Chukonu: A Fully-Featured High-Performance Big Data Framework that Integrates a Native Compute Engine into Spark

Bowen Yu, Guanyu Feng, Huanqi Cao, Xiaohan Li, Zhenbo Sun, Haojie Wang, Xiaowei Zhu, Weimin Zheng, Wenguang Chen
Department of Computer Science and Technology and BNRist, Tsinghua University
{yubw15,fgy18,caohq18,xh-li18,sunzb20}@mails.tsinghua.edu.cn
{wanghaojie,zhuxiaowei,zwm-dcs,cwg}@tsinghua.edu.cn

ABSTRACT

Apache Spark is a widely deployed big data analytics framework that offers such attractive features as resiliency, load-balancing, and a rich ecosystem. However, there is still plenty of room for improvement in its performance. Although a data-parallel system in a native programming language significantly improves performance, it may require re-implementing many functionalities of Spark to become a full-featured system. It is desirable for native big data systems to just write a compute engine in native languages to ensure high efficiency, and reuse other mature features provided by Spark rather than re-implement everything. But the interaction between the JVM and the native world risks becoming a bottleneck.

This paper proposes Chukonu, a native big data framework that re-uses critical big data features provided by Spark. Owing to our novel DAG-splitting approach, the potential Spark integration overhead is alleviated, and its even outperforms existing pure native big data frameworks. Chukonu splits DAG programs into run-time parts and compile-time parts: The run-time parts are delegated to Spark to offload the complexities due to feature implementations. The compile-time parts are natively compiled. We propose a series of optimization techniques to be applied to the compile-time parts, such as operator fusion, vectorization, and compaction, to significantly reduce the Spark integration overhead. The results of evaluation show that Chukonu has a speedup of up to 71.58× (geometric mean 6.09×) over Apache Spark, and up to 7.20× (geometric mean 2.30×) over pure-native frameworks on six commonly-used big data applications. By translating the physical plan produced by SparkSQL into Chukonu programs, Chukonu accelerates SparkSQL's TPC-DS performance by 2.29×.

PVLDB Reference Format:
Bowen Yu, Guanyu Feng, Huanqi Cao, Xiaohan Li, Zhenbo Sun, Haojie Wang, Xiaowei Zhu, Weimin Zheng, Wenguang Chen. Chukonu: A Fully-Featured High-Performance Big Data Framework that Integrates a Native Compute Engine into Spark. PVLDB, 15(4): 872-885, 2022.
doi:10.14778/3503585.3503596

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503596

1 INTRODUCTION

Big data refers to a large volume of rapidly growing datasets in heterogeneous formats from which interesting value can be extracted [31]. Apache Hadoop [2] is an early open-source big data solution that includes a distributed file system (HDFS) to persistently store big data, together with an analytics framework based on the MapReduce abstraction [13]. More recently, Apache Spark [44] has introduced a new abstraction called Resilient Distributed Datasets (RDD) [42] to enable fault-tolerant data reusing for iterative workloads. It can achieve performance that is better by an order of magnitude compared with Hadoop MapReduce. Spark offers an expressive and easy-to-use API that can naturally express functional transformation, MapReduce, and join, making it possible to build supportive libraries for graph computing [19], stream processing [9, 43], machine learning [25], and SQL queries [10]. Spark is widely deployed nowadays to serve big data analytics [3].

Despite Spark's advantage in in-memory dataset reusing, recent work has shown that its performance can be significantly improved. For example, by building big data analytics frameworks in C++, Thrill [11] achieves a geometric mean of 3.26× in speedup over Spark on typical big data workloads. Actually, there is plenty of room for performance improvement: For a Java matrix-multiply kernel, switching to C yields a 4.4× speedup, and performance further improves by 9.45× from the vectorization and AVX intrinsics provided by the C compiler [23].

However, performance is only one aspect of big data processing. Spark provides many other critical features, such as its lineage-based resiliency: Big data analytics are typically performed in multi-tenant commodity clusters, in which task failures are very common due to machine failures, network jitters, and preemptive scheduling, making checkpointing not efficient to handle such frequent failures. Spark's lineage-based fault-tolerance mechanism allows to just recompute partial data instead of all. Its resiliency also enables other features, such as load balancing, straggler mitigation, and auto-scaling, which improve the resource utilization of the clusters. Moreover, Spark's ecosystem, such as its performance profiler with a Web UI and its integration with various resource managers, makes it easy to deploy, monitor, and profile applications on various private or public clouds. Thrill [11] features a native RDD-like abstraction called DIA, but tightly couples the data distribution to physical machines and that invalidates resiliency. Husky [41] uses an upstream message logging fault-tolerance mechanism that incurs a non-negligible overhead even when there is no failure [40, 42]. Compared with Spark, they lack many essential features needed for them to be fully-featured.

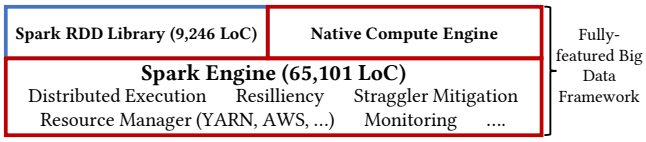

Figure 1: Building a fully-featured native big data framework by standing on the shoulders of Spark

It is clearly desirable to have a *fully-featured* native big data framework. An immediate solution would be to re-implement a new pure-native framework in a native programming language, which is straight-forward and theoretically possible but can be prohibitively expensive and unnecessary: The core component of Spark 3.0.1 has 74*K* lines of code[1], of which only 9*K* lines are directly related to the programming framework, including the RDD API and operator implementations, and the others are components that serve various big data features. Ousterhout et al. [32] have revealed that computation is the bottleneck of Spark. This inspires us to design an efficient big data framework by building a native programming framework that can reuse Spark's well-developed big data features (more than 87% of the LoC of the Spark core), as illustrated in Figure 1.

However, the feature-reusing approach creates several challenges. First, existing pure-native big data frameworks are incompatible with Spark's execution model, making it unfeasible to integrate them into Spark. For example, Thrill couples each dataset partition to a specific machine, and Husky relies on stateful task execution, which violate Spark's dynamic-scheduling and assumption of statelessness. Second, fine-grained interactions between the JVM and the native world, either through JNI or JNA, incur high overhead [15, 35] that risks becoming a new performance bottleneck. Therefore, a native programming framework that respects the Spark execution model is needed, and it is crucial to minimize the overheads related to Spark integration.

This paper presents Chukonu, a native big data framework with full features and high performances at the same time by taking the feature-reusing approach. It supports critical big data features by integrating into Spark and reusing its features. Chukonu is developed with a reasonable amount of human effort, 9K lines of C++ code, and 1.5K lines of Scala code, as it reuses well-developed big data features. Despite the risks of high integration overheads, Chukonu has successfully overcome the challenge: Its performance even goes beyond existing pure native big data frameworks that are designed for performance. This is due to our novel approach that splits the DAG program into compile-time parts and run-time parts: The *compile-time parts* handle such simple dataflow behaviors as transformation, data filtering, and per-partition sorting. The *run-time parts* handle such complex dataflow behaviors as scheduling, caching, and communication. Chukonu delegates the run-time parts to Spark so that complexities are offloaded.

To reduce the Spark integration overheads, Chukonu applies a series of optimizations to the compile-time parts: *Operator fusion* eliminates unnecessary JNA calls. *Vectorization* batches the per-element data processing to reduce the number of JNA calls. *Compaction* stores a batch of elements compactly in a few buffers so that data copies can be eliminated in JNA calls.

[1]Using `cloc` command. Comments and empty lines are excluded.

To execute the run-time parts in Spark, Chukonu implements a thin encapsulation over Spark. It is non-intrusive and can be easily submitted to existing environments, without the need of re-configuring existing cluster resource managers or recompiling existing Spark. It also makes several enhancements to further reduce the Spark integration overheads, including an optimized fast path for serialization, explicit pointer passing, and efficient data loading.

Despite Chukonu's superior performance, the optimizations needed for the compile-time parts prolong the compilation time. This makes Chukonu unsuitable for ad-hoc analytics when the reduced execution time is not large enough to cover the penalty from extra compilation time.

This paper makes the following contributions:

(1) We propose an approach to integrate a native compute engine into Spark with low overhead by splitting the DAG program into compile-time parts and run-time parts for native compilation and Spark execution respectively.
(2) We develop a series of optimizations to reduce the number of JNA calls and alleviate the JNA calls overhead, including operator fusion, vectorization, and compaction.
(3) To the best of our knowledge, this is the first implementation of a big data framework built based on the DAG splitting approach, with full features of the Spark core, that delivers performance competitive to pure-native frameworks.
(4) We provide a careful evaluation of Chukonu by comparing it with Spark and pure-native baseline frameworks.

