs in Naiad, which handle all epochs in the same

[26] `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=66853132`

Table 5.6: Loop optimizations in DDS.

System	Loop-Invariant Datasets			Loop-Pipelining	Incremental Loops
	In-Memory Caching	Repart.	Indexes		
Spark	Manual	Manual	Manual (IndexedRDD [50])	No	No
Flink	Auto	Auto	Auto (hash joins)	Only Unrolled	Yes
Naiad	Auto	Auto	Auto (Differential Dataflow)	Yes	Yes
Mitos	Auto	Auto	Auto (hash joins)	Yes	No
DryadLINQ	No	Manual	No	Only Unrolled	No
Twister	Manual	Manual	Manual	No	No
HaLoop	No	Auto	Auto	No	No

long-running dataflow job.) Since each batch is handled by a standard Spark batch program, we can use Spark's standard mechanisms for control flow (Section 5.1.3.4.2).

5.1.4 Optimizations

We turn our attention to the various optimizations to improve (control flow) efficiency in DDSs. In particular, we will discuss optimizations to: (i) reduce dataflow job launch overhead, (ii) eliminate redundancies involving loop-invariant data, (iii) loop pipelining, and (iv) efficiently handle incremental changes to distributed collections across iteration steps. Table 5.6 provides an overview of the varying loop optimizations in DDS.

5.1.4.1 Reducing Dataflow Job Launch Overhead

As previously discussed, launching a separate dataflow job for each iteration step is a simple approach to implement loops in DDS. However, this can incur a significant overhead due to potentially launching a large number of dataflow jobs (see Section 3.6.5). Thus, several methods to reduce the (dataflow) job launch time have emerged, which we will discuss next.

Starting system processes incurs an overhead, which is particularly significant in JVM-based systems due to JIT compilation [115]. To mitigate this problem, a commonly employed optimization is to simply reuse actively running worker processes between dataflow jobs.

Scheduling decisions can also be made faster. Sparrow [139] distributes the scheduling itself. Nimbus [118] caches scheduling decisions across iteration steps. By this, it achieves similarly low step overhead as in-graph control flow, while being more flexible in, e.g., adapting to load imbalances.

If the amount of computation to do in a dataflow job is small, it might be better to execute it in a single machine instead of distributing it [121]. Switching to a single-machine execution is beneficial any time when this way the execution completes faster than the overhead of a distributed execution. SystemML/SystemDS makes this decision by a heuristic based on memory usage estimates and on its decision for earlier operations [25]. ML4All [99] makes this decision based on whether the input has a single partition, while Rheem [3] employs cost-based optimization. A similar optimization is performed by DryadLINQ [187], which dynamically chooses how many parallel instances of an operator should be launched based on the input data size.

This reduces overhead because the overhead of launching an operator typically depends on the number of parallel instances [118], as our experiments in Section 3.6.5 also show.

5.1.4.2 Loop-Invariant Datasets

Loop-Invariant Datasets (LIDs) remain unchanged over the course of a loop. They provide several optimization opportunities because naive implementations often do redundant work with them.

Avoiding recomputation. As previously discussed, distributed collections in DDSs are typically ephemeral by default (i.e., they are not stored between dataflow jobs). If we launch new jobs at each iteration step, LIDs should be persisted. In Spark, this is done using the `cache` method. In the absence of such methods, LIDs can be written to a file before the loop and read at each iteration step, thereby avoiding the need to recompute LIDs.

Avoiding disk I/O. LIDs that are not too large can be stored in memory instead of disk. This can improve performance by eliminating disk I/O and deserialization. For example, Spark's `cache` method stores datasets in memory by default. However, in HaLoop [34, 35], data is assumed not to fit in memory, and is therefore cached to disk. Elgohary et al. [62] utilize compression to fit more cached data in memory, and perform linear algebra operations directly on the compressed data to also avoid decompression costs.

Avoiding repeated repartitioning. LIDs are often used in operations that require their inputs to be partitioned by key. For example, a standard way of performing an equi-join in DDSs is to partition both inputs according to the join key, and then perform a local join on each pair of partitions with matching keys. If LIDs are used in a join, then systems like Spark that are unaware of loops will repeatedly repartition in each iteration step. This can be avoided by manually inserting repartitioning before the loop. In SystemML/SystemDS, these repartitioning operations are inserted automatically, thereby achieving a substantial speedup over Spark [25]. Also note that for Alternating Least Squares (ALS), it is beneficial to keep two representations of the input matrix: both row and column partitioned.

Avoiding repeated network transfer. Even if LIDs are put into appropriate partitionings before a loop, cached partitions might not be in the machines where they are needed. In this case, they have to be transferred over the network in each iteration step. To avoid this, HaLoop tracks the locations of cached partitions, and schedules tasks accordingly [34, 35]. Besides HaLoop, DryadLINQ [187] also avoids repeated network transfers. In Spark, it is also possible to perform this optimization, but very cumbersome even for an expert user due to technical issues in Spark's locality awareness.

