ISSN 1000-9000(Print)
/1860-4749(Online)
CODEN JCTEEM

Vol.35 No.2 Mar. 2020

Journal of Computer Science & Technology

SPONSORED BY INSTITUTE OF COMPUTING TECHNOLOGY
THE CHINESE ACADEMY OF SCIENCES &
CHINA COMPUTER FEDERATION
SUPPORTED BY NSFC
CO-PUBLISHED BY SCIENCE PRESS &
SPRINGER

COMPUTER

Zhang YC, Wang XY, Wang C *et al.* Bigflow: A general optimization layer for distributed computing frameworks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(2): 453–467 Mar. 2020. DOI 10.1007/s11390-020-9702-3

Bigflow: A General Optimization Layer for Distributed Computing Frameworks

Yun-Cong Zhang[1], Xiao-Yang Wang[1,2,*], Cong Wang[1], Yao Xu[1], Jian-Wei Zhang[1], Xiao-Dong Lin[1]
Guang-Yu Sun[2], *Member*, *CCF*, *IEEE*, Gong-Lin Zheng[1], Shan-Hui Yin[1], Xian-Jin Ye[1], Li Li[1], Zhan Song[1], and Dong-Dong Miao[1]

[1] *Baidu Inc., Beijing 100193, China*

[2] *Center for Energy-Efficient Computing and Applications, Peking University, Beijing 100871, China*

E-mail: zhangyuncong@baidu.com; yaoer@pku.edu.cn; {bb7133, oclarms}@gmail.com; zjuzjw@163.com
linxiaodong@baidu.com; gsun@pku.edu.cn; scaugrated@gmail.com; yinshanhui_work@163.com
advancedxy@gmail.com; caterpillar_wing@163.com; csongzhan@126.com; himddfact@outlook.com

Received May 21, 2019; revised January 17, 2020.

Abstract As data volumes grow rapidly, distributed computations are widely employed in data-centers to provide cheap and efficient methods to process large-scale parallel datasets. Various computation models have been proposed to improve the abstraction of distributed datasets and hide the details of parallelism. However, most of them follow the single-layer partitioning method, which limits developers to express a multi-level partitioning operation succinctly. To overcome the problem, we present the NDD (Nested Distributed Dataset) data model. It is a more compact and expressive extension of Spark RDD (Resilient Distributed Dataset), in order to remove the burden on developers to manually write the logic for multi-level partitioning cases. Base on the NDD model, we develop an open-source framework called Bigflow, which serves as an optimization layer over computation engines from most widely used processing frameworks. With the help of Bigflow, some advanced optimization techniques, which may only be applied by experienced programmers manually, are enabled automatically in a distributed data processing job. Currently, Bigflow is processing about 3 PB data volumes daily in the data-centers of Baidu. According to customer experience, it can significantly save code length and improve performance over the intuitive programming style.

Keywords distributed computing, programming model, optimization technique

1 Introduction

Distributed computations over commodity servers provide cheap and efficient methods to process large-scale parallel datasets in modern data-centers. Recently, various distributed computing models and corresponding frameworks have been proposed to improve the abstraction of distributed datasets and hide the details of parallelization. For example, Google MapReduce [1] and its successors, such as Spark [2, 3] and Google FlumeJava [4], have been widely used for large-scale data analytic.

Most of these frameworks are built upon a single-layer partitioning model. Without loss of generality, we set the Spark system as an example to illustrate how the partitioning model works. Note that Hadoop and other platforms share a similar structure. Resilient Distributed Dataset (RDD) is the key abstraction in Spark, which is the minimum unit to be operated on with built-in functions. A collection is a single-node data split in an RDD. As shown in Fig.1(a), RDD on the left is turned into a new one with several collections after some functions (e.g., groupByKey, repartition). The resultant RDD is still treated as a single RDD afterwards, but we cannot apply built-in functions to these

Regular Paper
Special Section of ChinaSys 2019
This work is supported by the National Key Research and Development Project of China under Grant No. 2018YFB1003304 and Beijing Academy of Artificial Intelligence (BAAI).

*Corresponding Author

collections separately, which leads to considerable limitations as follows.

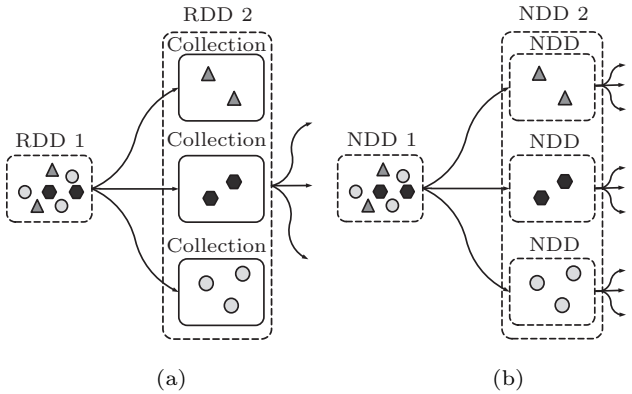


Fig.1. Comparison between RDD and NDD. (a) RDD-collection structure. (b) NDD-NDD structure.

Insufficient Function Re-Usability. As shown in Fig.1(a), the distributed dataset is shuffled into a set of single-node collections after a grouping operation in traditional models (e.g., the groupByKey method in Spark). However, these collections cannot be further processed with any existing functions, because those functions are designed for the entire distributed datasets. For instance, Naïve Bayes is a common classification method and has already been implemented for RDDs in some libraries, such as MLlib [5]. In some cases, however, the method may also be applied to each group of data (i.e., single-node data collection) rather than the entire RDD. It means that users have to rewrite the same method to apply the function to a single-node collection of data.

Difficulties in Optimizing User-Defined Functions. The user-defined functions applied to single-node collections are usually transparent to the framework. It means that these functions always work as black-boxes and it is hard to optimize their calculations by the framework. The framework fails to detect such type of processing patterns and consequently, cannot optimize the issue automatically. For example, Apache Spark① has to remind users that the utilization of the groupByKey transformation followed by an aggregation operation may be inefficient.

Explicit Management of Partitions. In traditional distributed computing frameworks, users have to specify how to cluster data towards partitions manually. For example, in secondary sort [6], users have to partition data with two features logically. While the first feature controls how to partition physically, partitioning on the second feature should be handled with user-defined functions manually. However, it is not always an easy task to implement an efficient partitioning strategy even for experienced developers. To this end, the development efficiency can be significantly improved, if we can provide a partition-free abstraction and complete these clustering optimizations inside the framework.

To solve these problems, we propose a new distributed dataset model called NDD, short for Nested Distributed Dataset. It is a more compact and expressive extension of the Spark RDD model. We also provide a proper method to transform the NDD models into equivalent jobs in traditional frameworks such as RDDs in Spark and MapReduce jobs in Hadoop. The method is named the scope tree abstraction, which enables the combination of the NDD model and traditional frameworks.

Based on the model, we have developed an optimization layer called Bigflow. It is compatible to various computation engines, providing automatic optimization techniques about performance, function reusability and coding complexity. We have demonstrated the system through some real-world workloads. The evaluation demonstrates that Bigflow achieves the comparable performance as the code manually optimized by experienced programmers. This work is open sourced under the Apache license 2.0②.