We evaluated Chukonu on our in-house Hadoop cluster managed by YARN, and compare Chukonu with Spark RDD, Spark Tungsten [7], Husky [41], and Thrill [11]. Six big data applications were selected to evaluate these systems. We also implemented a code generator that translates the physical plan produced by SparkSQL into Chukonu programs. All the queries of the TPC-DS benchmark were used to evaluate the structured analytics of Spark and Chukonu. Although it is standing on the shoulder of Apache Spark, the results show that Chukonu delivers superior performance. On the six big data applications, its average speedups in comparison with Spark, Husky, and Thrill were 6.09× (up to 71.58×), 2.53× (up to 7.20×), and 2.08× (up to 3.45×), respectively. For the TPC-DS benchmark, its speedup of the total query execution time in comparison with Spark was 2.29× (up to 5.14× for Q67). This shows that Chukonu has significantly better performance than Spark, and also has performance competitive to pure-native frameworks, which justifies the DAG splitting approach.

2 RELATED WORK

Data-Parallel Systems. Data-parallel systems [1, 2, 13, 20, 22, 34, 44] provide users with a programming abstraction to hide the complicated details of distributed processing. MapReduce [13] is a user-friendly programming model that enables low-overhead fault tolerance based on re-execution. This is required in cluster computing to resist server failures or preemptions. Dryad [22] introduces dataflow graphs to facilitate multi-stage pipelines. RDD [42] and Apache Spark [44] were subsequently proposed to enable in-memory data reusing for iterative workloads. This provides lineage-based fault tolerance to provide low-overhead resiliency. Piccolo [34] provides

an alternative approach that exposes the user to a distributed shared mutable KV interface. It relies on checkpoints for fault tolerance.

Optimizing Big Data Systems within JVM. Some studies have examined optimizing the performance of existing big data systems within JVM. Optimizing within JVM allows for the API to remain unchanged so that user applications do not need to change. Facade [30], Lu et al. [24], and Gerenuk [27] proposed the automatic transformation of user-defined functions to enable them to support accessing data in serialized form and reduce garbage collection overheads by contiguously storing the serialized data. NumaGiC [18] proposes a NUMA-aware garbage collector for JVM. Yak [29] proposes a big-data-friendly garbage collector that is aware of the object lifetime patterns of big data.

Native Integration into Apache Spark. Using the power of native compilation helps to eliminate the overheads of JVM and achieves significant speedup [8, 15, 25, 35, 38]. Rosenfeld et al. [35] accelerated SparkSQL in a C++ native engine but leaves Java UDFs within an embedded JVM. Flare [15] exhibits order-of-magnitude speedups compared with SparkSQL by native-compiling the entire query plan into a single-machine native backend, but abandons Spark's support for distributed execution. Anderson et al. [8] provided an order-of-magnitude speedup by integrating MPI [36] programs into Spark. However, MPI does not support the low-overhead fault tolerance required by big data analytics on commodity servers [13]. MLlib [25] and SparkJNI [38] accelerates performance-critical computation by offloading to native kernel functions, but this leads to computation without native kernels in the JVM world, which may be a new bottleneck, and incur the cross-language data marshalling overhead.

Pure Native Big Data Systems. Some studies have built native systems from scratch to improve performance [11, 21, 41]. Thrill [11] proposes a native data-parallel system based on the Distributed Immutable Array (DIA) memory abstraction that is tightly coupled with the execution model of MPI, and does not support lineage-based fault tolerance like RDD. Husky [41] and Tangram [21] introduce data mutation to enable fine-grained access and asynchronous execution, at the cost of expensive data logging and checkpointing for fault tolerance.

3 DAG SPLITTING OVERVIEW

In this section, we introduce the idea of **DAG splitting** and illustrate how it helps to integrate a native compute engine into Spark with low overhead. A big data analytics program can be logically represented by a directed acyclic graph in which vertices are operators and edges are dependencies between two operators, and we call it a **DAG program**. Our approach splits the DAG program into run-time parts and compile-time parts. The **compile-time part** is a subgraph of a DAG program that will be natively compiled and optimized at compile time. The **run-time part** is a subgraph of a DAG program that will be directly executed by Spark. In §3.1 and §3.2, we will discuss how a DAG program is split into two parts and how compile-time parts are optimized, respectively.

3.1 Policy of Splitting the DAG Program

Chukonu splits the DAG program according to the category of the operator. Each operator may belong to either the compile-time part or the run-time part. Figure 2 shows a simple example of how Chukonu splits the DAG program. The code is excerpted from the PageRank application, and it corresponds to the DAG program below. In this example, the shuffle and zip operators are in the run-time part, and other operators are in the compile-time part.

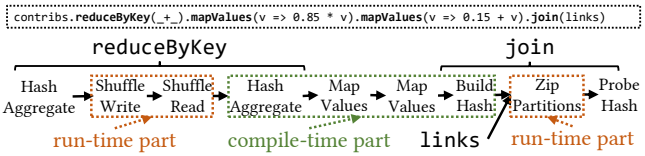

Figure 2: Chukonu splits the DAG program into compile-time parts and run-time parts.

It worths discussing the policy of assigning operators to the two categories. If an operator is assigned to the run-time part, it is delegated to Spark, and requires little human effort to implement, but potentially suffers from Spark integration overheads. We follow the principle that operators which are hard to implement should prefer the run-time part, unless they are shown to be a performance bottleneck. Under this approach, we try to avoid unnecessary human effort, as long as the Spark integration overheads are kept small. Chukonu assigns the following operators to the run-time part: First, the partition-pruning operator and operators with more than one dependencies, such as union, zip, and cartesian, are in the run-time part. This provides lineage-related information to Spark and reuses Spark's lineage-based fault tolerance mechanism. Second, the shuffle operator is in the run-time part. This reuses Spark's fault tolerant data shuffling mechanism, which is not only hard to implement but also intrusive to deploy: It requires re-configuring the cluster resource manager so that a dedicated shuffle service is deployed in each node of the cluster. Third, the cache operator is in the run-time part. This reuses Spark's intermediate data management mechanism. Fourth, the data-source operator is in the run-time part. This provides locality information to Spark and reuses Spark's locality-aware task scheduling. Other operators, such as `map`, `filter`, and `flatMap`, are assigned to the compile-time part, so that they can be optimized at compile time.

3.2 Optimizing Compile-Time Parts

Optimizations to compile-time parts are necessary to alleviate the Spark integration overheads. To illustrate this, Figure 3 (a) shows a naïve mapping from a compile-time part to Spark. Each `mapValues` transformation is directly mapped to a Spark RDD, and each value is mapped to a Java object. Despite its simplicity, the naïve integration suffers from a large overhead: Processing a value as simple as a long-double pair requires serialization/deserialization once in the JVM world and once in the native world, and the creation of a Java object. This cancels out the performance advantage of native efficiency. Chukonu performs the following three optimizations to compile-time parts: First, Chukonu performs *operator fusion*, meaning that each compile-time part of the DAG is fused, to eliminate unnecessary native/JVM interactions. Figure 3 (b) illustrates its effect: Two subsequent `mapValues` are fused into one `mapValues`. Then, Chukonu performs *vectorization*, which transforms per-element JNA calls into per-batch JNA calls on the boundary of the run-time part and the compile-time part, and also vectorizes the fused operators within the compile-time part. Figure 3 (c) illustrates its main

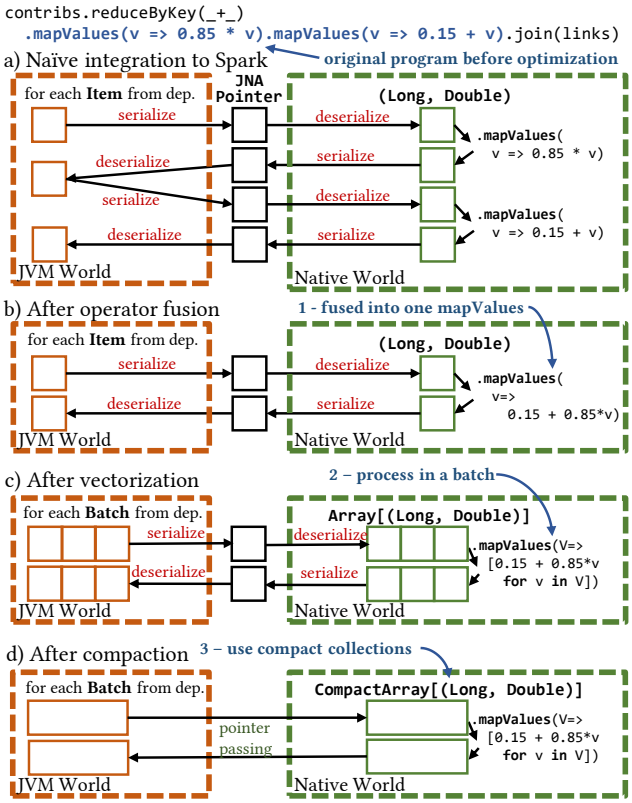

Figure 3: An example illustrating optimizations in Chukonu to alleviate overheads due to JNA calls

effect: The number of JNA calls is reduced because an array of pairs is passed by JNA at one time. Finally, Chukonu performs *compaction*, in which a few continuous memory buffers (called *compact data layout*) are used to represent a batch of elements. Figure 3 (d) illustrates its impact: A compact array of pairs backed by a buffer is explicitly passed by a pointer without data serialization. These optimizations are powered by code transformation. Figure 4 illustrates the optimizations of Chukonu from the perspective of code transformation with a slice of the page rank Chukonu program. These optimizations are further discussed in §4.1 and §4.2.