Avoiding index rebuilding. It may be desirable to index LIDs for certain operations, which can introduce redundancy in loops. For example, at the build side of a hash join, rebuilding the hash table in each iteration step is redundant. Caching such a hash index is not straightforward in Spark since the RDD interface allows for iterating over the elements of an RDD, but not accessing them in any other way, such as performing a lookup in a hash table. For a possible workaround in Spark, see *IndexedRDD* [50]. However, avoiding index rebuilding is easier for in-graph loops, where a loop is executed within a single dataflow job. In this case, the operator that performs the join is not restarted during each iteration step. Thus, it can keep its internal state between steps, including any indexes. Flink [65], Naiad [120, 132], and Mitos automatically perform this optimization.

Repeatedly sampling a dataset (as in, e.g., mini-batch gradient descent) also provides an optimization opportunity similar to the upfront hash table building for joins above. The standard Spark implementation of sampling scans the entire dataset, which is wasteful if only a small sample is needed [99]. Instead, random access to the elements of the dataset to be sampled opens the way for more efficient sampling algorithms, which do not scan the entire dataset. This is easy to achieve in in-graph systems: a sample operator could just keep each partition of the dataset in an array throughout all iteration steps, and sample from these arrays repeatedly. In Spark, a similar trick as *IndexedRDD* can be used instead[27].

Despite the above benefits of loop-invariant hoisting, it can also disastrously reduce performance in some cases.[28] Imagine that we have a matrix `X`, a column vector `v`, and we perform the following computation in a loop body: `t(X) %*% (X %*% v)`, where `%*%` denotes matrix multiplication, and `t(X)` denotes matrix transposition. If `X` is a loop-invariant matrix, then there are two opportunities for loop-invariant hoisting, but both of them hurts performance. If we hoist `t(X) %*% X`, then we get a much larger matrix than any of the matrices/vectors that occur in the original computation if `X` has few rows but many columns. If we hoist only `t(X)`, we prevent the applicability of fused operators for scan sharing. To avoid this danger, SystemDS does not apply loop-invariant hoisting by default.

5.1.4.3 Asynchronous Loops

In the research literature, the term *asynchronous iteration* is used for two different concepts: for microsteps (in the context of microsteps vs. supersteps), and for the pipelining optimization between supersteps. Let us treat each of these in sequence.

Many authors [46, 65, 76, 116, 132, 180, 183] use the term *asynchronous iteration* to mean *microsteps*. Microsteps [65] arise, for example, in parameter servers[29] [111, 168], the GraphLab programming model [116], and in the evaluation of Datalog queries [179]. In these cases, each microstep processes little data (e.g., a single training example, a single vertex and its neighbors). Typically, microsteps are executed concurrently, without a total ordering among them [109], and without global synchronization barriers across parallel tasks[30]. In contrast, supersteps [177] are synchronous iteration steps, where a substantial amount of data-parallel computation takes place in each iteration step (e.g., when all of the graph vertices simultaneously process incoming messages in Pregel [117]). Also note that supersteps occur sequentially (i.e., there is a total ordering among them) and subsequent steps are able to examine all of the results from earlier steps. In this thesis, we primarily focus on superstep programming models.

In contrast, several papers [63, 72, 193] use the term *asynchronous iteration* to mean loop pipelining, i.e., the removal of the full synchronization barrier between supersteps and thus pipelining supersteps. In other words, they optimize a superstep-iteration by overlapping the execution of supersteps. This optimization does not alter the semantics of the programming model, i.e., all user code sees exactly the same intermediate program states as without the

[27] `https://github.com/ggevay/rdd-sampling`

[28] This example was pointed out by Matthias Boehm in personal communication.

[29] Most of the dataflow systems that we discuss do not support the parameter server architecture. However, TensorFlow does provide support, by representing global state in its dataflow graphs: it includes special operations (and corresponding dataflow nodes) called *variables*, which manage mutable state that is readable and writable by other nodes.

[30] There are different consistency models that dictate how to handle conflicting microsteps [116]. In some models (e.g., eventual consistency), a microstep might see stale data.

optimization. We will discuss this optimization in the next section.

5.1.4.4 Loop Pipelining

Loop pipelining, i.e., overlapping the execution of successive iteration steps, is a natural optimization approach that offers several performance advantages. For example, the pipelining of data transfers between operators in successive iteration steps eliminates the need to store intermediate results between steps, which saves on memory usage or reduces disk I/O and possibly eliminates (de)serialization. An additional performance benefit of pipelining comes from mitigating stragglers at the end of iteration steps. This is because pipelining allows the system to already start the next iteration step when a straggler is still working on the current step.

Execution approaches that launch separate dataflow jobs for each iteration step do not fit well with this optimization. For example, Spark typically cannot perform pipelining across iteration steps: Since actions block the execution of the driver program, the next step cannot start while the dataflow job of the current step is running.