The rest of paper is organized as follows. Section 2 presents a review of the single-layer partition models in traditional frameworks. We describe the detailed concept of the NDD model in Section 3. Section 4 introduces the nested scope abstraction which is a practical way to implement the NDD model. Section 5 presents implementation details of the Bigflow framework. We present evaluation results of our system in Section 6. Section 7 summarize related work, followed by a conclusion in Section 8.

2 Background and Motivation

In order to present the details of the NDD model and Bigflow framework, we first provide some background on traditional cluster computing approaches.

Traditional distributed computing frameworks for general big data processing make it easy for developers to write distributed jobs. The cluster management,

①https://spark.apache.org, Jan. 2020.

②https://github.com/baidu/bigflow, Jan. 2020.

fault tolerance, and job partitioning are handled efficiently within the framework [1, 2]. To run computations on a large amount of data, distributed systems should split them across the cluster of nodes. The partitioning strategy in each computing step always has a significant impact on the performance. Most of those traditional methods still allow users to explicitly control the partitioning strategy to a certain degree. For example, the concept of partitioner exists in both Hadoop and Spark schemes [6, 7], which is the interface to partition data physically with respect to their keys. Wisely making use of repartitioning method helps avoid unnecessary shuffle operations and consequently jobs can speed up greatly, while on the contrast, performance suffers dramatically.

We use the Spark system as an example. The Resilient Distributed Dataset (RDD) [3] is the key abstraction in Spark. Users create RDDs and apply operations called transforms on them. When the data are key-value oriented, the partition strategy becomes significant because the entire dataset should be shuffled to prepare for subsequent transformations. If relevant keys are stored in the same partition, then those items are always in the same physical node and thus the unnecessary shuffling across the network is avoided. The concept of partitioner in the RDD model controls how RDD is partitioned by key. Transforms do not always preserve the partition information. For example, a union() operation on two key-value RDDs A and B discards the partitioner in resultant RDD if A and B do not share the same partitioner. To achieve a better performance, programmers should track the partitioning methods all the time. In fact, Hadoop has similar problems, for its dataset-partition structure is akin to RDD-collection structure in Spark.

In traditional approaches, partitioning operations always act as the single-layer model, where the dataset is shuffled into a set of single-node collections after a grouping transform. Those collections are parallel units of the framework, and user-defined functions are executed within each collection. Furthermore, the mapping strategy between logical partitions and physical nodes is transparent to users. Therefore, there is no guarantee which of those partitions are supposed to be located at the same node. In practice, there exist many tasks that require an explicit gathering strategy for those partitions. Consider the case in Fig.2:

1) several items are in different shapes and colors;

2) they are clustered according to their shapes;

3) the dataset is further divided by shapes and colors jointly, which means items are in the same partition only if their shapes and colors are both the same. In the single-layer partition model to the left, the subdividing is achieved with a repartition on a longer key combined with shapes and colors together;

4) items are collected according to their shapes.

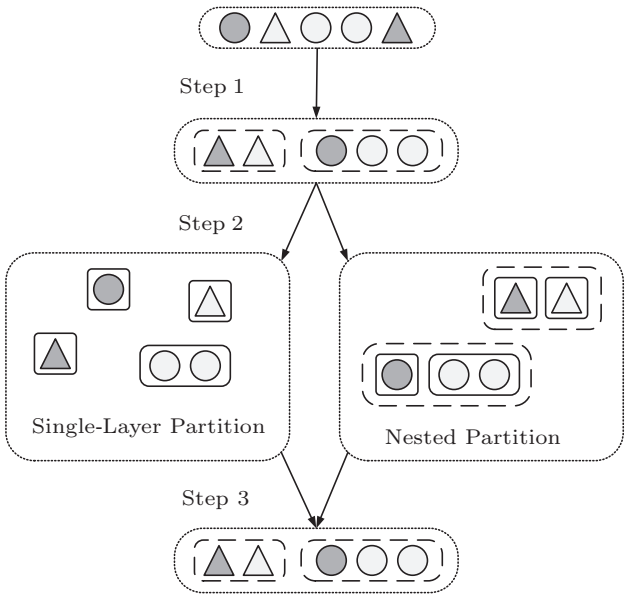


Fig. 2. Comparison between single-layer and nested partition models.

User-defined functions (UDFs) take place in each step. As they do not affect the shapes and colors of those items, they are not illustrated in Fig.2. Note that the single-layer partition on the left side is not efficient enough in step 3. It is difficult to coalesce back towards the former situation (grouped by shape only) without shuffling because the subordination information among those partitions has already been discarded in step 2.

In general, data in these tasks follow a tree structure, which is ignored in traditional systems. Take an RDD in Spark for example. The distributed dataset represents a collection of data partitioned across a cluster of machines. A partition in the RDD consists of data residing on one or more nodes, and operations imposed on the RDD are executed within each partition. Therefore, partitions are the basic units of parallelism in the RDD model. In other words, the concept of partitions in traditional execution models represents the granularity of the functions. Briefly, the whole logic can be

expressed as:

$$\texttt{Items}.\mathsf{groupBy}(shape).\mathsf{apply}(UDF)$$
$$.\mathsf{apply}(\mathsf{groupBy}(color).\mathsf{apply}(UDF))$$
$$.\mathsf{apply}(UDF)$$

However, single-level models such as RDD do not support a subdivision for each partition. To achieve the same performance as the nested structure, the $\mathsf{groupBy}(color).\mathsf{apply}(UDF)$ function is implemented manually and performs as a single black box for the framework, and the method UDF cannot be reused even if it shares the same logic. There are some other ways to achieve the same goal, but either a user-specified partitioner or an extra shuffling is needed.

A nested abstraction is able to solve these problems. Based on the observation, we propose the Nested Distributed Dataset (NDD) model. As illustrated in Fig.1(b), the NDD 1 turns into NDD 2 with several NDDs after some functions. In comparison with the RDD-collection structure, we can still apply built-in functions on those NDDs in NDD2. Furthermore, the computation framework will have more information about the partitions and thus automatically optimize those programs.

3 Nested Distributed Dataset (NDD) Model

In order to address those drawbacks in traditional models, we design the Nested Distributed Dataset (NDD) model, which is more compact and expressive than the Spark RDD. In this section, we first present the basic structure of the NDD model. Then, we provide an example of nested datasets to demonstrate the advantage of NDD over Spark RDD.

3.1 NDD Dataset Types

To provide the nested feature, the NDD model provides three core types of datasets listed in Table 1.

Table 1. Dataset Types in NDD Model

Data Type	Description
PCollection	A read-only, distributed dataset
PObject	A PCollection that has only one element
PTable	A grouped distributed dataset which consists of key-value pairs, the values of which are NDD datasets

PCollection is the basic dataset type in an NDD model. It represents a read-only distributed data collection. This concept is similar to the original Spark RDD datatype or the PCollection in FlumeJava infrastructure [4]. In the nested scope abstraction, the PCollection is the basic scope that connects with operators.

PObject is a special PCollection with only one element in it. It is always generated from an aggregating operation such as max and sum.

PTable is a (logic) collection of key-value pairs. For each element (i.e., key-value pair) of a PTable, the value can be a PCollection, a PObject, or a PTable itself. Obviously, the PTable enables the nested dataset in an NDD model. PTable normally derives from a grouping transformation operation, which is introduced in Subsection 3.2. For example, a PCollection consists of key-value pairs, represented as (K, V). A group_by_key method shuffles the PCollection into a PTable of (K, PCollection(V)), collecting all the values with the same key into the corresponding PCollection. In the nested scope abstraction, the PTable is the parent scope that keeps the subordination relationship.