4 CHUKONU PROGRAMMING FRAMEWORK

In this section, we introduce the detailed design of the Chukonu programming framework. The architecture of Chukonu is illustrated in Figure 5. Chukonu provides a native RDD-like API for users to build DAG programs. The Chukonu (dataset) representation implements the DAG splitting and optimizations to compile-time parts. Chukonu implements a thin encapsulation over Spark called the Chukonu engine to execute run-time parts. §4.1 introduces the Chukonu representation and discusses how the DAG splitting and optimizations are implemented. §4.2 presents typical compact data layouts. §4.3 presents the API of Chukonu.

4.1 Implementing the DAG Splitting

Chukonu's internal dataset representation for DAG programs (referred to as *Chukonu representation*) enables the DAG splitting. Similar to Apache Spark, the Chukonu representation of each RDD

Table 1: RDD representations of Spark and Chukonu

	Spark	Chukonu
DAG type	run-time only	hybrid
Op. implementation	compiled binary	code template
Op. fusion	no	yes
Vectorization	no	yes
Compaction	run-time	compile-time
Data partition	abstract iterator	batched object
Elements extract	iterator get	scanner scan
Distribution Placement	share the same design	

contains a set of partitions, a set of dependencies, a definition of computation, and metadata about dataset distribution and data placement. However, the Chukonu representation introduces several differences to enable optimizations that are summarized in Table 1. First, Chukonu representation tracks both the run-time part and the compile-time part of the DAG program to support the DAG splitting. Figure 6 illustrates how the Chukonu representation represents the hybrid DAG. In contrast, Spark representation can only represent the run-time part. Second, operators in Chukonu representation are in the form of code templates so that optimizations to compile-time parts are enabled by C++ template meta-programming. In contrast, operators in Spark representation are in the form of binaries and prevent compile-time optimizations.

Figure 7 demonstrates the Chukonu representation in a C++-specific way. `RDD` represents a lazily-evaluated dataset that consists of a run-time part and a compile-time part. The run-time part is a data structure that stores a graph of run-time operators, and each run-time operator knows how to evaluate the result given a partition id. The compile-time part is represented by the compile-time operator based on *scanner*, which pushes a sequence of values to a consumer callback function given a result of run-time operators. Figure 6 shows how to perform operator fusion. The `scan`, `filter`, and `flatMap` are a chain of compile-time operators. When the `produce` of `flatMap` is called, it prepares a consumer that handles incoming values and passes the consumer to the `produce` of `filter`. Similarly, `filter` calls `scan`, and `scan` finally pushes values in the result to the consumer given by `filter`. The compiler will fuse the nested consumers calls via inlining. This operator fusion mechanism is inspired by the code generation approach proposed by Neumann et al. [28], but our approach is suitable for compile-time optimizations. To support vectorization, the result of run-time operators is a batch of values rather than a single value. As evaluating run-time operators involves heavy native/JVM overheads, vectorization will significantly reduce such overheads. To support compaction, the batch of values can be stored in a memory- and computing-efficient compact data layout. These optimizations adhere to the deterministic assumption of the RDD abstraction, so that the lineage-based fault tolerance is preserved.

The Chukonu representation and the optimizations are kept internal within the Chukonu framework. Users only need to understand that an RDD is logically a distributed dataset of elements, regardless of its type of partitions or producers. This is enabled by the `placeholder-type` [6] feature provided by C++14.

Original Program: `contribs.reduceByKey(_+_).mapValues(v => 0.85 * v).mapValues(v => 0.15 + v).join(links)`

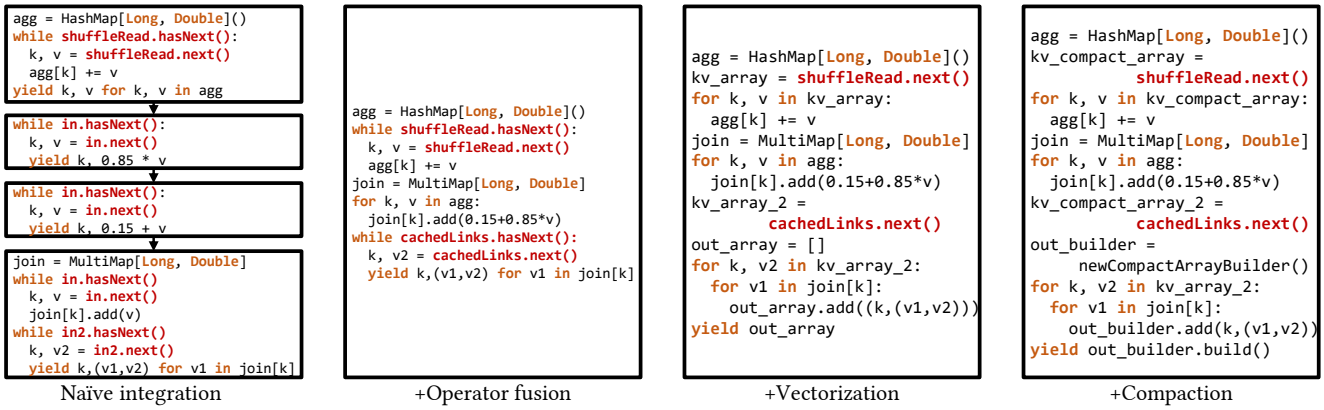

Figure 4: How Chukonu alleviates the integration overhead by code transformations. Most of the codes are executed natively except the iterator operations marked by red font, which present JNA call overheads.

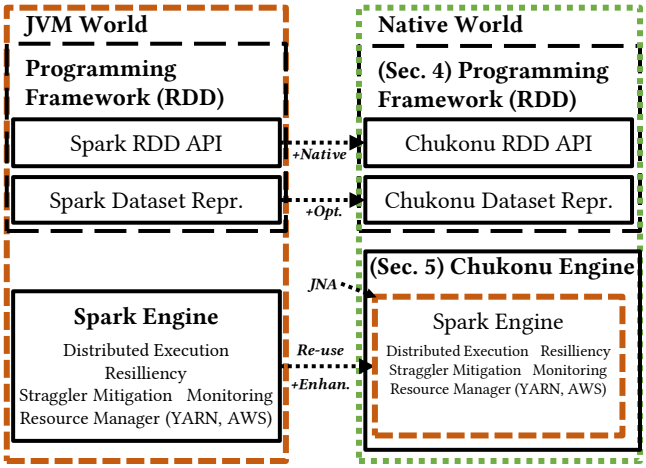

Figure 5: Chukonu architecture

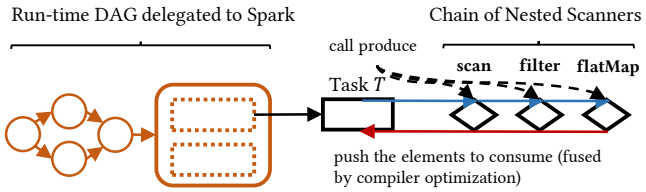

Figure 6: The hybrid DAG in the Chukonu representation

```
template <typename Scanner>
class RDD {
  DAG<RunTimeOperator> run_time_part;
  Scanner compile_time_part;
};
class RunTimeOperator {
  virtual Any compute(int partitionId) = 0;
};
template <typename T> class ScannerImpl {
  using value_type = T;
  /** Scans an input and feeds values to consumeOp */
  void produce(Any input, auto consumeOp) {...}
};
```

Figure 7: Chukonu representation in C++

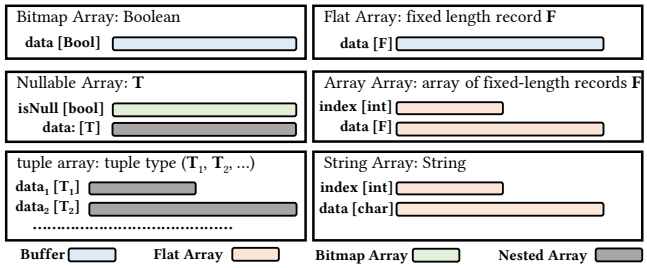

Figure 8: Chukonu provides code templates for generating compact data layout.

4.2 Typical Compact Data Layouts

For efficient processing, Chukonu internally represents partitions as *compact partition layouts*, in which elements of a partition are stored efficiently in a few buffers. This helps avoid memory fragmentation due to the allocation of small objects, and eliminates the requirement of run-time compaction of a garbage collector. Moreover, it increases data locality and provides a regular memory access pattern so that CPU efficiency can be improved.