In contrast, in-graph loops are naturally suited to pipelining. Examples of DDS that perform this optimization include iHadoop [63], Naiad/Timely Dataflow [120,132], TensorFlow [186], and Mitos. Flink, however, does not perform this optimization despite supporting in-graph loops, but this is just an implementation limitation. Elnikety et al. [63] report a 25% performance improvement in iHadoop due to pipelining. Other systems show even larger performance improvements: an up to 4x improvement in Mitos (Section 3.6.6.2) and a 2.5x–5x improvement in TensorFlow [186].

iHadoop [63] performs an additional trick on top of pipelining: it starts a subsequent step already before it evaluates the loop termination condition (akin to speculative execution in modern CPUs with branch prediction). This can provide a speedup because the termination condition evaluates to false in the majority of iteration steps, and the speculative computation is wasted only in the last step. This additional optimization could likely be incorporated into other systems, such as Mitos, as explained in Section 3.5.3.

Despite the benefits of pipelining, there is a caveat: the unlimited pipelining of iteration steps can cause excessive memory consumption. This occurs when operators require a significant amount of memory, and many instances belonging to different steps are started in parallel. To mitigate this problem, TensorFlow enables users to set the maximum number of iteration steps that the system will start in parallel.

5.1.4.5 Incremental Loops

Distributed collections in DDS, such as RDDs in Spark are typically read-only[31]. Consequently, even if we just want to make a small change to a collection in an iteration step, we still have to process the entire collection, including copying unchanged elements. For certain iterative algorithms, this can be a major source of inefficiency.

Mitos, similarly to Naiad/Timely Dataflow [120, 132], REX [124], and Flink [65], efficiently handle incremental changes in loops by keeping the states of long-running operators across iteration steps (see Section 3.5.2). Bear in mind that this requires implementing all of the iteration

[31]Distributed collections are read-only primarily because of fault tolerance: the original collection has to be available, in case we need to recreate another collection that is derived from it. However, see [50] for a workaround in Spark.

steps as a single dataflow job. Otherwise, we would lose internal operator states between iteration steps. More specifically, these systems keep collections in a data structure that allows for pointwise access, such as a hash table. With this, the system can efficiently modify any part of the collection without reprocessing the entire collection. SciDB [170] is an array database, which pushes incremental loops into the storage manager. Mitos differs from all the above systems by having a dedicated collection type for datasets that change incrementally.

5.1.4.6 Exploiting Locality Properties of Array Data

As discussed in Section 5.1.3.6.2, SciDB has a specialized model for processing large-scale array data iteratively. In this model, computing an update for an array cell needs information from nearby cells. Here we show two optimizations enabled by this locality property from Soroush et al. [170].

Mini-iteration processing. SciDB partitions arrays into chunks to process them in parallel. Furthermore, the chunks have a small overlap at their edges [31]. Soroush et al. [170] found that during iterative processing it is enough to synchronize the replicas of cells in the overlapping regions only at every few iteration steps. This cuts down on the communication cost, while it does not significantly increase the required number of steps to convergence.

Multi-resolution optimization. An array in SciDB is often the discretized version of a continuous measurement. In such a scenario, it is meaningful to analyze an array at multiple resolutions. Therefore, an optimization of SciDB is to first execute iterative computations on lower-resolution versions of an original input array, and then use the result as a starting point for the full-resolution computation. The system computes the lower-resolution versions from the original version in a user-defined way. Note that the user has to pay attention that the output of the iterative computation on the lower-resolution version is a valid intermediate step for the iterative computation on the original array.

5.2 Nested Parallelism

Flattening (unnesting) is a classic technique for handling nested parallelism [20, 66, 67, 78, 84, 166, 167, 175, 176], especially in the field of compilers [21, 44, 144, 165]. The idea originates from Blelloch, with the NESL nested-parallel vector language [20, 21, 22].

Flattening for Distributed Dataflow Systems. TraNCE [166,167], MRQL [66] and DIQL [67] are the closest works to ours: these systems flatten nested-parallel queries and translate them to distributed 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 run-time optimizations, which is crucial for achieving true flexibility in input data characteristics as shown by our experiments in Section 4.8.7. Our two-phase flattening process enables Matryoshka to perform run-time optimizations.

Compiling from Haskell. There are several works focusing on compiling from Haskell by utilizing flattening. For instance, Nepal [44] and Data Parallel Haskell (DPH) [144] extend Haskell with a data-parallel array type, which can be nested, and compile to a specialized multi-core backend, written in Haskell. Ulrich et al. [175, 176] compile similarly from Haskell, but to existing external relational backends. Still, Haskell is a purely functional language and therefore

these works do not support imperative control flow either. Note that these works consider ordered collections, in contrast to our unordered bags.