3.2 NDD Transformations

Unlike a limited number of transformations in Spark RDD, transformations in NDD are more flexible. In an NDD model, any function that receives at least one NDD and returns a new NDD is defined as a valid NDD Transformation. To this end, the transformations of NDD are similar to those in Google's Dataflow model [8].

Table 2 lists some common transformations in an NDD model. The first column provides the usage of them. p, f and r represents the input NDD, the input function, and the result NDD respectively. Their types are shown in column two to four respectively.

Table 2. Typical NDD Transformations

Transformation	p	f	r
$r = p.\mathsf{map}(\mathrm{f})$	[V]	V → T	[T]
$r = p.\mathsf{filter}(\mathrm{f})$	[V]	V → bool	[V]
$r = p.\mathsf{reduce}(\mathrm{f})$	[V]	(V, V) → V	PObject(V)
$r = p.\mathsf{group\_by}(\mathrm{f})$	[V]	V → K	{K:[V]}
$r = p.\mathsf{flatten}()$	{K:[V]}		[(K,V)]
$r = p.\mathsf{count}()$	[V]		PObject(int)

Note that a bracket [V] indicates a PCollection with data of type V. Expression {K:V} means a set of key-value pairs that keys and values are of type K and type V respectively. Therefore, {K:[V]} represents a PTable data type.

As shown in Table 2, each NDD is able to receive some native transformations, such as map, filter and join, which are similar to those in Spark RDD. The reduce transformation generates a PObject data type. As mentioned before, the grouping operations (i.e., group_by) generate a PTable data type.

The apply method is also available for any NDD data type. It applies a certain transformation against the NDD and returns a new NDD. In addition, a dedicated method called apply_values is introduced for the PTable data type. It can apply the transformation to the value (i.e., a PCollection data type) in each key-value pair of a PTable individually and returns a new PTable. An example is listed as follows:

$$\texttt{book.group\_by}(\textit{lambda word} : \textit{word})$$
$$\texttt{.apply\_values}(\texttt{count})$$

The data book is a PCollection, which is a collection of multiple words (e.g., [a, b, a]). A group_by() operation transforms the PCollection into a PTable, whose elements are key-value pairs. As the word itself is used as the key, all same words in book are collected into a PCollection. As the result of the group_by() transformation, all the PCollections are organized with their own keys in a PTable (e.g., {a:[a, a], b:[b]}).

Then, the apply_values transformation takes effect. It takes a count() transformation as the input argument. According to its definition, the count() transformation is applied on every PCollection (e.g., [a, a] and [b]) in the PTable. It digests an NDD and returns a PObject with a single value, which represents the number of the elements of the NDD. Consequently, the piece of code above actually performs a word counting procedure for the original PCollection words (e.g., {a:2, b:1}).

Some of the transformations have a one-to-one relationship with the operators in the nested scope abstraction. Some others are responsible for moving towards another scope, such as group_by and flatten.

Besides, the NDD model also supports the method of side-input similar to FlumeJava. If processing on each record in an NDD requires the whole dataset from another NDD, the side-input method may be utilized to accomplish the command.

3.3 Case study

The NDD model is derived from Spark RDD; therefore they share a lot in common. One of the main differences between them is the organization of the dataset after the grouping operation, as introduced in Section 1.

In this subsection, we will take the Website UV Calculation as an example to explain several benefits introduced from the nested design. The algorithm aims at calculating the number of unique visitors towards each website according to the visiting log. Suppose that we have already implemented a function called count_distinct to calculate the distinct visitors towards a single website:

```
# Single-Website UV Calculation function
def count_distinct(users):
    return users.distinct().count()
```

It is easy to reuse the function in the multi-website circumstance, where the dataset is partitioned by website first and then the count_distinct function is applied to each partition:

```
# Multi-Website UV Calculation
logs.group_by_key()\
    .apply_values(count_distinct)
```

Note that count_distinct consists of several built-in transformations. Therefore, the execution platform knows more about its implementation details and is able to perform optimizations accordingly. On the contrary, we cannot reuse the count_distinct function in the Spark RDD model. The argument "users" is an RDD instead of a single-node list. We should reinvent the wheel to achieve the target:

```
# Spark version
def uv(users):
    return len(set(users))
logs.groupByKey()\
    .mapValues(lambda x: uv(x))
```

The internal design of the function uv cannot be observed by the execution platform, and thus most of the optimizations on it should be implemented manually.

4 Nested Scope Abstraction

Besides the nested dataset model, NDD further hides the partition management completely from users. We introduce the nested scope abstraction in this section, which is a practical way to implement the NDD model. Based on the abstraction, we try to strip the management completely from users to hide more details of parallelism. The impact on the performance of applications is presented at last.

4.1 Scope Tree

We introduce the scope tree abstraction in our work to help implement the NDD model. To avoid the ambiguity, we use the term "scope" to represent the logical concept of those partitions. It stems from the behavior of computation operations, which only access data within the boundary of each single scope. For example, a count operation executed on the global scope in Fig.2 calculates the amount of all items. In comparison, a count on the first layer scope calculates the number of items in each shape.

There are two main components in computation jobs: computation operations and logical scopes. In our abstraction, all those scopes and operators are organized in a tree. The transformations that relate to partition managements, such as group_by_key and flatten, are omitted in the scope tree. An example is shown in Fig.3. The leaf nodes are operators, and the non-leaf nodes are scopes. The edge between scopes denotes a subdivision and the parent scope of an operator represents its range of computation. Those dotted arrows do not belong to the edges of the scope tree. They indicate the direction of dataflows among those leaf nodes, and form a directed acyclic graph (DAG). The parent nodes of O2 and O4 are S1 and S4 respectively, and S1 is the grandparent of S4. Therefore, the data that flow from O2 to O4 actually experience two grouping transformations. Similarly, O5 and O6 have different parents, thereby a hidden flatten operation lies between them to recover the scope from S4 to S2.

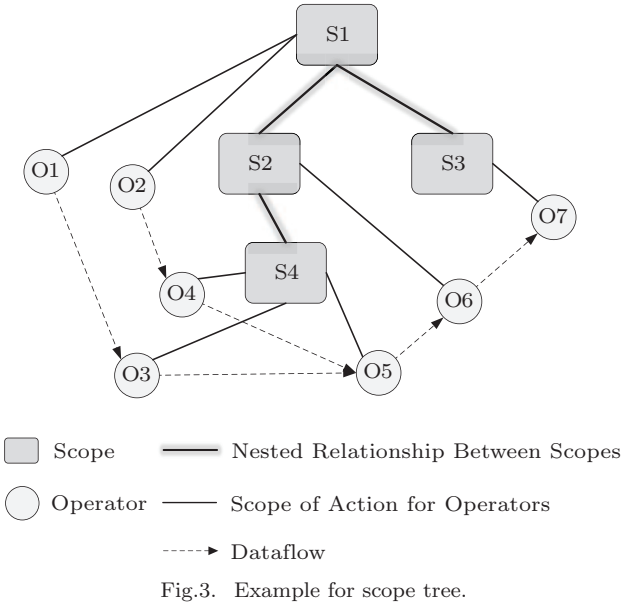


Fig.3. Example for scope tree.

The nested abstraction fits in with not only the secondary sort algorithm but some more general cases as well. There are no limits for them to perform a subdivision on existing scopes or attach operators on any scope and thus improve the function re-usability. Compared with the nested scope abstraction, operators in the single-layer partition model could only be applied to the entire dataset.