Chukonu provides several templates to generate compact partition layout for various data structures, as shown in Figure 8. The flat-array places fixed-length elements in a buffer. The bitmap places boolean elements in a buffer. The array-array and the string-array place array elements in the CSR layout [45]. The nullable-array places nullable elements in a bitmap and an array. The tuple-array places tuple elements in a tuple of arrays. Although more kinds of compact layouts can be implemented if needed, Chukonu's compact partition layouts cover most of the cases in big data analytics: All of our workloads are based on these compact partition layouts.

Elements yielded from the compact data layout are *views*, like string-views or spans, and avoid the need to physically construct a string or vector to reduce the object creation overhead. Views are fundamentally pointers, and thus need to be carefully handled. Chukonu guarantees that input views, or their slices, can be returned from a UDF. However, it is illegal to return a view pointing to local object. To solve this problem, Chukonu allows users to return objects in their object form, and internally converts them to their view form.

4.3 Chukonu RDD API

The Chukonu RDD API is designed to be almost identical to the Spark RDD API, but in a native programming language C++. It provides a dataset abstraction based on coarse-grained immutable transformations in which a new dataset is created based on previous datasets. Dataset transformations are lazily evaluated: Datasets generated by the transformations do not hold the values but the DAG to generate the values. The name and semantics of its operators are identical to those of Spark to make it easily accessible to Spark users. Owing to the modern C++ syntax, programs based on the Chukonu RDD API and Spark RDD API are very similar. Figure 9 compares logistic regression programs with both Spark and Chukonu RDD APIs. They generate almost identical DAGs and the differences at the source code level reflect only language-specific details, such as the "auto" in C++.

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // randomly initialized vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
        p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

a) Logistic regression in Spark API (Scala)

```
auto points = ctx.textFile(...)
                 .map(parsePoint).persist()
auto w = // randomly initialized vector
for (int i=1; i<=ITERATIONS; ++i) {
  auto gradient = points.map(FN((auto p){
    return p.x*(1/(1+exp(-p.y*(dot(w,p.x)))-1)*p.y);
  }, w)).reduce(FN((auto l, auto r) { return l+r; }))
  w -= gradient
}
```

b) Logistic regression in Chukonu API (C++)

Figure 9: Logistic regression in Spark API and Chukonu API

Owing to a lack of standard serialization support in C++, Chukonu uses the serialization interface of cereal [4], which is a widely-used C++ serialization library. Spark requires that user-defined functions be serializable to enable the passing of control and support resiliency. Lambda expressions in standard C++ simplify the writing of UDFs but are not serializable. Therefore, Chukonu provides a macro "FN" to help users define UDFs through lambda expressions.

5 CHUKONU ENGINE

The Chukonu engine borrows most of the features of the Spark engine but enhanced it to support efficient Spark integration. §5.1 introduces how the Chukonu engine reuses Spark features. §5.2 introduces how the Chukonu engine provides safe explicit pointer passing for efficient Spark integration. §5.3 introduces how Chukonu eliminates integration overheads.

5.1 Integration Methodology

Figure 10 shows the workflow of running a Chukonu program, and illustrates the integration methodology. In the native world, the Chukonu engine contains a library that provides a trivial C++ binding of Apache Spark (referred to as *CppSpark*). Due to the space limit, we do not present its API here. The transformation of run-time part of DAGs is realized by CppSpark. The Chukonu

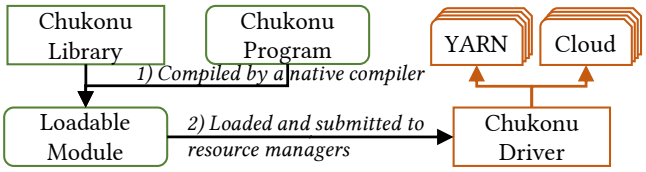

Figure 10: Chukonu workflow illustrating its integration methodology.

programs are compiled together with the library to yield a dynamically loadable module. In the JVM world, the Chukonu engine contains a driver that executes the compiled Chukonu programs. The Chukonu driver is a normal Spark application implemented in Java that can be submitted to Spark-compatible clusters, like normal Spark applications. Once submitted successfully, the Chukonu driver loads the compiled Chukonu program, registers the engine implementations to the module, and invokes the main routine of the Chukonu program. To support remote execution, the Chukonu driver also instructs newly created executors to prepare for the environment, for example, downloading the compiled Chukonu programs from HDFS.

5.2 Safe and Efficient Explicit Pointer Passing

Figure 11 illustrates how the Chukonu driver and the Chukonu library interact to provide a safe and efficient explicit pointer passing mechanism between JVM and the native environment. The Chukonu driver in the JVM world provides the DAG building and evaluation functionalities to the Chukonu library. This is provided by the JNA, in which the Chukonu driver passes the Java functions as native function pointers to the Chukonu library. The DAG in the Chukonu library is backed by the Spark RDD. Each object in the Chukonu library is backed by a Java object in the Chukonu driver. The Java object holds the pointer of objects. When finalized, Java wrappers free the underlying Chukonu objects. However, deferred finalization due to Java garbage collections will bloat the memory usage, making it desirable to release the Java wrappers as soon as possible, without waiting for the GC to start.

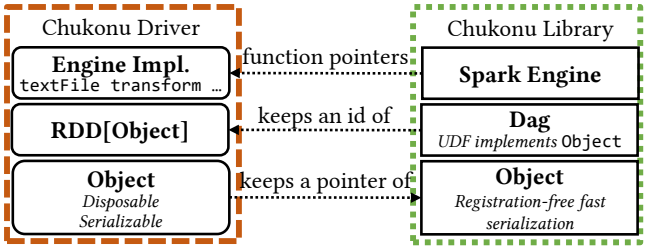

Figure 11: Interaction between Chukonu driver and Chukonu library. Dashed arrows are "owns-a" relationship, i.e., destruction of owner will destruct the owned.

Chukonu enforces a move-in move-out life-time contract for correct eager object freeing. For example, to call a native function in the JVM world, pointers passed to the function are considered to have been moved to it. After this, the JVM site can no longer access the moved pointers, and it is the native function's responsibility to free the input pointers or move them elsewhere. Such a life-time contract may encounter a few exceptions in which move-in is

not allowed, for example, when in-memory cached data are to be reused later and should not be moved in. To fix this issue, Chukonu wraps Chukonu objects within a `std::shared_ptr`, which is a smart reference mechanism provided by the C++ standard that offers automatic lifetime management similar to the references in Java. In the case of in-memory cached data, the reference pointed to by the raw pointer is duplicated and then moved in.

5.3 Other Overhead Elimination

The Chukonu engine also eliminates other overheads, including a fast data serialization and an accelerated data loading method.

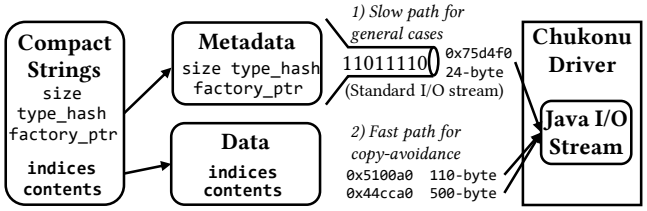

Figure 12: Chukonu provides a fast path for serializing the compact data layout.

5.3.1 Fast data serialization. Cereal supports serialization from/to standard I/O streams, which is a common practice, and is general enough to support every serializable type. However, its limitation is in its performance: Serialization based on I/O streams is inefficient. Chukonu proposes a data serialization technique that provides a fast path based on pointer-passing for the compact data layout that divides the data serialization process into two stages, meta-data serialization and data serialization, as illustrated in Figure 12. Meta-data serialization uses stream-based serialization considering its generality and makes it compatible with cereal. It is suitable for serializing meta-data such as dependencies, producers, and UDFs in the Chukonu RDD representation. Data serialization is a fast path that supports zero-copy serialization for continuous memory regions in compact data layouts, involves only a small amount of pointer-passing overheads, and does not incur the large data-copying overhead.

5.3.2 Acceleration of data loading. It is necessary to use Spark's data source to enable locality-aware scheduling. However, trivially reusing Apache Spark's data source API incurs a large overhead because it includes per-element data processing in the JVM world, which would be much more efficient if processed in the native world. For example, Spark's data source for text files creates a Java string for each line in the text file. Thus, the Chukonu engine provides an accelerated data loading implementation, in which the data source is partitioned and scheduled correctly, but each partition is read into a native buffer as a whole and passed to the native world for per-element data processing.