Other Functional Languages. Henriksen et al. [84] compile from Futhark, a purely functional array language allowing for nesting, to parallel GPU code. They focus on dynamically choosing which levels to parallelize, by employing a two-phase flattening process. However, their work cannot be directly applied to dataflow engines as they require different abstractions and optimizations. Slesarenko [165] performs flattening of arbitrarily complex types (e.g., more than two levels), but he concentrates on the language level, and does not provide details of the backend translation. His advanced datatype-generic programming techniques could complement our work by greatly reducing our boilerplate code for handling complex nested types.

SystemML/SystemDS parallel for. As discussed before in Section 4.8.5, SystemML/SystemDS [24, 25, 74] has a *parallel for* construct [26], by which users can express task parallelism on top of the data parallelism of linear algebra operators. SystemDS' optimizer chooses between different parallel execution plans, which vary in how they execute each level: sequentially, or parallel on multiple cores of one machine, or in a MapReduce/Spark job. However, as SystemDS does not employ flattening, it can run into the problems of the inner- or outer-parallel workarounds.

Manually launching parallel operations from inside other parallel operations. Ray/RLlib [113, 129] allows for nested parallelism but does not employ flattening. Also, it has a different programming model from our parallel collection operations: Its unit of physical parallelization (task) is directly exposed to the users: Task creation is in one-to-one correspondence with certain method calls in the user code (`f.remote`). This is in contrast to our programming model, where multiple task creations are hidden behind the abstraction of parallel collection operations: When the user calls a parallel collection method (e.g., `map`), then our system translates that to multiple task creations (i.e., multiple RDD partitions are created). By exposing physical parallelization in its API, Ray requires users to carefully control parallelization at each nesting level: they have to avoid under- or over-parallelization, i.e., dividing the work to too few or too many parallel tasks. Katsogridakis et al. [100] extended Spark to launch Spark jobs from inside Spark jobs. However, the multiple levels of parallelism are not flattened, and thus their system runs into the problem of the inner-parallel workaround: launching too many Spark jobs.

Other works. Junghanns et al. [95, 96] mentions supporting collections of graphs, but they then say that applying an operation on all graphs in a collection of graphs (i.e., `map`) is work in progress, and do not go into details. Pig [136] automates the outer-parallel workaround: Its language allows the user to use the same collection types and collection operations at inner nesting level as outside, but it translates to parallel operations only at the outer level. Therefore, it inherits the performance problems of the outer-parallel workaround. Emma [9, 11, 12] removes certain simple kinds of nesting, e.g., it employs *fold-group fusion* to flatten the situation when a grouping is followed by a `map` whose UDF aggregates each group. In other cases it falls back to the outer-parallel workaround.

Skew handling. Lastly, data skew handling in large-scale data processing is a well-studied problem [19, 108, 172], 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 handling technique in dataflow engines, e.g., Hurricane's task cloning can mitigate skew issues in joins or grouped aggregations [19]. Smith et al. [166, 167] handle skew by separating a bag into light and heavy keys, and executing

different operator implementations for the two cases.

Chapter 6

Conclusion

In this final chapter, we summarize the contributions of the thesis, and provide an outlook on potential future work.

6.1 Mitos

Modern data analytics often requires complex control flow, which is not well-supported in current distributed dataflow systems: they either suffer from poor performance or are hard to use. We presented the Mitos data analytics system, which uses the standard, easy-to-use, imperative constructs for writing programs with control flow. To execute these programs efficiently, it translates such a program to a single dataflow job.

Mitos uses an intermediate representation based on static single assignment form, abstracting away from specific control flow constructs. This intermediate representation facilitates both the dataflow job building and the coordination of the distributed execution of control flow statements. In particular, we devised a meta-programming-based approach for building a single dataflow job of a distributed dataflow system from a program with imperative control flow statements. Also, our mechanism for coordinating the distributed execution of control flow statements enables important optimizations, such as loop pipelining and loop-invariant hoisting.

The experimental evaluation shows that Mitos outperforms Spark by up to 45× thanks to compiling iterative programs to a single dataflow job, which eliminates the job launch overhead from iteration steps and enables loop optimizations. Interestingly, the results also show that Mitos outperforms Flink, which also compiles iterative programs to a single dataflow job, by up to a factor of 10.5× while also being easier to use. This speedup is due to Mitos having less per-step overhead and having the loop pipelining optimization.

6.2 Matryoshka

Although modern data analytics tasks could benefit from 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 program as input and creates an equivalent flat program, which can be executed efficiently on an existing dataflow engine. Thus,

Matryoshka frees users from the burden of implementing and choosing between workarounds, and instead lets users compose complex analytics programs in straightforward ways.

We experimentally evaluated Matryoshka using five common data analytics tasks. We found that Matryoshka is up to two orders of magnitude faster than baselines (the DIQL system as well as the outer- and inner-parallel workarounds) and also scales better when increasing the cluster size. Importantly, it provides uniform performance across varying data characteristics, e.g., (skewed) inner collection- or inner computation sizes. We also found that Matryoshka can flatten programs that DIQL is not able to flatten.