4.2 Hiding Partitions

Since the data processing job is normally deployed on a cluster of servers, its data partitioning strategy becomes a vital factor that affects the processing performance. To fine-tune computation jobs, programmers should always keep an eye on the current partition strategy. Apparently, it would be beneficial to have a programming model, which hides more partition details from users and optimizes the job automatically.

With the help of the nested abstraction, we are able to express the same scenario where logical partitions are gathered on physical nodes explicitly without considering the partitions. To simplify the abstraction, we further hide the partitioning completely from users.

It is well known that data shuffling across the network is always an expensive operation in distributed systems. However, the partitioning operation does not belong to the computation logic. Fine tuning on those partitioning strategies also results in obscure codes.

We use the Spark system as an example to introduce more details. The data within an RDD are split into several partitions, each of which guarantees its data to be on the same physical node. Some RDDs have partitioners that record how to match each data item towards its corresponding partition. Transformations assign partitioners for their output RDDs in two ways: setting user-specified partitioners explicitly, or preserving partitioners from input RDDs to avoid some unnecessary shuffles. But there are also transformations that cannot set partitioners for their output RDDs.

Many optimization techniques are based on the behavior of those partitioners. For example, if the dual input RDDs of a join operation are co-partitioned (i.e., they share the same partitioner), the corresponding partitions are matched together directly without any extra shuffles.

However, there are several drawbacks when the users are partition-aware. First, transformations behave differently on partitioner assignment, and thus users have to keep an eye on it all the time. The situation gets more sophisticated when user-specified functions are packaged in libraries acting as black boxes.

Second, fine-tuning Spark jobs with partitioners manually is always monotonous and short-sighted in a complex project.

Our target is to make our NDD model partition-free, in order to improve the efficiency in development and maintenance. Therefore, it should cover main usages of partitioners without much performance penalty:

1) control the concurrency according to the amount of data;

2) explicitly cluster specific keys into partitions to perform special tasks such as a secondary sort;

3) make use of the narrow dependency property to reduce network shuffling;

4) handle data-skew problems with user-specified partitioners.

The first case means a parameter denoting the number of concurrent threads at groupByKey-like operations is needed. The partitioners are necessary for traditional systems in the second case, and we have already solved the issue with the nested abstraction. The following two indicate that our abstraction should manage those kinds of optimizations automatically. Thus, it is necessary to carry out a performance analysis for our new abstraction.

4.3 Performance Discussion

In this subsection, we analyze that several effective optimizations results from the usage of partitioners could also be discovered automatically, as long as some basic information is provided. To simplify the discussion, we assume that the computation work remains the same when the partition strategy changes. Thus, the main reasons that influence the performance are data shuffling, data transmission and data skew.

4.3.1 Poor-Informed Data Distribution

Firstly, let us consider the most common case, where users have no idea about the distribution of keys. The consideration of partitioner preservation and the explicit usage of partition-aware techniques often aim at reducing the unnecessary shuffling and data transmission.

Simple optimization patterns for partition-aware techniques are easily recognized because the framework actually knows the same as users. One of the most important considerations is making use of the narrow dependency property in partition-aware and co-located operations [7]. When the input RDDs share the same partitioner, data shuffling could be avoided. As all the partitioners are determined by the framework instead of users, it gains more flexibility to adjust partitioners according to the context.

For example, we have a union operation following with a join: $A.\mathsf{union}(B).\mathsf{join}(C)$. union discards the output partitioner if two partitioners from A and B differ, and a shuffle occurs for the result of union to match the partitioner of C. But if A and C share the same partitioner, users may repartition B with the very partitioner and the union will preserve it. Compared with the original realization, the optimized version only shuffles dataset B and avoids the repeated partition operation against dataset A.

In our framework, it is possible to detect the pattern and carry out the same optimization. When viewed as a whole, the optimization actually propagates the partitioner *backward* along the transformation chain from C to A and B. In practice, partitioners can be spread further in more complex jobs, and thus those jobs reward more in reducing unnecessary shuffles. Although the greedy method does not provide an optimal plan for partitioner scheduling, it still works well for most general cases.

4.3.2 Well-Informed Data Distribution

Our abstraction reserves interfaces that control the number of concurrent threads explicitly as well as coalesce tiny partitions together. However, if users have knowledge about the distribution of keys, simply hiding the partition details still seems harmful. For example, a sampling job can reveal some data skew problems, and an elaborately designed partitioner may alleviate the predicament.

The ratio dictionary interface is introduced for partitioning transformations to solve the problem. When meeting with a data skew problem, the programmer notices the keys with the largest amount of items first. The ratios of abnormal keys are reported to the framework in a dictionary, and a partitioner will be set accordingly to divide the dataset evenly on physical nodes. Firstly, the abnormal keys are allocated via the first-fit decreasing algorithm. Then the other keys are collected proportionally to fill up the residual quota of each physical node. The more information the user provides in the dictionary, the fairer the partitioning will be.

Most of the usages of partitioners have already been covered in the abstraction. However, users can even imitate a partitioner, if they insist to do this, with a key extracting transformation along with the ratio directory method. The original partition indexes are extracted

as keys and reported to the framework from the ratio directory. As the number of keys is equal to that of physical nodes, there will just be one key per partition.

From the discussion above, we can conclude that normal occurrence of partitioners can be taken over by our nested scope abstraction.

In Section 1, we have discussed several limitations of single-layer approaches: insufficient function reusability, difficulties in optimizing user-defined functions, and explicit management of partitions. With the help of the nested abstraction, we are able to solve those problems naturally.

1) Different from the single-layer partition abstraction, the nested scope abstraction does not bound the logical partition towards a physical single-node data collection. Inappropriate for this situation, it seems like that a partitioning on an RDD results in a group of smaller RDDs. You may further partition them, flatten them back, or reuse any existing functions applied for an entire RDD.

2) Distributed data processing frameworks gain knowledge on how to parallelize jobs from the combination of its operators. A user-defined function passed to each single-node collection is actually a black box for the framework. In the nested scope abstraction, however, partitions can be treated as smaller distributed datasets, which are able to accept functions implemented with built-in operators in the framework. A detailed example is introduced in Subsection 3.3.

3) Explicit management of partitions in special tasks is represented as a multi-layer scope tree and is much easier to realize and understand.

5 Bigflow Implementation

The Bigflow framework implements the NDD model over some existing computation engines and enables several automatic optimization techniques. Traditionally, these optimization techniques can only be applied by an experienced developer in more complicated hand-optimization. In other words, using Bigflow can achieve hand-optimized performance using a more compact and efficient modular programming style. Its design details are introduced in this section.

5.1 Architecture Overview

An architecture overview of Bigflow is illustrated in Fig.4. A complete flow is composed of following stages.

• Stage 1. A user writes a Bigflow program using its programming interface.

• Stage 2. The Bigflow program is sent to the API layer. In this layer, the program is first translated into an API plan. Then, it is translated into a logical plan with proper optimization.

• Stage 3. The logical plan is sent to the core layer. In this layer, it is further optimized to generate a physical plan, which is engine-relevant and ready for processing.