6 EVALUATION

We examined the benefits and limitations of building a big data framework standing on the shoulder of Apache Spark by evaluating the performance of our prototype system Chukonu, and comparing its performance with those of vanilla Apache Spark, domain-specific frameworks based on Apache Spark, and two pure native big data frameworks: Thrill and Husky. We evaluated these methods on four typical domains of big data: unstructured analytics, graph computing, machine learning, and structured analytics. Two representative big data applications were selected for each of the first three domains, and all queries of TPC-DS [33] were selected to represent structured analytics. In summary, we studied the following questions throughout the evaluation:

(1) How much performance can be improved by using Chukonu from the perspective of Spark users? (§6.2)
(2) How effective is the compile-time optimizations and the elimination of Spark integration overheads? (§6.3)
(3) Does Chukonu consume less memory than Spark? (§6.4)
(4) What are Chukonu's advantages compared with state-of-the-art pure-native big data frameworks? (§6.5, §6.6, §6.7)
(5) How much room is Chukonu leaving for future improvement? (§6.8, §6.9)

6.1 Experimental Methodology

We ran the experiments on our Hadoop cluster, which consisted of 8 nodes, each with 56 cores (two Intel E5-2680 v4 processors), 256GB of main memory, a 100Gbps network (InfiniBand EDR, configured IPoIB to expose it as a normal TCP/IP network), and a $6.4TB$ NVMe SSD that could provide 5.3GB/s of bandwidth. The distributed file system (HDFS) and cluster resource manager (YARN) were provided by Hadoop 3.0.0. The metastore of Apache Hive 3.1.2 was deployed to store the metadata of tables for structured analytics. Apache Spark 3.0, which was recently released and has significant performance enhancements over its predecessor, was used in our evaluation. Spark's shuffle service was configured in YARN to manage the intermediate results. Because Spark 3.0 supports both Java 8 and Java 11, we also used the newer version of JVM (Oracle JDK 11.0.8) because it has better performance. Both Chukonu and Spark programs were submitted through YARN, in which each container for the Spark executor had 7 cores and 22GB of memory. This follows the convention to avoid using large executors because this hampers performance [5]. The Chukonu and Chukonu programs are compiled by GCC 10.1.

We selecteded six representative big data applications. *WC* (Word Count, counting the occurrences of words from a given corpus) and *TS* (TeraSort, sorting a distributed dataset with 10-byte keys and 90-byte records with global ordering) were selected to represent *unstructured analytics*, which are non-iterative workloads that involve processing large datasets with heterogeneous data formats stored in a distributed file system, and optionally writing the results back into the distributed file system. *PR* (PageRank) and *CC* (Weakly Connected Components) were selected to represent *graph computing*. *KM* (K-Means[2]) and *LR* (Logistic Regression) were selected to represent *machine learning*.

Example applications of Apache Spark[3] were used as reference Spark applications, and were then manually ported to Chukonu as Chukonu programs. Several other baseline systems were also evaluated to compare with Chukonu. *PR* and *CC* were implemented using GraphX [19], which is a graph computing framework built on top of the RDD API. *KM* and *LR* were implemented using MLLib [25],

[2]The number of clusters K = 500
[3]See https://github.com/apache/spark/tree/v3.0.1/examples

which is a machine learning framework built on top of the RDD API, but users were provided with both the RDD API and the DataFrame API. In MLlib, numerical computations were accelerated by using a fast native BLAS library. Both Spark's and Chukonu's *KM* and *LR* were directly implemented by using Scala and C++ without native BLAS acceleration. To examine the improvement brought about by using a native BLAS library from Spark, we also implemented an accelerated version of *KM* and *LR* for Spark (referred to as *Spark-BLAS*) by using the native BLAS library bundled in MLlib. Project Tungsten [7] optimizes Apache Spark via unsafe serialized data processing with an SQL-like DataFrame API and query planner [10], and should be considered a baseline to investigate improvements in it. *WC* and *TS* were implemented directly using the DataFrame API. *PR* and *CC* were implemented using GraphFrame [12], which is a graph computing framework built on top of the DataFrame API. Thrill[4] and Husky[5] were also tested in the experiments to reveal their native performance as reference. Most of the applications have built-in implementations in the framework, and we have implemented the rest (*CC* for Thrill[6] and *TS*, *KM* for Husky[7]).

Table 2: Datasets used in the evaluation

App	Dataset (Size)
WC	Gutenberg corpus (43GB) [14]
TS	TeraSort generator in Hadoop examples (233GB)
PR, *CC*	Twitter-2010 (24GB) [17]
KM	Synthetic 80m points from BigDataBench [39] each with 64 features (43GB)
LR	Synthetic 120m points by duplicating MNIST [16] each with 784 features (213GB)

The datasets are summarized in Table 2. The datasets were stored in HDFS in text format, with the only exception being *TS*, which requires a binary format. *End-to-end time* is reported, which is the time from the start to the end of the application, and may include the time required to read from HDFS, pre-process data, and write to HDFS. To properly warm-up the JVM, the performance of each application was measured three times in the same session, and the average of the latter two times was reported. *PR*, *KM*, and *LR* performed 10 iterations in a single run. *CC*, instead, was iterated until it converged in a single run.

Since Spark 3.0, Spark provides monitoring features to periodically report the performance metrics of each executors in csv files, which makes it easy to profile and find the bottleneck of applications. Our evaluation uses Spark's performance monitoring feature with a one-second sampling interval. The clock skew of the cluster was within 100ms, and thus it was acceptable to aggregate the metrics of nodes in the cluster by timestamp to form the overall metrics. We only report the metrics of interest due to space limit.

To evaluate the TPC-DS benchmark, we used the data generator and query scripts developed by Databricks[8]. A scale factor of 1,000 was selected so that a 1TB dataset was generated. The metadata of tables were stored in the metastore of Apache Hive, and the data of tables were stored as Snappy-compressed Parquet files in the HDFS. We implemented a translator to translate SparkSQL's physical plan into the C++ source code of Chukonu programs. Chukonu shared the execution plans of Spark, which ensured a fair comparison between Chukonu and Spark. Owing to the DAG-splitting design of Chukonu, complexities due to input data formats and DAG scheduling were offloaded to Spark, leading to a simple implementation: 2K lines of Scala code for the translator and 1.9K lines of C++ code for the operators.

6.2 Overall Performance

The overall performance results are sketched in Figure 13, and provides preliminary answers to our questions. Chukonu achieved an average speedup of 6.09× with respect to Spark, which is a significant improvement. When considering each application, their speedups diverged: *TS* recorded only a 1.41× speedup, but *KM* had a speedup as high as 71.58×. By checking the Spark Web UI, we found that the relatively small speedup of *TS* was owing to a large amount of HDFS write: Chukonu was unable to accelerate the HDFS write, this rendering its acceleration in computation less significant. However, *KM* is a computation-intensive application, which makes accelerating computation important.

Although using the Tungsten backend is shown to be effective for structured analytics workloads like TPC-DS, our results show that despite a slight improvement, the Tungsten backend was not significantly effective for either unstructured analytics or graph computing: Chukonu had an average speedup of 2.04× compared with SparkSQL for *WC* and *TS*, and 8.98× compared with GraphFrame for *PR* and *CC*. Moreover, GraphFrame performed poorly compared to GraphX for both *PR* and *CC*: GraphFrame slowed down by 6.85× and 1.97×, respectively. This result seems counter-intuitive because the Tungsten backend should have eliminated the creation of Java objects by directly processing serialized data within a region of continuous memory. However, such an improvement is obtained by using an unnatural, low-level API called *Unsafe* in Java that is not directly applicable to existing applications. It requires a code generator, SparkSQL here, to fill the gap. Even if we can use the DataFrame API to represent unstructured analytics or graph computing, SparkSQL itself is designed for structured analytics workloads, and may not be effective for other domains. This suggests an advantage of Chukonu: Using an easy-to-use native API, it is guaranteed to achieve native performance without the need for domain-specific code generators.

Chukonu achieved competitive performance compared with state-of-the-art pure native frameworks including Thrill and Husky. The average speedups of Chukonu with respect to Thrill and Husky were 2.08× and 2.53×, respectively. Chukonu's end-to-end performance was better than that of Thrill and Husky for most applications, except Husky's *CC*. Husky's *CC* had an end-to-end speedup of 1.39× compared with Chukonu. This is because Husky is based on mutable datasets, which allows the user to place a source vertex's neighboring vertices list and mutable state together. However, the immutable execution model requires storing them separately, and this incurs a join overhead. It should be noted that despite Husky's

[4]https://github.com/thrill/thrill [commit 12c5b59]
[5]https://github.com/husky-team/husky [commit 9e66349]
[6]https://github.com/thu-pacman/thrill/tree/vldb22
[7]https://github.com/thu-pacman/husky/tree/vldb22
[8]https://github.com/databricks/spark-sql-perf [commit ca4ccea]. Although TPC-DS defined 99 queries, this implementation further divided each multi-part query into multiple single-part queries, and thus had 103 query scripts.