Crucially, Matryoshka can flatten programs that have loops at inner nesting levels. As discussed throughout the thesis, loops are important in modern data analytics, and thus loop support at inner nesting levels significantly extends the class of programs that we support.

6.3 Survey of Control Flow Handling in Dataflow Systems

In Section 5.1, we surveyed the research literature on how different dataflow systems support iterations and other control flow. Incorporating iterations in distributed dataflow systems proved to be surprisingly challenging. This is mainly because concentrating on just the performance aspects introduces a new problem: it complicates execution engines and causes many low-level execution details to leak through to the user-facing APIs. Consequently, users are required to become experts on system internals, which limits a system's potential user base. Thus, in recent years there is ongoing research that focuses on simultaneously addressing both performance and ease-of-use (e.g., Mitos).

In our survey, we provided an overview of the complex design space that is opened up due to the tension between the performance and ease-of-use requirements: We discussed not only performance optimizations, but also the usability issues of the various programming models with respect to iteration and other control flow constructs.

6.4 Future Research

In this section, we discuss potential future work.

6.4.1 Speculative Execution in Mitos

As discussed in Section 3.5.3, we could add a form of speculative execution to Mitos. Currently the system has to wait for the loop exit condition to be evaluated before starting the next iteration step. However, the system could start to speculatively execute the next iteration step already before the exit condition is evaluated (similarly to modern CPUs). This would enable loop pipelining in cases where the evaluation of the loop exit condition currently prevents loop pipelining.

6.4.2 Unifying Mitos and Matryoshka into a Single System

Due to historical reasons, Mitos is implemented on Flink, while Matryoshka is implemented on Spark. Thus, combining them would involve changing Matryoshka's backend. (It would not be possible to change Mitos' backend to Spark, since Mitos requires cyclic dataflows.)

With changing Matryoshka's backend, a simple way to combine the two systems would be to just use Mitos' execution layer (see Section 3.4) to execute the lifted control flow that Matryoshka produces (see Section 4.5). However, this would not make the generality of Mitos available at inner nesting levels: A separate lifting procedure would have to be devised for each control flow construct that we would like to support at inner nesting levels. A more complete way of combining the two systems would be to devise a lifting procedure for Mitos' control flow handling, lifting it as one (complex) control flow construct. This way, we would support any general imperative control flow at inner nesting levels.

Lifting Mitos' control flow handling is challenging for two reasons. First, when lifting a control flow construct, we have to make sure that it never happens that the lifted version executes a lifted bag operator exponentially more times than the maximum control flow path length among all the original UDF invocations. This would be a performance problem, because invoking an operator has an overhead (see Section 3.6.5). For example, the lifting of while loops described in Section 4.5 does not have the exponential explosion problem: the lifted version of a while loop will execute lifted bag operators in the loop body simply as many times as the largest number of iteration steps among all the original UDF invocations. However, a naive lifting of Mitos' general control flow handling could have the exponential explosion problem: Imagine a naive lifting, where there is a one-to-one correspondence between bag identifiers (in the Mitos sense, see Section 3.4.2.1) and lifted operator invocations. This naive lifted version of Mitos' control flow handling would execute a lifted bag operator as many times as the number of different execution paths reaching this operator among all the original UDF invocations. Now the problem is that the number of different execution paths can be exponential in the number of control flow steps. For example, if there is a while loop with an if statement in its body, then the number of different execution paths can be exponential in the number of iteration steps, since each step of the loop might execute a different if branch.

To avoid the above-mentioned exponential explosion of the number of lifted bag operator invocations when lifting Mitos' control flow handling, we would have to find a partitioning of all possible execution paths into a "small" number of equivalence classes, and call lifted operators only once per equivalence class. For this "small" number, we might hope to achieve $O(lb)$ classes for each execution path length l, where b is the number of basic blocks in the source code.

Another challenge in lifting Mitos' control flow handling is that we have to make it scalable in the total number of control flow steps across all UDF invocations. For example, in the current implementation control flow decisions are broadcasted, which would be a performance bottleneck when all UDF invocations are making control flow decisions.

6.4.3 Nested Parallelism in Stream Processing

We will now discuss how nested collections could model a windowed stream, a central concept in stream processing, and how this would enhance the compositionality of a stream processing system.

Both Mitos and Matryoshka are batch processing systems, i.e., they operate on bounded datasets. In contrast, stream processing systems operate on unbounded datasets, i.e., streams of data arriving continuously over time. To process such continuously arriving data, streaming programs often group the elements of a stream into finite chunks, called windows. (This grouping can be based on event timestamps, sessions, etc. Note that windows can overlap with each other.)

A window is processed by non-parallel code in current stream processing systems, such as

Flink. This can remind us of one of the batch processing scenarios that we used as a motivation for Matryoshka: In earlier batch processing systems, grouping operations emit each group as a non-parallel collection, which can thus be processed only by non-parallel code. Matryoshka enables treating groups as inner parallel collections. A future work could be to transfer Matryoshka's ideas to stream processing, where it could enable treating stream windows as parallel collections. This would enhance the compositionality between batch- and steam processing programs, as one could then apply a parallel batch program on each stream window.