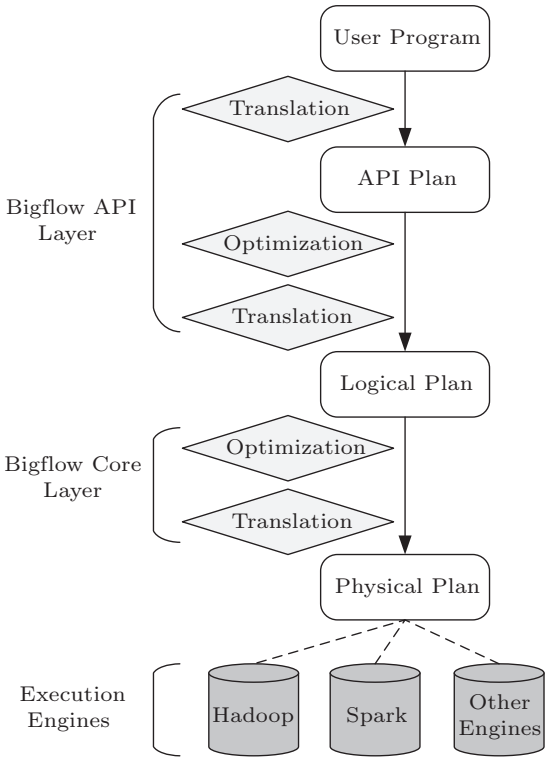


Fig.4. Bigflow architecture overview.

Detailed design of each stage is introduced in following subsections.

5.2 Bigflow Programming Interface

The Bigflow framework provides a Python interface for the programmers. Each NDD is represented as an object and each transformation as a function.

The concept of pipeline is introduced in the Bigflow system as a basic unit of computation. A user program is normally represented with one or multiple pipelines. A pipeline can be processed on any under-layer execution engine specified by the user. There are several operations for a pipeline: 1) to generate NDDs, 2) to write the NDDs onto the external storage, and 3) to control the jobs. Note that all transformations are deferred before they are triggered by either the `run` or `get` method. A simple example of Bigflow program is listed as follows to illustrate how it works.

```
pipeline = base.Pipeline.create('SPARK')
p1 = pipeline.parallelize([-1, 3, -7])
p2 = pipeline.parallelize(0)
result = p1.filter(lambda x, y: x < y, p2)
result.write('hdfs:///out_path').run()
```

First, the program establishes a pipeline over the Apache Spark engine. From the pipeline, two NDDs, p1 and p2, are generated with the method parallelize. p1 is generated based on a list and p2 is generated from an integer. Then a filter transformation filters out all the elements in p1 that are not smaller than the number in p2. Finally, a run action triggers the whole calculation procedure, and the result will be $[-1, -7]$.

5.3 Bigflow API Layer

The Bigflow API layer first transfers the input Bigflow program to an API plan. Then, the API plan is translated into a logical plan. Several optimizations are applied during translation.

5.3.1 API Plan

An API plan organizes the details of an input program as a directed acyclic graph (DAG).

Basically, every operation in the program is mapped to a node in the DAG. Edges between the nodes denote the direction of data flows. According to the character of each operation, some optimizations may be applied to the API plan. An optimization example is to merge the adjacent shuffles that share the same key. As the NDD abstraction hides the control of partitions from the programmer, the Bigflow framework gains more flexibility to trigger this type of optimizations. Having these optimizations, an API plan is translated into a logical plan.

5.3.2 Logical Plan

A logical plan consists of nodes, scopes, and edges. It implements the scope tree presented previously in Section 4.

A node denotes an operator on a subset of the dataset. All these nodes in the logical plan occupy all the leaf nodes in the plan tree.

A scope represents the execution scope and granularity for its node children. All the non-leaf nodes are scopes.

A edge indicates the direction of a dataflow. It has a "partial" feature, which means that the node on the target side does not need to gather all the input data from this direction before calculation. The partial feature is determined accordingly in three cases as follows.

First, the mapping-like operations, such as map, filter and flat_map, always accept partial edges. They process one item at a time, regardless of other preceding or subsequent items.

Second, the reducing-like operations, such as sum, reduce and aggregate, are supposed to accept half-partial ones. These operations aggregate the input set of data, which seems to be non-partial. However, the entity of the reduce functions is suggested to be commutative and associative. As a consequence, it is able to do some pre-aggregation work before all the data arrive.

Third, some other operations, such as transform and accumulate, have only non-partial edges.

5.3.3 Plan Translation

The translation from an API plan into the corresponding logical plan is intuitive. The core idea is building the scope tree and mount nodes along with their edges.

An API plan consists of operations for computation and data partition. The former ones correspond to nodes in the logical plan, and the latter ones help build the scope tree to indicate a scope shift for following operations. Directions between operations are reserved as edges to indicate the dataflow.

The root of scope tree is a default GlobalScope. A grouping operation such as group_by and cogroup generates a child for the current scope, and succeeding computation operators are mounted onto the new scope node. The flatten operation does not create new scope nodes but leads the following operators towards the parent scope. The translation goes on until all those operations in the API plan are mounted on the scope tree. At last, the original directions in the DAG are converted to edges with their own partial features.

There is information loss during the translation. The API plan knows the property of each operator well, but the logical plan just treats them in the same way. Thus those optimizations related to the operators should take place before this translation stage.

5.4 Bigflow Core Layer

The Bigflow core layer takes the logical plan as the input, and transfers it to a physical plan. For the Spark engine, partitioners for each RDD are automatically managed by the system. For the Hadoop engine, we mostly follow the same way introduced by FlumeJava [4]. Besides the engine-relevant translation,

some common optimization techniques are utilized to achieve better performance, which is mentioned in this subsection.

5.4.1 Partial Unit Promotion

According to the partial feature in the NDD model described in Subsection 5.3.2, some pre-aggregation may be performed on the mapper side, in order to reduce shuffle overhead.

A node is tagged as partial only in the case that all its input edges are partial ones. The node with only half-partial edges is split into a partial node and a non-partial one. For example, a half-partial count operation is converted to a partial count_p ProcessNode and a non-partial sum node. The rest nodes are tagged to be non-partial.

In the partial unit promoting phase, Bigflow moves the partial nodes with their parent scopes forward to the upstreaming task, in order to do some pre-aggregation work and thus reduce the amount of data to shuffle. The scope promoted with the partial node is tagged as partial, which starts shuffling before all data are ready. As the partial property guarantees that operators are commutative and associative, the scope is able to split a large amount of input data into pieces. It caches the data until the amount reaches a threshold. Then a shuffle operation is triggered to sort the cached data, pass the result onto the next node, and wait for another trigger event. The size of cache depends on the memory capacity of the computation node, thereby the risk of out-of-memory exception can be avoided efficiently.

The scope remaining in the downstream side also changes its default behavior. As the promoted scope has already accomplished shuffling within each batch, the downstream scope does not have to shuffle the data again. It just collects all the data under the same key into a group. When the underlying engine, Hadoop for instance, performs a sort along with each shuffling, the grouping work at the Reducer side benefits a lot because data are organized in order.

5.4.2 Partition Rearranging

The data skew problem is classical in the distributed computing field. When the dataset does not follow the uniform distribution, the workload will not be evenly partitioned among computing nodes. This kind of phenomena always leads to poor performance or even an out-of-memory exception.

The Bigflow framework provides some special optimization according to the NDD abstraction. For example, consider a multi-level partition case. Assume that the partition on the first key has a serious skew problem. But each PCollection in the result PTable will be further subdivided on another key afterward. The framework changes the single-key partition policy to a dual-key version. Partitions become more finely grained, and thus more likely to be evenly distributed among those computing nodes.

However, if the parent scope has some other nodes after the child scope, this kind of optimization will introduce extra data transmission overhead. Therefore, the optimization only occurs when the data skew is serious enough. In the current implementation, users have to inform the Bigflow framework of the serious skew on some input edges via ratio dictionary. We are trying some automatic tactics to solve the problem at runtime.

5.4.3 Example of Core Layer Process

We use the Website UV Calculation introduced in Subsection 3.3 as an example to illustrate the process of Core layer. To simplify the discussion, we abbreviate the name of this case as WebUV.

Fig.5 illustrates the whole process. Grey blocks stand for scopes, rounded rectangles represent operators, and arrows with solid line denote the direction of dataflow. Hence, the skeleton of a logical plan is actually a scope tree as introduced in Subsection 4.1. A dashed line box indicates the border of a single stage, where all computations are accomplished within each physical node. Those arrows with dashed lines indicate the changed parts through the optimization.

Fig.5(a) is the initial state of the logical plan. Notice that the input data have a default partition. It is often the case when the data are located in a distributed file system such as HDFS. We may make use of this property to perform some pre-aggregations on those nodes.

Operator "Take 1" comes from the distinct transformation. It is half-partial and therefore can be separated into a partial one and a non-partial one. Then partial unit promotion optimization takes place, lifting the partial Take 1 to the upstream stage. Fig.5(b) outperforms Fig.1(a) because the mapper side Take 1 reduces the data to be transferred across the network.

If a serious data skew problem occurs in Fig.5(b), several nodes will undertake most of the work and the entire job suffers from poor performance. Then the partition rearranging optimization is a good choice to al-

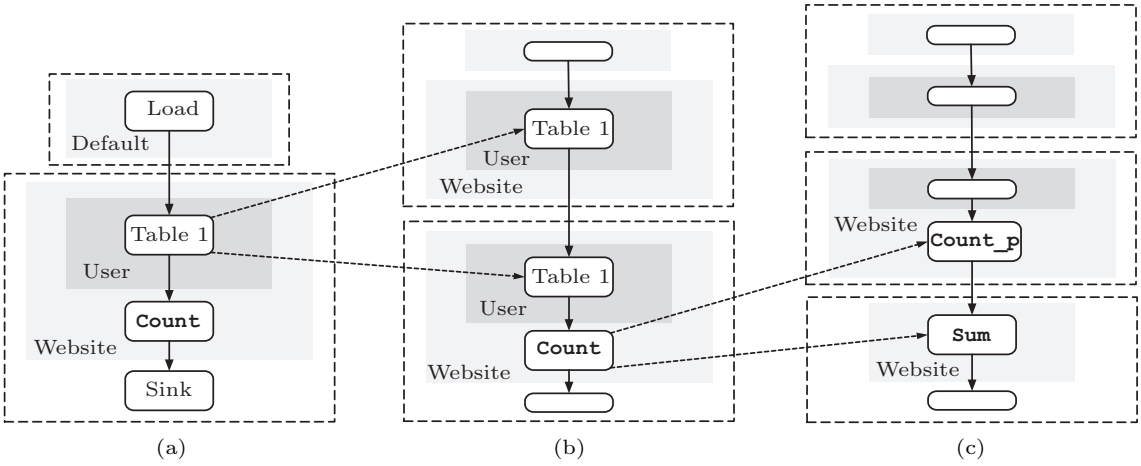


Fig.5. Core layer example. (a) Original plan. (b) After partial unit promotion. (c) After partition rearranging.

leviate the issue. As shown in Fig.5(c), the half-partial operation count is split into a partial count_p and a non-partial sum, and a new stage forms. Compared with the unbalanced distribution stemming from the first key, the second stage is more finely partitioned according to two keys jointly in the new logical plan. Although the data skew problem is lessened, extra data transmission is introduced. Hence, the optimization is triggered only if the user informs the framework of the serious unfairness.

6 Evaluation

In this section, we collect some feedback from the Bigflow users and measure the performance of the system on a large cluster of machines.

6.1 User Experience

The initial release of the Bigflow system was in October 2015 as an infrastructure for distributed computing jobs within Baidu. Since then, it has adopted many users to migrate their applications from the Hadoop and Spark platforms. The Bigflow system aims at improving the performance as well as reducing the difficulty of coding. According to the feedback and statistics from the users, those applications running on the Bigflow system benefit a lot from it. Currently, over 3 PB data are processed though Bigflow daily.

Fig.6 illustrates some feedback from developers of several active applications which are migrated from Hadoop and Spark platforms. It includes the performance improvement and the script size reduction. The feedback shows that Bigflow is able to reduce the code size so as to improve the execution performance a lot.

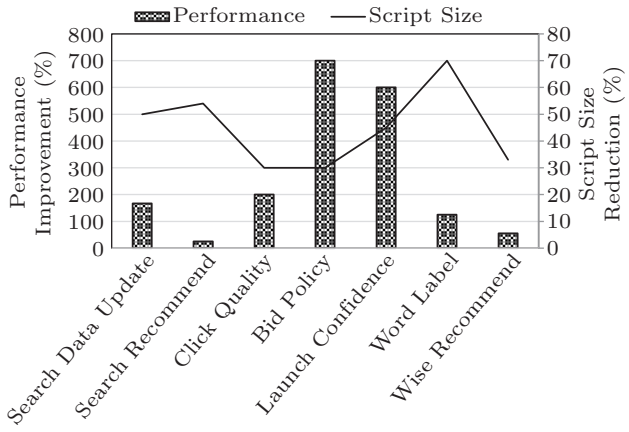


Fig.6. Bigflow user experience.

6.2 Evaluation Results

As an optimization layer, the Bigflow framework aims for two targets: improving data processing performance and reducing coding complexity at the same time.

6.2.1 *Experimental Setup*

Our experiments are evaluated on a cloud platform in the real data center system. The platform contains thousands of computing nodes, providing configuration space to limit the resource for each job. Physical machines are connected by 10 Gbps Ethernet.

We use several workloads to illustrate the advantages of the Bigflow system, as listed in Table 3. WebUV and WordCnt are famous benchmarks for evaluating different big data frameworks, while gTop$N$ and gBayes represent some common nested patterns in real-world jobs.

To evaluate the efficiency of the Bigflow system in

Table 3. List of Benchmarks

Name	Description	Dataset
WebUV	Website UV calculation: calculate the number of unique visitors towards multiple websites according to the visiting log	Real world dataset
WordCnt	Word count: count the word frequency for the English-language Wikipedia Database[3]	English-language Wikipedia database
gTop$N$	Grouping Top$N$: find the largest N integers in each group	Dataset generated according to the Pareto distribution
gBayes	Grouping Naïve Bayes: build a classification model for web page classification in each group	Real-world dataset grouped by languages

detail, we run these benchmarks over the Hadoop engine and Spark engine respectively. The configurations of these platforms are described in Table 4.

Table 4. Evaluation Platform Configuration

Platform	Number of Cores	RAM per Core (GB)
Hadoop 2.7	77 151	6
Spark 2.1	417	2

6.2.2 *Performance Comparison*

The Bigflow framework serves as an optimizing layer over Hadoop and Spark engines, carrying out several efficient optimizing techniques automatically. To measure the efficiency of the system, we compare the execution time of all benchmarks on the Spark engine and the Hadoop engine separately.

Fig.7 illustrates the comparison between Spark and Bigflow on Spark. PySpark is the official Python version for Spark, which also wraps Scala Spark as the underlying execution engine in the same way as Bigflow. The execution time is normalized according to Scala Spark. We can find that Bigflow outperforms PySpark in those cases by 30% on average.

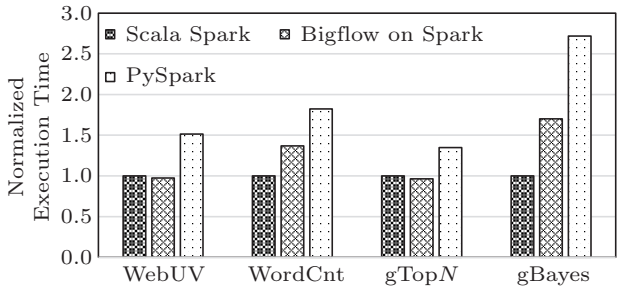


Fig.7. Normalized execution time based on Spark engine.

Note that Bigflow provides APIs in Python, which is a slower language compared with Scala. In cases such as WebUV and gTop$N$, Bigflow is even faster than Scala Spark in spite of the language overhead.

Fig.8 illustrates the comparison between Hadoop and Bigflow on Hadoop. The execution time is normalized according to Hadoop. The results show that, with the help of Bigflow framework, the execution time is reduced by 50% for Hadoop engine.

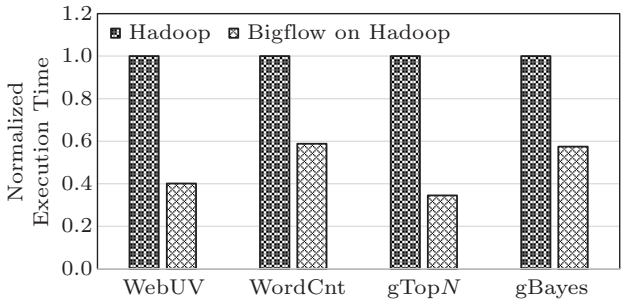


Fig.8. Normalized execution time based on Hadoop engine.

6.2.3 *Code Development Efficiency*

To compare the code development efficiency, we implement the WebUV case in multiple ways:

- intuitive versions using Hadoop/PySpark,
- hand-optimized versions using Hadoop/PySpark,
- intuitive versions using Bigflow on Hadoop/Spark engines.

Results are normalized based on the execution time of the first version with the same execution engine. Fig.9 compares the execution time as well as the code length.

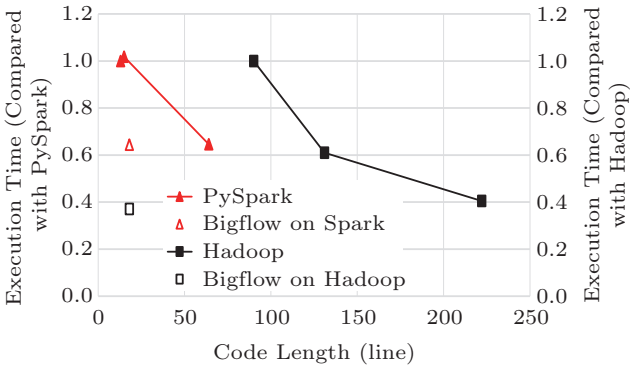


Fig.9. Comparison among different approaches.

[3]https://en.wikipedia.org/wiki/Wikipedia:Database_download, Jan. 2020.

According to the result, the size of Bigflow code is similar to that of the Spark version without optimizations, but reduces a lot in comparison with the hand-optimized version. Meanwhile, the Bigflow achieves the similar run-time performance as the optimized version. While optimizations can improve the performance greatly, they always take lots of efforts. Bigflow provides a reasonable way to improve the code development efficiency.

We can also tell from the result that the first optimization does not make a difference for Spark but works well with Hadoop. This kind of phenomena occurs all the time when tuning computation jobs, where the same optimization performs differently for different execution engines. Bigflow serves as a general optimization layer over them, and is able to bring about proper optimizations according to the underlying execution engine.

7 Related Work

In recent years, there has been an increasing amount of literature on distributed processing systems. Bigflow is inspired by several frameworks, which will be briefly described and compared in this section.

MapReduce [1] hides many low-level distributed implementations and allows users to simply create large-scale data processing systems by providing a high-level computation abstraction. The broad success of this paradigm motivates the development of various related systems. Apache Hadoop④ is one of the most popular adoptions of the MapReduce programming model. Dryad [9] introduces embedded user-defined functions in its general-purpose data-parallel execution engine, which inspires an open-source implementation called Apache Tez [10]. There are also other successors, such as Pig [11] and Hive [12]. Bigflow offers full support for Hadoop and its variants.

Traditional MapReduce-style batch systems work well for computations that can be broken down into a map step, a shuffle step, and a reduce step. However, one common drawback of those approaches is that they should keep temporal states on disks between jobs. Spark [2] is a data-processing framework that is designed to be fast for interactive queries and iterative algorithms by offering in-memory processing capabilities. Spark depends heavily on RDDs [3], which is further evolved into NDD in our Bigflow system.

Google's FlumeJava [4] is a Java library that provides a few simple abstractions for programming data-parallel computations. These abstractions are higher-level than those provided by MapReduce and provide better support for pipelines. Apache Crunch⑤ is an evolution of FlumeJava running on top of Hadoop MapReduce and Apache Spark. Several optimizations raised in the FlumeJava framework, such as ParallelDo Fusion and MSCR Fusion, are also adopted in the implementation of Bigflow on Hadoop.

Stratosphere [13] is an open-source software stack that also brings about automatic program parallelization and optimizations. Similar to Spark, it allows for expressing analysis jobs with dataflow graphs, which provides an intuitive way to describe data analysis jobs. Bigflow provides more parallel methods than Stratosphere, such as map, union and a series of methods based on keys.

DMLL [14] is an intermediate language based on common parallel patterns that capture the necessary semantic knowledge to efficiently target distributed heterogeneous architectures. It adopts the concept of nested parallelism that provides a way to exploit parallelism in nested structures such as multi-layer loops. It mainly aims at the heterogeneous cases, and its optimizations on those architectures are actually orthogonal to Bigflow.

Apache Beam⑥ proposes a unified model for defining both batch and streaming data-parallel processing pipelines that can be used to execute data processing pipelines on separated distributed engines. The model behind Beam is based on the Google Dataflow model [8] that supports out of order processing. It is a unified layer over several execution engines but still lacks special optimizations for batch engines up to now.

There are also other high-level abstractions for cluster programming based on different computing models, such as Piccolo [15], Twister4Azure [16] and Lambda-Blocks [17]. To the best of our knowledge, most of current approaches are inefficient for nested partitions, which are rather common in complicated projects.

8 Conclusions

In this paper, we presented an open-source framework called Bigflow, serving as an optimization layer

④https://hadoop.apache.org, Jan. 2020.

⑤https://crunch.apache.org, Jan. 2020.

⑥https://beam.apache.org, Jan. 2020.

over execution engines. It relies on a compact and expressive NDD data model, which can help solve several limitations in traditional distributed computing frameworks. Currently, it is widely employed in the datacenters of Baidu. According to the evalutation, Bigflow reduces the execution time by 30% and 50% when compared with PySpark and Hadoop respectively.

References

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008, 51(1): 107-113.

[2] Zaharia M, Chowdhury M, Franklin M J, Shenker S, Stoica I. Spark: Cluster computing with working sets. In *Proc. the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, June 2010, Article No. 5.

[3] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin M J, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. the 9th USENIX Conference on Networked Systems Design and Implementation*, April 2012, pp.15-28.

[4] Chambers C, Raniwala A, Perry F, Adams S, Henry R R, Bradshaw R, Weizenbaum N. FlumeJava: Easy, efficient data-parallel pipelines. In *Proc. the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010, pp.363-375.

[5] Meng X, Bradley J, Yavuz B *et al.* MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research*, 2016, 17: Article No. 34.

[6] Parsian M. Data Algorithms: Recipes for Scaling up with Hadoop and Spark. O'Reilly Media Inc., 2015.

[7] Karau H, Warren R. High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark (1st edition). O'Reilly Media Inc., 2017.

[8] Akidau T, Bradshaw R, Chambers C *et al.* The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 2015, 8(12): 1792-1803.

[9] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 2007, 44(3): 59-72.

[10] Saha B, Shah H, Seth S, Vijayaraghavan G, Murthy A, Curino C. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 2015, pp.1357-1369.

[11] Gates A F, Natkovich O, Chopra S, Kamath P, Narayanamurthy S M, Olston C, Reed B, Srinivasan S, Srivastava U. Building a high-level dataflow system on top of Map-Reduce: The pig experience. *Proceedings of the VLDB Endowment*, 2009, 2(2):1414-1425.

[12] Thusoo A, Sarma J S, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive: A warehousing solution over a Map-Reduce framework. *Proceedings of the VLDB Endowment*, 2009, 2(2): 1626-1629.

[13] Alexandrov A, Bergmann R, Ewen S *et al.* The stratosphere platform for big data analytics. *The VLDB Journal*, 2014, 23(6): 939-964.

[14] Brown K J, Lee H, Rompf T, Sujeeth A K, de Sa C, Aberger C, Olukotun K. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proc. the 2016 IEEE/ACM International Symposium on Code Generation and Optimization*, March 2016, pp.194-205.

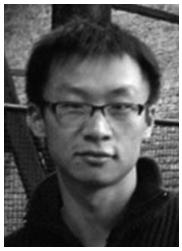
[15] Power R, Li J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proc. the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010, pp.293-306.

[16] Gunarathne T, Zhang B, Wu T L, Qiu J. Scalable parallel computing on clouds using Twister4Azure iterative MapReduce. *Future Generation Computer Systems*, 2013, 29(4): 1035-1048.

[17] Caneill M, de Palma N. Lambda-blocks: Data processing with topologies of blocks. In *Proc. the 2018 IEEE International Congress on Big Data*, July 2018, pp.9-16.

Yun-Cong Zhang received his B.S. degree from Nanyang Institute of Technology, Nanyang. He is now working in Baidu in the field of distributed computing and autonomous driving. He used to lead a team to build a unified presentation layer for distributed systems in the Department of Infrastructure. Now he is working in the Planning & Control team (PNC) in the Intelligence Driving Group.

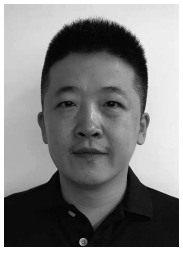
Xiao-Yang Wang received his B.S. degree in computer science from Peking University, Beijing, in 2015, where he is currently pursuing his Ph.D. degree with the Center for Energy-Efficient Computing and Applications. His current research interests include distributed systems and edge computing systems.



Cong Wang received his B.S. and M.S. degrees in computer science from Beihang University, Beijing, in 2009. Now he is working on distributed systems in Baidu.

Yao Xu received his B.S. degree in computer science from Northeast Forestry University, Harbin, in 2009. Now he is working on distributed systems in Baidu.

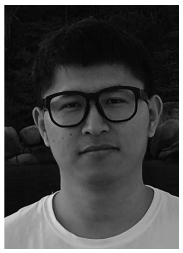
Jian-Wei Zhang received his M.S. degree in computer science from Zhejiang University, Hangzhou, in 2010. Now he is working on distributed systems in Baidu.



Xiao-Dong Lin received his M.S. degree from Dalian University of Technology, Dalian, in 1998. He joined Baidu in 2005 and holds the position as deputy general manager of Baidu Cloud Security. He works in the fields from low-level hardware and kernel to upper-layer distributed platforms. With extensive experience and unique insights, he led the construction of Baidu's scratch-free product security research and development system, and later serves as Baidu's director of infrastructure.

Guang-Yu Sun received his B.S. and M.S. degrees from Tsinghua University, Beijing, in 2003 and 2006, respectively. He received his Ph.D. degree in computer science from Pennsylvania State University, State College, USA, in 2011. He is currently an assistant professor of CECA (Center for Energy-Efficient Computing and Applications) at Peking University, Beijing. His research interests include computer architecture, VLSI Design, and electronic design automation (EDA). He has also served as a peer reviewer and technical referee for several journals, which include IEEE Micro, IEEE Transactions on Very Large Scale Integration Systems (TVLSI), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), etc. He is a member of CCF and IEEE.

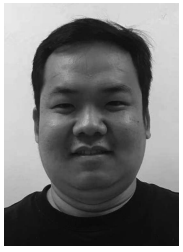
Gong-Lin Zheng received his B.S. degree in software engineering from South China Agricultural University, Guangzhou, in 2015. He is now working on distributed systems in Baidu.



Shan-Hui Yin received his M.S. degree in computer software and theory from Ocean University of China, Qingdao, in 2014. He is working on distributed systems in Baidu.

Xian-Jin Ye received his B.S. degree in information and communication engineering from Zhejiang University, Hangzhou, in 2013. He is working on distributed systems in Baidu.

Li Li received his B.S. degree in computer science from Yangtze University, Jingzhou, in 2013. Now he is working on distributed systems in Baidu.



Zhan Song received his M.S. degree in circuits and systems from Xidian University, Xi'an, in 2014, and is currently working on distributed systems in Baidu.

Dong-Dong Miao received his B.S. degree in computer science from Nanyang Institute of Technology, Nanyang, in 2013. Now he is working on distributed systems in Baidu.

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Volume 35, Number 2, March 2020

Special Section on Learning and Mining in Dynamic Environments

Special Section of ChinaSys 2019

JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

《计算机科学技术学报》

Volume 35 Number 2 2020 (Bimonthly, Started in 1986)

Indexed in: **SCIE, Ei, INSPEC, JST, AJ, MR, CA, DBLP**

Edited by:

THE EDITORIAL BOARD OF JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

Guo-Jie Li, Editor-in-Chief, P.O. Box 2704, Beijing 100190, P.R. China

Managing Editor: Feng-Di Shu E-mail: jcst@ict.ac.cn http://jcst.ict.ac.cn Tel.: 86-10-62610746

Sponsored by: Institute of Computing Technology, CAS & China Computer Federation

Supervised by: Chinese Academy of Sciences

Undertaken by: Institute of Computing Technology, CAS

Published by: Science Press, Beijing, China

Printed by: Beijing Kexin Printing House

Distributed by:

China: All Local Post Offices

Other Countries: Springer Nature Customer Service Center GmbH, Tiergartenstr. 15, 69121 Heidelberg, Germany

Available Online: https://link.springer.com/journal/11390

ISSN 1000-9000

国内统一刊号: CN11-2296/TP 国内邮发代号: 2-578 RMB ¥150.00