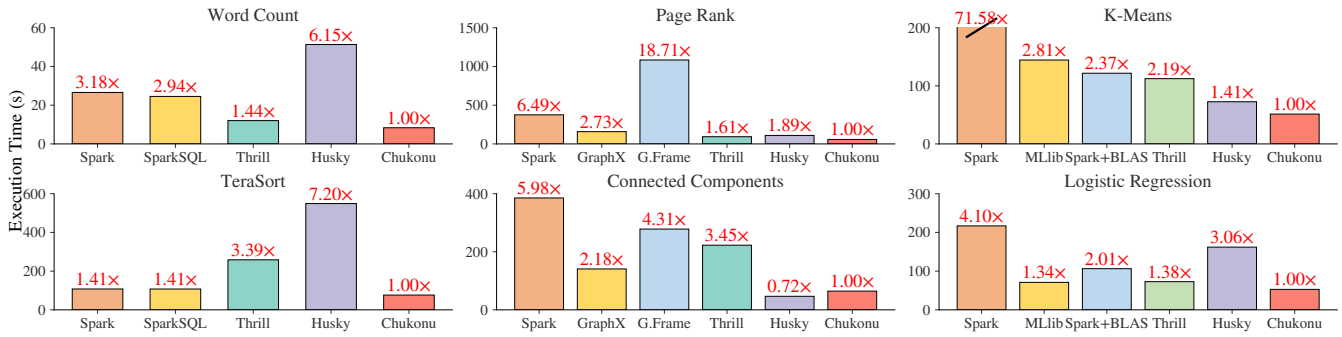

Figure 13: The end-to-end time and normalized time for each big data application (lower is better)

performance advantage in *CC* due to its mutable execution model, it was slower than Chukonu on the other five big data applications. Notably, for unstructured analytics workloads (*WC* and *TS*), Husky was 6.65× slower on average.

Chukonu's straight-forward implementations were better than those of *all* domain-specific frameworks based on Spark, with an average speedup of 2.85×, including Spark-BLAS, GraphX, MLLib, SparkSQL, and GraphFrame, although they are free to include optimizations tailored to a specific domain. For example, Spark-BLAS offloads numerical computations to a well-optimized native BLAS library, which significantly improves performance. But the conversion between Java objects and native buffers caused by the JNI calls requires memory copies, thus canceling out part of the advantage of the native BLAS library. This suggests that using Spark is not a free lunch: It offers Spark compatibility and provides RDD abstraction, but reduces efficiency. Chukonu can be an efficient fundamental framework based on which domain-specific frameworks can be built: It has Spark compatibility, provides RDD abstraction, but preserves opportunities for optimizing domain-specific frameworks.

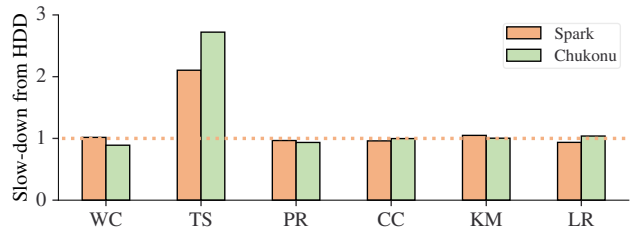

Figure 14: Slowdown of changing each NVMe SSD to 6 HDDs

Although our experiments were performed in an all-flash cluster equipped with NVMe SSDs, the performance advantage of Chukonu still holds in conventional HDD clusters. Figure 14 illustrates the performance slowdown when each NVMe SSD was changed to 6 HDDs for each machine in our cluster. Most of the applications, except *TS*, were not affected because the size of the dataset fitted into the memory, and was cached in either Spark's block manager or the operating system's page cache. However, *TS*'s HDFS writing operation suffers from the low bandwidth of HDDs because it waits for the data to be completely written to disks for durability.

6.3 Optimizations Validity

Chukonu relies on several techniques to optimize Chukonu programs and reduces the overhead incurred due to integration into Apache Spark. To validate these techniques, we evaluated two strawman baselines (referred to as *Strawman A/B*) and compared their performance with that of Chukonu.

Strawman A turns off the optimizations to Chukonu programs, including operator fusion, vectorization, and compaction, by mapping each element-centric user-defined function directly to Apache Spark. Strawman A involves per-element overhead of object creation and object serialization, representing the naïve integration approach. Evaluation results revealed that Strawman A has an average slowdown of 5.89× compared with Chukonu, as shown in Figure 15, justifying the benefits of the optimizations. Its *WC*, *TS*, and *PR* were slower than those of Spark, which suggests that overheads in the naïve integration approach canceled out the benefits of native efficiency. Strawman A also required more memory because the elements in datasets were stored in object form. *CC* ran out of memory due to the requirement of a large amount of memory.

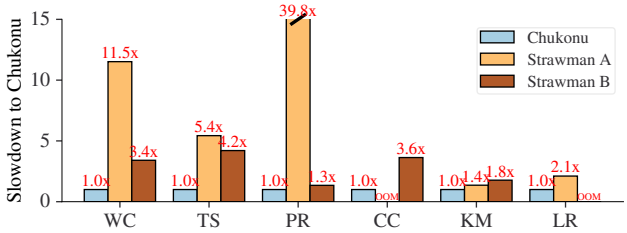

Figure 15: Normalized execution times of Strawman A and B relative to Chukonu

Strawman B is a baseline that turns off the enhancements of the Chukonu engine. Rather than explicit memory management with pointers, it uses Java's on-heap buffers to store continuous memory regions. It simplifies object lifetime management, but requires an additional memory copy to convert a buffer between its Java form and its native form. Moreover, HDFS data sources of Apache Spark were used directly, rather than Chukonu's efficient HDFS data source. The results reveal that Strawman B has an average slowdown of 2.62× compared with Chukonu, as shown in Figure 15. The simple lifetime management not only degraded the performance, but also consumed more memory because buffers both in Java form and the native form needed to be allocated simultaneously to perform memory copy. *LR* ran out of memory due to this issue.

6.4 Memory Consumption

We also evaluated the memory consumption of Chukonu by analyzing the `ProcessTreeJVMRSSMemory` metric, which monitors the amount of physical memory being used by the executor. Table 3 summarizes the memory usages, defined as the maximum amount of memory used during the executions of Spark and Chukonu for each application. The results show that Chukonu consumed on average 30.02% of memory consumed by Spark. This justifies the design of Chukonu: Although the partition-centric execution design of Chukonu coarsened data evaluation, which potentially increases the amount of memory used, its memory overhead was canceled out by the benefits of the compact data layouts and native executions.

Table 3: Maximum memory usages of Spark and Chukonu

	WC	TS	PR	CC	KM	LR
Spark (GB)	635	904	1415	1578	943	1488
Chukonu (GB)	108	780	369	419	123	859
%	*17*	*86*	*26*	*26*	*13*	*57*

To confirm the results in detail, we analyzed traces of the memory usages of Spark, Spark libraries, and Chukonu, as shown in Figure 16, from which several observations can be made. First, using the maximum memory usages reported in Table 3 is a valid way to summarize the memory requirement in normal conditions, because the maximum memory is reached due to a steady increasing, rather than due to spikes. Second, the memory usages of Chukonu were the best in *WC*, *PR*, *CC*, *KM*, and *LR*, with the only exception being that the *TS* of SparkSQL consumed less memory. This is because the Tungsten backend optimized *TS* while avoiding the memory overhead due to the partition-centric execution in Chukonu. However, its effectiveness depends on the code generator, which may even have a negative impact if it cannot handle this well, as in the case of *PR* of GraphFrame.

6.5 Fault Tolerance

We evaluated the fault tolerance functionality of the baseline systems via failure injection. Both Husky and Thrill crashed when one of their workers was killed. Chukonu supported fault tolerance with robust and fast recovery, owing to the lineage-based fault tolerance provided by the Spark engine. When a failure was injected, a randomly chosen executor process was killed. For non-iterative applications, including *WC* and *TS*, which consist of a map stage and a reduce stage, a failure was injected at the beginning of the reduce stage. For iterative applications, including *PR*, *CC*, *KM*, and *LR*, a failure was injected at the beginning of the 6-th iteration.

Figure 17 shows the results. Like Spark, Chukonu survived the tests and produced results as expected, with an average slowdown of 1.22×, comparing to 1.07× of Spark. The recovery overheads of *WC* and *TS* were small despite failures occurring in the reduce stage because the intermediate data produced by the map stage were managed by the shuffle service, and remained intact upon executor failure. The overheads of other iterative applications were incurred primarily from recomputing the lost cached data, such as reading from HDFS or the shuffle service. The recovery overhead of Chukonu *LR* was high because it had a short computation time, and reading from HDFS accounted for a significant portion of its end-to-end time.

6.6 Straggler Mitigation

We evaluated the straggler mitigation functionality of Chukonu by constructing stragglers through misconfiguration, in which one of the 8 machines simulated a straggler by offlining its 53 CPU cores, leaving only 3 CPU cores online. Chukonu can tolerate stragglers by detecting tasks that fall behind the overall progress and speculatively launching cloned tasks. We compared the performance slowdown of Chukonu and Spark under this configuration, with speculative execution enabled. Figure 18 shows that the average slowdowns of Chukonu and Spark were 1.35× and 1.50×, respectively, near the theoretical slowdown of 1.14× due to a reduction in computing resources.