A possible approach to transferring Matryoshka's ideas to stream procesing is to rely on incrementalization, as in Naiad/Differential Dataflow [132]: Input streams continuously change the inputs of a program that is expressed in the same way as a batch program, and the system continuously computes the appropriate changes to the output, and emits these changes as an output stream. Note that in a streaming scenario, it would be important to compile the entire program into a single dataflow job, and thus Mitos' ideas are also needed if we want to support control flow at the inner nesting levels of a streaming program.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] 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.

[3] 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.

[4] 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.

[5] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition*. Pearson Addison Wesley, 2007.

[7] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119. ACM, 1979.

[8] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[9] A. Alexandrov. *Representations and Optimizations for Embedded Parallel Dataflow Languages*. PhD thesis, Technische Universität Berlin, 2019.

[10] 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.

[11] 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.

[12] 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.

[13] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. Formalizing a broader recursion coverage in SQL. In *International Symposium on Practical Aspects of Declarative Languages*, pages 93–108. Springer, 2013.

[14] 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.

[15] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Readings in Artificial Intelligence and Databases*, pages 376–430. Elsevier, 1989.

[16] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, A. Barnawi, S. Sakr, et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

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

[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[19] 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.

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

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

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

[23] C. Boden, A. Spina, T. Rabl, and V. Markl. Benchmarking data flow systems for scalable machine learning. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 1–10, 2017.

[24] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqi, and S. B. Wrede. SystemDS: A declarative machine learning system for the end-to-end data science lifecycle. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR 2020)*, 2020.

[25] 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.

[26] 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.

[27] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[28] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Cidr*, volume 5, pages 225–237. Citeseer, 2005.

[29] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1151–1162. IEEE, 2011.

[30] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2), 2012.

[31] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[32] Y. Bu, V. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling Datalog for machine learning on big data. *arXiv preprint arXiv:1203.0160*, 2012.

[33] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *arXiv preprint arXiv:1407.0455*, 2014.

[34] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[35] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The HaLoop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169–190, 2012.

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

[37] 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.

[38] P. Carbone. *Scalable and Reliable Data Stream Processing*. PhD thesis, KTH Royal Institute of Technology, 2018.

[39] P. Carbone, S. Ewen, Gy. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[40] P. Carbone, G. E. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi. Large-scale data stream processing systems. In *Handbook of Big Data Technologies*, pages 219–260. Springer, 2017.

[41] 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.

[42] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

[43] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. *ACM sigplan notices*, 45(10):835–847, 2010.

[44] 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.

[45] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[46] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

[47] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[48] S. Chlyah, N. Gesbert, P. Genevès, and N. Layaïda. On the optimization of iterative programming with distributed data collections. 2020.

[49] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proceedings of the VLDB Endowment*, 9(12), 2016.

[50] A. Dave. IndexedRDD. `https://github.com/amplab/spark-indexedrdd`, 2014. [Online; accessed 30-April-2017].

[51] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[52] J. M. Decker, D. Moldovan, G. Wei, V. Bhardwaj, G. Essertel, F. Wang, A. B. Wiltschko, and T. Rompf. The 800 Pound Python in the Machine Learning Room. `https://www.cs.purdue.edu/homes/rompf/papers/decker-preprint201811.pdf`, 2018. [Online; accessed 2-Nov-2020].

[53] 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.

[54] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl. Optimistic recovery for iterative dataflows in action. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1439–1443. ACM, 2015.

[55] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL away. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR 2020)*, 2020.

[56] A. Eisenberg. New standard for stored procedures in SQL. *ACM SIGMOD Record*, 25(4):81–88, 1996.

[57] A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL3. *ACM Sigmod record*, 28(1):131–138, 1999.

[58] J. Ekanayake. Architecture and performance of runtime environments for data intensive scalable computing. *School of Informatics and Computing. Bloomington, Indiana University*, 2010.

[59] J. Ekanayake, T. Gunarathne, G. Fox, A. S. Balkir, C. Poulain, N. Araujo, and R. Barga. DryadLINQ for scientific analyses. In *2009 Fifth IEEE International Conference on e-Science*, pages 329–336. IEEE, 2009.

[60] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM, 2010.

[61] J. Ekanayake, S. Pallickara, and G. Fox. MapReduce for data intensive scientific analyses. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 277–284. IEEE, 2008.

[62] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment*, 9(12):960–971, 2016.

[63] E. Elnikety, T. Elsayed, and H. E. Ramadan. iHadoop: asynchronous iterations for MapReduce. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 81–90. IEEE, 2011.

[64] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative parallel data processing with Stratosphere: an inside look. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1053–1056. ACM, 2013.

[65] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11):1268–1279, 2012.

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

[67] 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.

[68] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 49–60, 2014.

[69] S. Feuerstein and B. Pribyl. *Oracle PL/SQL Programming*. O'Reilly Media, Inc., 2005.

[70] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[71] G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl. The power of nested parallelism in big data processing –hitting three flies with one slap–. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pages 605–618, 2021.

[72] 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.

[73] G. E. Gévay, J. Soto, and V. Markl. Handling iterations in distributed dataflow systems. *ACM Computing Surveys (CSUR)*, 54(9), 2021.

[74] 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.

[75] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[76] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation OSDI 12)*, pages 17–30, 2012.

[77] 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.

[78] T. Grust. Comprehending queries. In *Ausgezeichnete Informatikdissertationen 1999*. Springer, 2000.

[79] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. RaSQL: Greater power and performance for big data analytics with recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 467–484, 2019.

[80] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, and A. Whitaker. Demonstration of the Myria big data management service. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014.

[81] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[82] 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.

[83] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[84] 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.

[85] D. Hirn and T. Grust. PL/SQL without the PL. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2677–2680, 2020.

[86] D. Hirn and T. Grust. One with recursive is worth many gotos. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pages 723–735, 2021.

[87] M. Imran, G. E. Gévay, and V. Markl. Distributed graph analytics with Datalog queries in Flink. *LSGDA 2020 - International Workshop on Large Scale Graph Data Analytics*, 2020.

[88] M. Imran, G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl. Fast Datalog evaluation for batch and stream graph processing. *World Wide Web: Internet and Web Information Systems*, 2021.

[89] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

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

[91] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning. *ACM SIGMOD Record*, 49(1):43–50, 2020.

[92] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, and B.-G. Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, 2019.

[93] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, T. Kim, and B.-G. Chun. Speculative symbolic graph execution of imperative deep learning programs. *ACM SIGOPS Operating Systems Review*, 53(1):26–33, 2019.

[94] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[95] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm. Management and analysis of big graph data: current systems and open challenges. In *Handbook of Big Data Technologies*, pages 457–505. Springer, 2017.

[96] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm. Analyzing extended property graphs with Apache Flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, page 3. ACM, 2016.

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

[98] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge and information systems*, 27(2):303–325, 2011.

[99] Z. Kaoudi, J.-A. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal. A cost-based optimizer for gradient descent optimization. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 977–992, 2017.

[100] 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.

[101] 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].

[102] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 15–28, 2013.

[103] 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.

[104] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[105] 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.

[106] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.

[107] S. Kruse, Z. Kaoudi, B. Contreras-Rojas, S. Chawla, F. Naumann, and J.-A. Quiané-Ruiz. RHEEMix in the data jungle: a cost-based optimizer for cross-platform systems. *The VLDB Journal*, pages 1–24, 2020.

[108] 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.

[109] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[110] H. Lee, M. Kang, S.-B. Youn, J.-G. Lee, and Y. Kwon. An experimental comparison of iterative MapReduce frameworks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2089–2094. ACM, 2016.

[111] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

[112] Z. Li, Y. Fang, Q. Liu, J. Cheng, R. Cheng, and J. Lui. Walking in the cloud: parallel SimRank at scale. *Proceedings of the VLDB Endowment*, 9(1):24–35, 2015.

[113] 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.

[114] L. Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.

[115] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16)*, pages 383–400, 2016.

[116] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[117] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[118] O. Mashayekhi, H. Qu, C. Shah, and P. Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 513–526, 2017.

[119] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[120] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Composable incremental and iterative data-parallel computation with Naiad. *Microsoft Research*, 2012.

[121] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS*, volume 15, pages 14–14. Citeseer, 2015.

[122] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[123] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[124] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: recursive, delta-based data-centric computation. *Proceedings of the VLDB Endowment*, 5(11):1280–1291, 2012.

[125] D. Moldovan, J. Decker, F. Wang, A. Johnson, B. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. AutoGraph: Imperative-style coding with graph-based performance. In *SysML*, 2019.

[126] D. Moldovan, J. Decker, F. Wang, A. Johnson, B. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. AutoGraph: Imperative-style coding with graph-based performance. *Proceedings of Machine Learning and Systems*, 1:389–405, 2019.

[127] D. Moldovan, J. M. Decker, F. Wang, A. A. Johnson, B. K. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. AutoGraph: Imperative-style coding with graph-based performance. *arXiv preprint arXiv:1810.08061*, 2018.

[128] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-Virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 117–120, 2012.

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

[130] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[131] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. *HotCloud*, 10:12–12, 2010.

[132] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[133] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 113–126, 2011.

[134] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: quoted domain-specific languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 25–36. ACM, 2016.

[135] B. Nemeth, T. Haber, J. Liesenborgs, and W. Lamotte. Automatic parallelization of probabilistic models with varying load imbalance. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 752–759. IEEE, 2020.

[136] 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.

[137] S. M. Orzan. *On distributed verification and verified distribution*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[138] 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.

[139] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[140] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[141] S. Pallickara, H. Bulut, and G. Fox. Fault-tolerant reliable delivery of messages in distributed publish/subscribe systems. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, pages 19–19. IEEE, 2007.

[142] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL-and operator-centric data analytics in relational main-memory databases. In *EDBT*, pages 84–95, 2017.

[143] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. 2017.

[144] 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.

[145] G. Piatetsky. Largest Dataset Analyzed Poll shows surprising stability, more junior Data Scientists. `https://www.kdnuggets.com/2016/11/poll-results-largest-dataset-analyzed.html`, 2016. [accessed 14-Oct-2020].

[146] 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.

[147] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel. Recursive query facilities in relational databases: a survey. In *Database Theory and Application, Bio-Science and Bio-Technology*, pages 89–99. Springer, 2010.

[148] F. Rastello. *SSA-based Compiler Design*. Springer Publishing Company, Incorporated, 2016.

[149] T. Rohrmann, S. Schelter, T. Rabl, and V. Markl. Gilbert: Declarative sparse linear algebra on massively parallel dataflow systems. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, pages 269–288. Gesellschaft für Informatik, Bonn, 2017.

[150] L. Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2010.

[151] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012.

[152] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 127–136, 2010.

[153] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

[154] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[155] L. Ryzhyk and M. Budiu. Differential Datalog. *Datalog*, 2:4–5, 2019.

[156] S. Sakr, A. Liu, and A. G. Fayoumi. The family of MapReduce and large-scale data processing systems. *ACM Computing Surveys (CSUR)*, 46(1):11, 2013.

[157] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to Rome: optimistic recovery for distributed iterative data processing. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1919–1928. ACM, 2013.

[158] 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.

[159] J. Seo, S. Guo, and M. S. Lam. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289. IEEE, 2013.

[160] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 6. ACM, 2016.

[161] M. Shaw, P. Koutris, B. Howe, and D. Suciu. Optimizing large-scale semi-naïve datalog evaluation in Hadoop. In *International Datalog 2.0 Workshop*, pages 165–176. Springer, 2012.

[162] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta. M3R: increased performance for in-memory Hadoop jobs. *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.

[163] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with Datalog queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1135–1149. ACM, 2016.

[164] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

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

[166] J. Smith, M. Benedikt, B. Moore, and M. Nikolic. TraNCE: Transforming nested collections efficiently. *Proceedings of the VLDB Endowment (PVLDB)*, 2021.

[167] J. Smith, M. Benedikt, M. Nikolic, and A. Shaikhha. Scalable querying of nested data. *arXiv preprint arXiv:2011.06381*, 2020.

[168] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

[169] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[170] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient iterative processing in the SciDB parallel array engine. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 39. ACM, 2015.

[171] 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.

[172] 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.

[173] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[174] 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.

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

[176] 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.

[177] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[178] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *Proceedings of CIDR 2017*, 2017.

[179] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *Proceedings of the VLDB Endowment*, 8(12):1542–1553, 2015.

[180] Q. Wang, Y. Zhang, H. Wang, L. Geng, R. Lee, X. Zhang, and G. Yu. Automating incremental and asynchronous evaluation for recursive aggregate data processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2439–2454, 2020.

[181] H. Wu, J. Liu, T. Wang, D. Ye, J. Wei, and H. Zhong. Parallel materialization of Datalog programs with Spark for scalable reasoning. In *International Conference on Web Information Systems Engineering*, pages 363–379. Springer, 2016.

[182] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, Z.-H. Zhou, M. Steinbach, D. J. HAnd, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[183] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *ACM SIGPLAN Notices*, volume 50, pages 194–204. ACM, 2015.

[184] C. Xu, M. Holzemer, M. Kaul, J. Soto, and V. Markl. On fault tolerance for distributed iterative dataflow processing. *IEEE Transactions on Knowledge and Data Engineering*, 29(8):1709–1722, 2017.

[185] D. Yan, Y. Bu, Y. Tian, A. Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[186] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, M. Isard, M. Kudlur, R. Monga, D. Murray, and X. Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, page 18. ACM, 2018.

[187] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[188] 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.

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

[190] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, Berkeley, CA, USA, 2012. USENIX Association.

[191] Q. Zhang, A. Acharya, H. Chen, S. Arora, A. Chen, V. Liu, and B. T. Loo. Optimizing declarative graph queries at large scale. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 1411–1428, 2019.

[192] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 13. ACM, 2011.

[193] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, 2012.

[194] K. Zhao and J. X. Yu. All-in-one: Graph processing in RDBMSs revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1165–1180. ACM, 2017.

[195] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing*, 2010.

Acronyms

DDS Distributed Dataflow System

RDD Resilient Distributed Dataset

HDFS Hadoop Distributed File System

UDF User-Defined Function

DSL Domain-Specific Language

SSA Static Single Assignment form

IR Intermediate Representation

TC Transitive Closure

FL Fixpoint Loop

MR MapReduce

LID Loop-Invariant Dataset