6.7 Programmer Productivity

We then compared the programmer productivity of Chukonu with the baseline systems: Spark, Thrill, and Husky. Programmer productivity was measured by counting the lines of code (LoC) of each application using the `cloc` tool, excluding blank lines and comments. The result is shown in Figure 19. The LoCs of Chukonu programs were slightly higher than those of Spark due to different programming languages. Due to RDD abstraction, Chukonu had much fewer lines of codes to implement its applications.

6.8 Bottleneck Analysis

To assess how much performance is Chukonu leaving on the table, we analyzed its resource utilization. Ousterhout et al. [32] found that the CPU is the major bottleneck in Spark. With the significant improvement in CPU efficiency, the average CPU utilization of Chukonu was 78.05% that of Spark's, as shown in Table 4.

Table 4: Average numbers of cores of Spark and Chukonu (maximum=448)

	WC	TS	PR	CC	KM	LR
Spark (cores)	312	231	300	313	295	369
Chukonu (cores)	203	184	211	215	293	332
%	*65*	*80*	*70*	*69*	*100*	*90*

What causes Chukonu's CPU under-utilization? The traces of resource utilization shown in Figure 20 provide detailed information. First, repetitive launching and waiting for a group of small tasks for each iteration was the primary cause of CPU under-utilization, as reflected in the iterative workloads in *KM*, *LR*, *PR*, and *CC*. This is because task-launching incurs unscalable overheads on the driver side, for example, task serialization and state tracking, and task-waiting causes the computing resources to become idle. A large number of small tasks magnify their effects. Second, shuffle read was a major source of CPU under-utilization, and shown by the shuffle-intensive workloads like *TS*, *PR*, and *CC*. Network bandwidth was not to blame: Our cluster provided about 80GB/s of total bandwidth, and was far from saturated. Thus, the problem lies in the software stack, either in the Spark shuffler or in the TCP/IP protocol. We exepcted an ideal speedup of 1.91× if the CPU under-utilization could be eliminated, and plan to address this in future work.

6.9 Compilation time

The compilation time of Chukonu was compared with those of the baselines. For C++ frameworks including Chukonu, Thrill, and

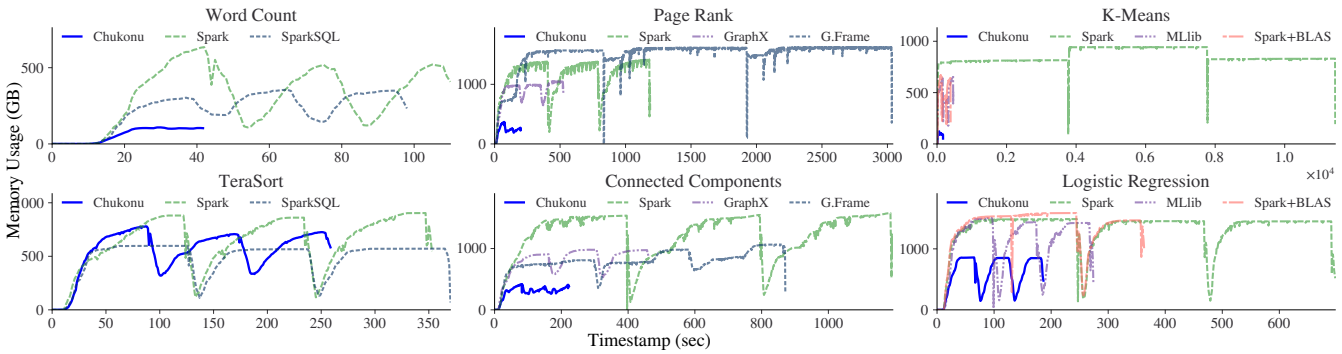

Figure 16: Memory usage traces of Spark, Spark libraries, and Chukonu for each application

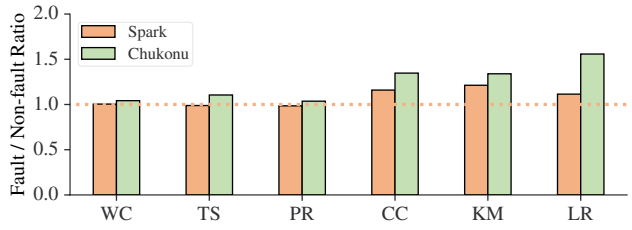

Figure 17: Comparing the recovery overhead of Chukonu and Spark

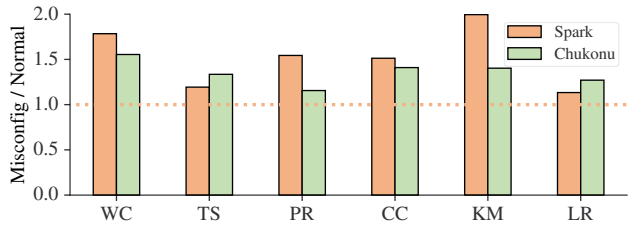

Figure 18: Comparing the slowdown due to misconfiguration in Chukonu and Spark

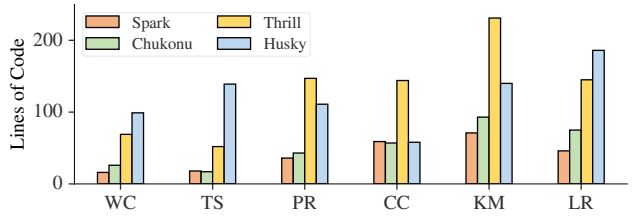

Figure 19: Comparing the lines of code of each application

Husky, the times to compile the `.cpp` source file into a `.o` binary file were measured. For Spark, the time to compile the `.scala` source file into a `.class` bytecode file was measured. The results are shown in Figure 21. Because Chukonu performs operator fusion, vectorization, and compact data layout for RDD abstraction, its compilation time was significantly higher than those of Thrill and Husky. Thrill performs only operator fusion for DIA abstraction, because of which its compilation time was shorter than that of Chukonu but longer than that of Husky. Spark had the shortest compilation time because Java programs are fast to compile.

It is worth discussing how the long compilation time of Chukonu affected its applicability. We considered two cases: *scheduled analytics* that were periodically executed and *ad-hoc analytics* that were impromptu and constructed to answer immediately. Because scheduled analytics allow for the reuse of the compiled binaries, the penalty owing to long compilation times vanished in multiple runs. Thus, Chukonu is applicable to scheduled analytics. However, the compilation times should be accounted for in ad-hoc analytics. Figure 21 illustrates the latency of ad-hoc analytics. The average speedup of Chukonu compared with Spark was 3.25×, illustrating penalties due to compilation times. Even though the execution time of Chukonu's *WC* decreased by 18.23*s*, its compilation time increased by 25.70*s*, making Chukonu's *WC* worse than that of Spark for ad-hoc analytics. The other five applications of Chukonu were faster even when the compilation times were included. This is because their durations were much longer than that of *WC*, and the reduced execution time tended to compensate for the compilation time. Thus, Chukonu is applicable to long running ad-hoc analytics.

6.10 Structured Analytics

We evaluated the performance of Chukonu on the TPC-DS benchmark and compared it with that of Spark. The total execution times of Spark and Chukonu were 2757.79s and 1204.75s, respectively, with a speedup of 2.29×. Most of the queries were short: 61 queries had a query time of less than 10s in Spark. Several long-duration queries dominated the total query time: the top five queries accounted 1322s (48%) to Spark's total run time.

Figure 22 shows the speedup of each query. It shows that Chukonu accelerated its performance on 98 queries. Most notably, the time of Q67 was reduced from 786.63*s* to 152.93*s* with a speedup of 5.14×. This suggests that Chukonu's optimization is effective and its integration into Spark incurs a low overhead. Five queries were slightly slower than Spark, and their average slowdown was only 2.76%. This is because their bottlenecks were in processing the Parquet input data format, which is not accelerated by Chukonu.

7 DISCUSSION

7.1 Chukonu versus SparkSQL

It is possible to generate native code from SparkSQL to improve performance [15]. But SparkSQL is dedicated to structured analytics. It is difficult and counterintuitive to use SparkSQL to express numerically-intensive algorithms, such as Logistic Regression and K-Means. Although SparkSQL supports UDFs to extend its functionality, its query optimizer cannot optimize these UDFs efficiently because the query language and the UDFs are in different systems.

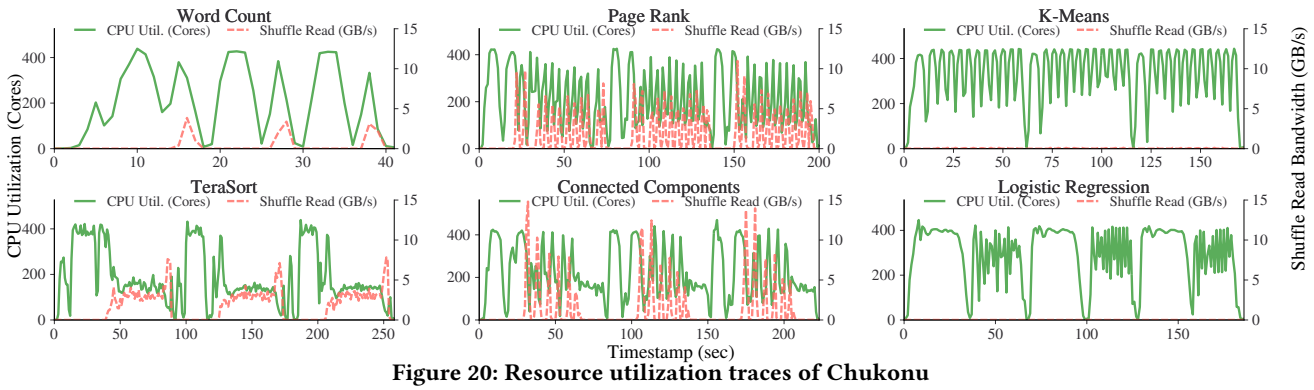

Figure 20: Resource utilization traces of Chukonu

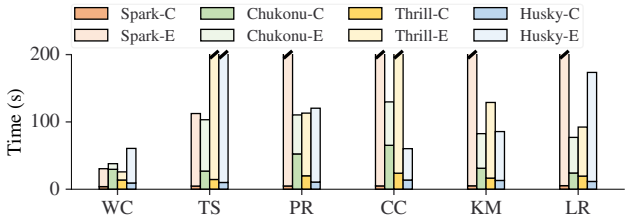

Figure 21: Compilation times (like Spark-C) and execution times (like Spark-E). The summation of both represents the latency in ad-hoc analytics.

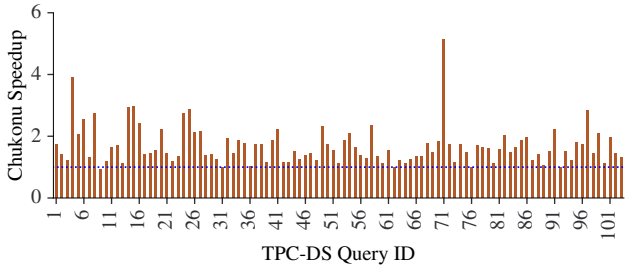

Figure 22: Speedup of Chukonu for each TPC-DS query. Queries are ordered by names and mapped to 1~103. For example, Q14a is mapped to 14 and Q14b is mapped to 15.

Chukonu is an optimized RDD layer enabling general-purpose native compilation, based on which we can intuitively express these applications. We argue that a fast RDD layer system is critical for data processing because it is a fundamental layer for other modules, such as graph processing and machine learning.

7.2 Chukonu versus Flare

Flare accelerates SparkSQL using native acceleration. Lacking a fast RDD-layer forces Flare to abandon Spark RDD altogether [15]; it turns to a single-machine architecture and sacrifices scalability. Chukonu is a fast and native RDD infrastructure, based on which we implemented a simple SQL layer that accelerated SparkSQL, and enabled faster distributed structured analytics. Although we are glad to try Flare, its source code is not publicly available..

7.3 Pros and Cons of Using C++

Although traditional C++ leads to tedious implementations, modern C++ has a rich set of language constructs that allows for building a concise API. Chukonu is based on C++20, and can provide an API that is almost identical to the Spark RDD API. A potential problem with C++ is its longer compilation time, as revealed in §6.9. We argue that a compilation time within a minute is acceptable for big data analytics, considering the improvement in computing efficiency. Another potential problem is that C++ lacks standard support for an interactive REPL. A community C++ REPL like Cling [37] could provide interactive analytics support. But for now, it does not support C++20, and thus Chukonu cannot use it for interactive analytics. We will consider this in our future work when C++20 support for Cling is available.

7.4 Alternative Execution Engines

Ray [26] is a distributed execution engine that implements actor abstraction, supports stateful execution, and scales for fine-grained task-launching. Extending Chukonu to build native RDD abstractions on top of Ray can help overcome the task launching overhead in Spark and improve scalability. We will try this in future work.

8 CONCLUSION

In this paper, we proposed an verified cost-effective means of building a fast, general, and resilient big data system Chukonu by reusing the attractive big data features provided by Spark. Users develop Chukonu programs using a native RDD API, which is aligned with the RDD API of Spark. With a novel DAG splitting approach, Chukonu programs can be efficiently executed from within Spark. The results of evaluations on six big data applications in our 448-core in-house Hadoop cluster showed that Chukonu can obtain an end-to-end speedup of up to 71.58× (geometric mean 6.09×) over Spark, and up to 7.20× (geometric mean 2.30×) over pure-native big data frameworks. Chukonu also accelerates SparkSQL's TPC-DS performance by 2.29×. The results justify Chukonu's feature-reusing approach: Chukonu not only has comprehensive features inherited from Spark, but also delivers performance competitive to pure-native frameworks. We expect that our work will motivate future efforts in the community to build fast, easily programmable, and resilient data parallel systems.

ACKNOWLEDGMENTS

This work was partially supported by National Key Research & Development Plan of China under grant 2017YFA0604500 and NSFC U20B2044. The corresponding author is Wenguang Chen.

REFERENCES

[1] [n.d.]. Apache Flink. https://flink.apache.org/. [Online; accessed 2021-12-22].

[2] [n.d.]. Apache Hadoop. https://hadoop.apache.org/. [Online; accessed 2021-12-22].

[3] [n.d.]. Apache Spark Survey 2016 Report. https://pages.databricks.com/2016-spark-survey.html. [Online; accessed 2021-12-22].

[4] [n.d.]. Cereal. https://github.com/USCiLab/cereal. [Online; accessed 2021-12-22].

[5] [n.d.]. How-to: Tune Your Apache Spark Jobs. https://blog.cloudera.com/how-to-tune-your-apache-spark-jobs-part-2/. [Online; accessed 2021-12-22].

[6] [n.d.]. placeholder type specifiers. https://en.cppreference.com/w/cpp/language/auto. [Online; accessed 2021-12-22].

[7] [n.d.]. Project Tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. [Online; accessed 2021-12-22].

[8] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. 2017. Bridging the Gap between HPC and Big Data Frameworks. *Proc. VLDB Endow.* 10, 8 (April 2017), 901–912. https://doi.org/10.14778/3090163.3090168

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[11] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. 2016. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data (Big Data)*. 172–183. https://doi.org/10.1109/BigData.2016.7840603

[12] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems* (Redwood Shores, California) *(GRADES '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2960414.2960416

[13] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[14] deepmind. accessed September 16, 2020. *PG-19 Language Modelling Benchmark*. https://github.com/deepmind/pg19.

[15] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

[16] Yann LeCun et al. accessed September 16, 2020. *THE MNIST DATABASE of handwritten digits*. http://yann.lecun.com/exdb/mnist.

[17] Laboratory for Web Algorithmics. accessed September 16, 2020. *Webgraph Datasets*. http://law.di.unimi.it/datasets.php.

[18] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[19] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[20] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 75–88.

[21] Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu. 2019. Tangram: bridging immutable and mutable abstractions for distributed data analytics. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 191–206.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[23] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020). https://doi.org/10.1126/science.aam9744 arXiv:https://science.sciencemag.org/content/368/6495/eaam9744.full.pdf

[24] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 936–947. https://doi.org/10.14778/2994509.2994513

[25] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[26] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 561–577.

[27] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. ACM, New York, NY, USA, 538–553. https://doi.org/10.1145/3341301.3359643

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[29] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 349–365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen

[30] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 675–690. https://doi.org/10.1145/2694344.2694345

[31] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. 2018. Big Data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences* 30, 4 (2018), 431–448. https://doi.org/10.1016/j.jksuci.2017.06.001

[32] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[33] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *VLDB*, Vol. 7. 1138–1149.

[34] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 293–306. http://dl.acm.org/citation.cfm?id=1924943.1924964

[35] Viktor Rosenfeld, Rene Mueller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ Environment. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 419–431. https://doi.org/10.1145/3127479.3132022

[36] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press.

[37] V Vasilev, Ph Canal, A Naumann, and P Russo. 2012. Cling – The New Interactive Interpreter for ROOT 6. *Journal of Physics: Conference Series* 396, 5 (dec 2012), 052071. https://doi.org/10.1088/1742-6596/396/5/052071

[38] Tudor Alexandru Voicu and Zaid Al-Ars. 2019. SparkJNI: A Toolchain for Hardware Accelerated Big Data Apache Spark. In *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*. 152–157. https://doi.org/10.1109/ICBDA.2019.8713201

[39] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[40] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 338–352.

[41] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *Proc. VLDB Endow.* 9, 5 (Jan. 2016), 420–431. https://doi.org/10.14778/2876473.2876477

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[43] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[45] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu