# Supporting Fine-grained Dataflow Parallelism in Big Data Systems

Sebastian Ertel
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
sebastian.ertel@tu-dresden.de

Justus Adam
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
justus.adam@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

## ABSTRACT

Big data systems scale with the number of cores in a cluster for the parts of an application that can be executed in data parallel fashion. It has been recently reported, however, that these systems fail to translate hardware improvements, such as increased network bandwidth, into a higher throughput. This is particularly the case for applications that have inherent sequential, computationally intensive phases. In this paper, we analyze the data processing cores of state-of-the-art big data systems to find the cause for these scalability problems. We identify design patterns in the code that are suitable for pipeline and task-level parallelism, potentially increasing application performance. As a proof of concept, we rewrite parts of the Hadoop MapReduce framework in an implicit parallel language that exploits this parallelism without adding code complexity. Our experiments on a data analytics workload show throughput speedups of up to 3.5x.

## CCS CONCEPTS

• **Information systems** → *DBMS engine architectures*; • **Software and its engineering** → *Parallel programming languages*;

**ACM Reference Format:**
Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In *PMAM'18: 9th International Workshop on Programming Models and Applications for Multicores and Manycores, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178442.3178447

## 1 INTRODUCTION

Over the last decade big data analytics became the major source of new insights in science and industry. Applications include the identification of mutations in cancer genome [10] and the tracking of other vehicles around an autonomously driving car. The big data systems (BDS) that enable such analyses have to be able to process massive amounts of data as fast as possible. In order to do so, current BDS apply coarse-grained data parallelism, i.e., they execute the same code on each core of the nodes in a cluster on a different chunk of the data. As such, the application is said to scale with the number of cores in the cluster. However, not every aspect of a big data application exposes data parallelism. Whenever this is the case, current big data systems fail to scale.

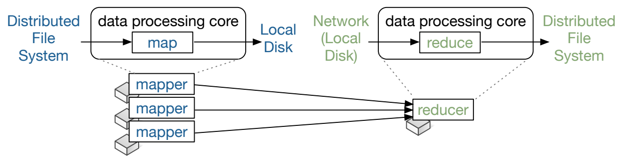


**Figure 1: Data processing cores in Hadoop's MapReduce.**

### 1.1 Scalability Issues of Big Data Systems

A typical big data analysis program assembles a set of predefined operations and applies them to the data. Execution proceeds in multiple phases where each phase applies a part of the program to the data that was output by a previous one. Afterwards, the results are redistributed among the nodes in the cluster to compute the next phase. For example, the famous MapReduce programming model defines exactly two phases as shown in Figure 1: a map and a reduce phase [4]. The map phase is data parallel by definition but the data parallelism of the reduce phase depends on the application. For example, in data analytics queries, the join operation for two tables can not be performed in a data parallel way (when the input data is not partitioned). In such a case, a single node receives all results from the map phase and becomes the throughput bottleneck. Even if the reduce phase runs in a data parallel fashion, the load across the reducers can only be balanced evenly by making assumptions on the map output. These assumptions are hard to estimate correctly such that the load is often spread unevenly decreasing data parallelism.

BDS have been traditionally designed to execute applications in a massive coarse-grained data parallel way across a cluster of machines. The assumption was that applications would process large amounts of simply structured data, e.g., text, that is fast to serialize and deserialize, i.e., transforming them to bytes (and its inverse operation on the receiver side). This setup led to the common belief that network I/O, instead of computation, is the performance bottleneck in these systems.

Only recently, researchers have shown that I/O is not always the limiting factor for performance [17]. Over the last years, big data applications have grown in complexity and use much more complex data structures especially in the area of data analytics. At the same time, network hardware for big data appliances improved, reaching transfer rates above 40 GB/s [11]. Authors in [24] benchmarked the current state-of-the-art BDS Apache Spark and Apache Flink in a high bandwidth cluster setup. The results show that reduce operations do not profit from modern multi-core architectures since their cores do not take advantage of fine-grained parallelism. Consequently, the data throughput does not increase for faster network devices, i.e., it does not scale with the network.

To better exploit new hardware, the design of BDS must be revisited. This cannot be done with local optimizations for individual steps in the data processing core, but requires fundamental design changes [24]. Redesign is non trivial due to the complexity of the code bases of state-of-the-art BDS, e.g., with over 1.4 million lines of code in Hadoop MapReduce (HMR). Approaching this task with common parallel programming means, like threads, tasks or actors and their respective synchronization via locks, futures or mailboxes, inevitably increases code complexity even further. As a result, these systems become even harder to reason about, maintain and extend. We believe that a much more concise and scalable redesign can be achieved with new programming abstractions that enable a compiler to prepare the code for an efficient parallel execution at runtime. This paper represents first steps in this direction.

## 1.2 Contributions

In this paper, we present a rewrite for the processing core of current big data systems to increase data throughput, effectively improving scalability with new hardware. Our rewrite uses an implicit parallel programming language to provide concise code that is easy to maintain. The corresponding compiler transforms the program into a dataflow graph that the runtime system executes in a pipeline and task-level parallel fashion across the cores of a single cluster node. Hence, our approach extends the coarse-grained data parallelism in BDS with fine-grained pipeline and task-level parallelism inside the data processing cores to resolve the throughput bottlenecks.

The contributions of this paper are as follows:

(1) We contribute an analysis of the code base of HMR, Spark and Flink. This analysis shows that the different data processing cores are structurally equivalent and thus suffer the same scalability issues. Further, our study reveals design patterns that indicate pipeline-parallelizable code.
(2) As a case study, we change the HMR data processing core and present four different rewrites using an implicitly parallel language. Our rewrites are minimally invasive and reuse existing code. The rewritten code is concise and free of concurrency abstractions.
(3) We compare our data processing core to the original HMR implementation using parts of the TPC-H database benchmark. Experimental results report throughput speedups of up to 3.5x for compute-intensive configurations.

The rest of the paper starts with the analysis of state-of-the-art big data systems in Section 2. Afterwards, we give a brief introduction into implicitly parallel programming with Ohua (Section 3) and then present the HMR rewrites in Section 4. Section 5 compares our rewritten data processing core to the original one. Finally, we review related work in Section 6 and conclude in Section 7.

## 2 THE CORE OF BIG DATA PROCESSING

In this section, we study the three big data systems with a focus on the programming style of their data processing cores. This analysis serves to identify common code patterns across the different BDS that lend themselves well for a pipeline and task-level parallel execution. With this analysis we also provide concrete reasons for the scalability issues discussed above (cf. Section 1.1) and reported in [24]. This has only been supported by experimental observations but not by an in-depth analysis of the code structure. The first part of this section lists and explains the code of the data processing cores in these BDS. Then, Section 2.2 analyzes promising code patterns to exploit other forms of parallelism.

## 2.1 Data Processing: Code Study

We investigate the three most-widely-used systems in todays big data domain, Hadoop MapReduce (HMR), Spark and Flink.

*2.1.1 Hadoop MapReduce.* The MapReduce programming model consists of two functions: a `map` function that pre-processes the data and a `reduce` function that performs an aggregation [4]. The `map` function is executed in a data parallel fashion on every chunk of the input data. The results are partitioned and dispatched to nodes that execute the `reduce` code. The number of `reduce` tasks depends on the number of partitions defined by the application.

Figure 2a lists the data processing core of the framework in the `run` method. The main concept is the abstraction of a context through which data is retrieved and emitted. The `Context` class implements an iterator-like interface that is used in the `run` method to drive the input side, i.e., retrieve the data. In case of the `Mapper`, it retrieves one key-value pair at a time and passes it to the `map` function (Lines 14–17). The implementation of the `Reducer` looks almost identical, expect for the fact that it receives a list of values with each key. We omit it here for brevity and concentrate in the rest of the paper on the `Mapper` keeping in mind that our rewrites apply in the same way to the `Reducer`. The `map` function emits data via the `context`, driving the output side of the processing in the task (Lines 8–9). The `context` implementation uses the abstractions of the `InputFormat` and `OutputFormat` to interface with the various big data storage systems such as HDFS [20] and HBase [2, 5].

*2.1.2 Spark.* Spark [25] provides a new API which is heavily based on Scala and much closer to the original higher-order functions `map`, `reduce` and `filter`. In Spark, these functions, also called transformations, are applied to Resilient Distributed Datasets (RDD), the key abstraction in Spark. RDDs are immutable and therefore transformations create new RDDs. Like HMR, Spark partitions the input data and the intermediate results (if possible) to apply coarse-grained data parallelism.

Internally, Spark distinguishes among two types of tasks: the `ShuffleMapTask` and the `ResultTask`. All transformations are executed on a `ShuffleMapTask`, except the one performed on the final RDD in the program. A `ShuffleMapTask` first performs transformations and finally shuffles the result data over the network. Lines 15–18 in Figure 2b list the corresponding implementation code. Spark groups transformations to compose a pipeline that executes in a single task. Each transformation applies a function to all key-value pairs of the partition/RDD. While doing this, it

```
public class Mapper<KEYIN, VALUEIN,
                    KEYOUT, VALUEOUT> {
  /* The default implementation
     is the identity function. */
  protected
  void map(KEYIN key, VALUEIN value,
           Context ctxt) {
   ctxt.write((KEYOUT) key,
              (VALUEOUT) value);
  }

  public
  void run(Context ctxt) {
   while (ctxt.nextKeyValue())
     map(ctxt.getCurrentKey(),
         ctxt.getCurrentValue(),
         ctxt);
  }}
```

**(a) Hadoop**

```
private[spark] class
ShuffleMapTask(partitionId: Int,
               partition: Partition)
 extends Task[MapStatus] {

 override
 def runTask(ctxt: TaskContext)
 :MapStatus = {
 /* Deserialization and init of
    variables omitted for brevity. */
  var writer: ShuffleWriter[Any, Any] =
   manager.getWriter[Any, Any](
        dep.shuffleHandle,
        partitionId, ctxt)
 writer.write(
  rdd.iterator(partition, ctxt)
     .asInstanceOf[
      Iterator[_<:Product2[Any, Any]]])
 writer.stop(success = true).get}}
```

**(b) Spark**

```
1  public class DataSourceTask<OT>
2   extends AbstractInvokable {
3   private
4    InputFormat<OT, InputSplit> format;
5   private Collector<OT> output;
6
7   @Override public
8   void invoke() throws Exception {
9    OT reuse =
10      serializer.createInstance();
11   while (!this.taskCanceled &&
12          !format.reachedEnd()) {
13    OT returned;
14    if((returned =
15         format.nextRecord(reuse))
16         != null)
17      output.collect(returned);
18   }}}
```

**(c) Flink**

**Figure 2: The data processing cores of Hadoop MR, Spark and Flink are all based on the abstraction of a context.**

uses the concept of an iterator to navigate over the whole set of key-value pairs in a partition. Spark chains iterators where each iterator performs one transformation and passes the results on to the next. The code that uses the outermost iterator instance moves one key-value pair through the whole transformation pipeline before acquiring the next. In the `ShuffleMapTask`, the `writer` shuffles the results over the network. This on-demand processing of data actually implements a lazy evaluation strategy such that actions like `take(n)` do not process the whole data but only the first *n* records requested. The `ResultTask` either sends the results back to the program executing the Spark query or persists them in the underlying storage system.

*2.1.3 Flink.* HMR and Spark were designed to perform computations over data stored in a distributed storage system, called batch processing. However, big data applications often gather and process data continuously which is referred to as data streaming. Therefore, Flink [1] extends Spark's programming model with a notion of time which in turn requires to incorporate stateful transformations into the runtime system. In order to do so, Flink builds on top of the dataflow execution model that is popular for massively parallel processing (MPP) in distributed database systems [8]. It derives a dataflow graph from a program where each function call is represented as a node, referred to as *operator*. An operator can be replicated for a data parallel execution if its private state and its incoming data streams can be partitioned.

Similar to Spark, Flink groups pipelines of operators working on the same data partition together into a chain of operators and executes them in a single task. Flink uses a push-based model to drive the pipeline while Spark builds on a pull-based approach. Therefore, the first operator in the pipeline retrieves the data and drives the rest of the pipeline. We list the code for the `DataSourceTask` in Figure 2c. Like HMR, it uses an `InputFormat` abstraction to load data from distributed storage and pushes every element into a chain of collectors, i.e., operators (Lines 11–17). Each collector applies a reduction function to the pushed element and stores the result into its private state. A collector emits its private state to the next collector in the chain only when all values were processed and `close` was called by the previous collector.

## 2.2 Analysis

We now analyze these three implementations in terms of execution models, design patterns and their applicability to pipeline and task-level parallelism.

*2.2.1 Dataflow Inside.* All three systems are dataflow systems. They all build a directed acyclic graph (DAG) to represent the computation. The state of the nodes in the DAG is partitioned and replicas are created for each partition to introduce data parallelism. HMR implements a very coarse-grained system that allows to write a DAG with only the two nodes, `map` and `reduce`, that are executed in a massively parallel way. Higher-level systems and languages such as Hive (SQL) [22], Pig (Pig Latin) [6, 16] and IBM's SystemML (R) [7] derive more fine-granular DAGs and use HMR as a foundation to execute them. Spark has such a program analyzer that automatically converts the program into a DAG built-in. This allows Spark to inspect the dependencies and derive data pipelines for RDDs which are then run in a data parallel fashion across the RDD partitions. HMR with Hive or Pig, Spark and Flink analyze the DAG, use certain metrics to fuse nodes and map them onto tasks which are then executed in a single JVM process [9]. Hence, the dataflow execution model can be seen as the de-facto standard for big data processing engines.

However, the dataflow processing model has not been used to execute the data processing pipeline inside a task. Instead, parallelism is exploited very sparsely. HMR uses a dedicated thread on the output-side of the mapper in order to buffer and combine records before writing them to local disk. Furthermore, the reducer uses multiple threads to retrieve the map outputs for its partition over the network in parallel. The rest of the processing is performed

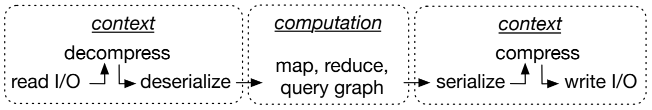


**Figure 3: The data processing core of all three BDSs.**

sequentially. While HMR executes each task on a new JVM process, Spark and Flink use a single JVM per cluster node to run tasks in parallel which speeds up task startup times significantly. However, for reduce tasks which are not executable in a data parallel fashion, even Spark and Flink do not benefit from a multi-core machine.

*2.2.2 Design Patterns and Parallelism.* Interestingly, although only reductions are purely sequential, a lot more code is executed sequentially. This is due to the composition of the data processing pipeline shown in Figure 3 which always consists of at least the following steps: retrieval of the data from I/O (disk or network), deserialization, computation, serialization and writing the data to I/O (disk or network). Decompression and compression can be dynamically added via configuration parameters. Furthermore, in HMR (Hive, SystemML), Spark and Flink, the computational part, i.e., the actual operations applied to the data, often encompasses more than a single function. To enable this dynamic composition of the processing core, the code uses the iterator and observer design patterns. The iterator allows chaining transformations in Spark while the observer pipelines operators in Flink. Furthermore, all three BDS integrate the various file formats to access the vast landscape of big data storage systems. For this, they all rely on the `InputFormat` and `OutputFormat` abstractions that were defined in earlier HMR versions. The `InputFormat` defines an iterator while the `OutputFormat` uses an observer, i.e., data is pulled from and pushed to I/O. The iterator and observer design patterns are duals [15] and we use them as an indicator in the code to derive parallelism. Both enforce encapsulation of state inside the concrete class. Hence, in between these two sequentially executing duals there exists a pipeline parallel execution which is semantically equivalent. Furthermore, we found that the (de-)compression and (de-)serialization steps can be executed in a task-level parallel fashion for a key and its value(s).

In the rest of the paper, we investigate how an implicitly parallel language would influence the code structure and scalability of such a data processing core. Although, we port only HMR to Ohua, this section made the case that Spark and Flink would benefit from the same rewrites.

## 3 IMPLICIT PARALLEL PROGRAMMING IN OHUA

This section introduces the main concepts of Ohua's language and runtime system using the simplest HMR rewrite as an example.

### 3.1 Algorithms and Stateful Functions

The Ohua programming model makes the following fundamental observation: An application is defined in terms of an *algorithm* that comprises smaller independent and self-contained functional building blocks. For example, in the HMR system, one algorithm defines the data processing for a mapper and another for the reducer. The associated functional building blocks include the functions to deserialize and decompress a single record in the data set. In Ohua, these functions may access their private state throughout the whole computation and are therefore called *stateful functions*.

We list the most coarse-grained algorithm implementation for the mapper in Figure 4a. We define the algorithm inside a Clojure function called `coarse`, because Ohua is an embedded domain specific language in Clojure[1]. We invoke this algorithm inside the `run` method of our new `Mapper` (see Figure 2a). The algorithm uses a higher-order map function (`smap`) to define a computation (Lines 8–12) that is applied to each of the records (Lines 5 and 13) in the data chunk to be processed by this `Mapper`. Note that we define this computation in terms of another algorithm named `compute-and-output`, i.e., algorithms may call other algorithms. In the following, we refer to this language as the *algorithm language*.

The overall algorithm uses the application-defined `Mapper` and HMR's `Context` abstraction. The iterator that is defined at Line 5 uses the `Mapper$Context` to retrieve one record at a time. The `hmr-map` function (Line 8) emits the list of key-value pairs produced for a single `map` invocation on the user-supplied `Mapper`. The `output` function (Line 10) passes one key-value pair at a time to the `Mapper$Context` and executes the output side of the pipeline. As an example of a functional building block for this algorithm, we list the implementation of the `output` function in Figure 4b. It is implemented as a Java class and identified as stateful function via the `@defsfn` annotation[2]. A call to `output` in a Java program differs from one in an Ohua algorithm:

| **Java** | **Ohua** |
|---|---|
| `o.output(k, v, writer);` | `(output k v writer)` |

The Java call needs an instance *o* of class `Output` which defines the state that is accessible inside the function. At runtime, the instance encapsulates this (private) state while the surrounding Java program needs to create and maintain it. In Ohua this instance is implicit, i.e., it is created and managed by the runtime system. For each function call Ohua creates such an instance once and reuses it throughout the entire computation. When the function performs side-effects against a field of the class in one call then these changes are visible to the succeeding calls. An example for such a state is the dictionary used in many compression algorithms. Note that the tried to keep our rewrite as minimal as possible without requiring edits to the HMR code base. As such, the class `Output` does not extend from `Context` but delegates to a context instance it receives from the surrounding program. This instance counts towards the state of `output`, including all transitive references such as for instance to the compression dictionary.

We define the Ohua language (constructs relevant to this paper) in Figure 5. The syntax is in line with that of Clojure. The language features variables, abstractions and applications for algorithms and lexical scoping of variables. The central contribution is the application of potentially stateful functions defined on the JVM in either Java, Scala or Clojure to variables. Program evaluation is entirely data driven just as known from any other functional

[1] Clojure primer: Function abstraction such as `(defn fun [arg 1 arg2] code)` is equivalent to `public Object fun(Object arg1, Object arg2){ /*code*/ }` in Java. Function application such as `(fun 4 5)` is equivalent to `fun(4, 5)`.

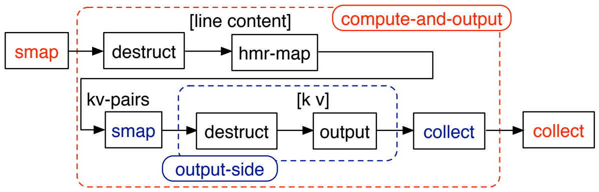
[2] Ohua additionally supports stateful function implementations in Scala and Clojure.

```
(defn coarse
  [^org.apache.hadoop.mapreduce.Mapper$Context reader
   ^org.apache.hadoop.mapreduce.Mapper mapper
   ^org.apache.hadoop.mapreduce.Mapper$Context writer]
  (let [records-on-disk (new InputIterator reader)]
    (ohua
      (smap
        (algo compute-and-output  [ [line content] ]
          (let [kv-pairs (hmr-map line content mapper)]
            (smap
              (algo output-side [ [k v] ] (output k v writer))
               kv-pairs)))
         records-on-disk))))
```

**(a) Coarse-grained algorithm for the data processing in HMR's mapper.**

```
public class Output {
  @defsfn public
  void output(Object key, Object value, Context ctxt) {
    ctxt.write(key, value); }}
```

**(b) Stateful function output.**



**(c) The dataflow graph that Ohua derives.**

**Figure 4: The data processing core of HMR implemented in Ohua. Figure 4a lists the code of the algorithm defined in the `Mapper` implementation of Figure 2a. The algorithm code is concise, i.e., free of clutter and concurrency constructs, and reuses the original implementation. For example, the stateful function `output` is written in Java (see Figure 4b) emits the key and its value via HMR's context abstraction. From the algorithm code, Ohua derives the dataflow graph in Figure 4c.**

*Terms:*

| | | | |
|---|---|---|---|
| $t$ | ::= | $x$ | variable |
| | \| | (algo $[x]$ $t$) | abstraction |
| | \| | $(t\ t)$ | application |
| | \| | (let $[x\ t]$ $t$) | lexical scope (variable binding) |
| | \| | (f $x_1 \dots x_n$) | apply JVM function f to $x_1\ \dots\ x_n$ with $n \geq 0$ |
| | \| | (if $t\ t\ t$) | conditionals |
| | \| | (seq $t\ t$) | sequential evaluation order (side-effect dependency) |

*Values:*

| | | | |
|---|---|---|---|
| $v$ | ::= | $o \in V_{\text{JVM}}$ | JVM value |
| | \| | (algo $[x]$ $t$) | abstraction |
| | \| | $[v_1 \dots v_n]$ | list of $n$ values |

*Predefined Functions:*

| | |
|---|---|
| (smap (algo $[x]$ $t$) $[v_1 \dots v_n]$) | apply abstraction to list |

**Figure 5: Definition of Ohua's algorithm language.**

language. To control the evaluation otherwise, the developer can either use conditionals or the higher-order function `seq`. With `seq`, the developer can express dependencies on I/O side-effects such as for example writing to disk or network communication. In the next section, we show that HMR relies a lot on such dependencies when reporting the mapper's progress, e.g., the bytes written to HDFS. A value is either an object $o$ from the domain $V_{\text{JVM}}$, i.e., JVM objects emitted by stateful functions, an algorithm abstraction or a list of values. We also predefine the function `smap`, Ohua's variant of the well-known higher-order function `map`.

## 3.2 Dataflow-based Runtime System

Despite incorporating state, Ohua can derive a parallel execution. To this end, the compiler first transforms the program expression into a form that binds each result of a stateful function call to a variable and then lowers this expression into a dataflow graph. On the left-hand side of Figure 6, we define the terms of our dataflow representation. The basic constructs of our dataflow representation are nodes, edges and ports. A node retrieves data via its input ports, performs a computation (step) and emit the result via its output ports. The data travels through the graph via edges where an edge originates at one output port and terminates at an input port. An output port may have multiple outgoing edges to pass the value to more than a single node while an input port always only connects a single incoming edge. We classify nodes according to their correspondence of retrieved and emitted data. A $1-1$ node retrieves at most one data item from any of its input ports to produce one data item to be emitted via its output ports. A $1-N$ node emits $n$ data items to its output port before it retrieves the next data item among its input ports. A $N-1$ node retrieves $n$ data items from any of its input ports before it produces a single output data item. Predefined $1-1$ dataflow nodes include `not`, the negation of a boolean value, `true`, the mapping of any value to a boolean true value, `ctrl`, the conversion of a boolean value into a control signal and `sel`, a selection. A selection node receives a choice on its lower input port that specifies from which of its input ports to forward the next data item. We also define nodes to work with list values. The $len-[]$ node has a $1-1$ correspondence. The $1-N$ node $[] \rightsquigarrow$ takes a list and emits the values one at a time, i.e., it streams the values. In order to perform the inverse $N-1$ operation, the $\rightsquigarrow []$ requires $n$, the number of elements in the result list, to be available on its upper input port. Finally, the node $[\rightsquigarrow$ streams the values from a list that is unbounded. An unbounded list resembles the idea of an iterator, i.e., a list where it is possible to walk over the values but where the actual size is only known once the last value was accessed. This is especially important when we retrieve data via I/O such as the records from the chunk file. As such, $[\rightsquigarrow$ additionally outputs the length of the list.

With these constructs defined, we translate the terms of our language from Figure 5 into dataflow on the right-hand side of Figure 6. Note that a stateful function call maps to a node with an edge to itself that transfers the state from one invocation to the other.

This is the common representation for state in dataflow models. We translate control flow into dataflow using `ctrl` node. In case of `seq`, the `ctrl` node always receive a positive input and sends a control signal that enables the downstream subgraph to proceed with the computation. In case of conditionals, the downstream subgraph to enable depends on the result of the subgraph for the condition. The translation of `smap` computations over bounded and unbounded lists is a straightfoward use of the according list-handling nodes. For brevity reasons, we omit the translation of access to variables in the lexical scope from within an expression passed to `if` or `smap` and refer the interested reader to [23]. To keep the visual presentation of the algorithms in their dataflow representation concise and clear, we omit state edges and translation details such as $len-[]$ nodes. For clarity reasons, we refer to $[] \rightsquigarrow$ and $[\rightsquigarrow$ as `smap` and to $\rightsquigarrow []$ as `collect`. For example, Figure 4c depicts the dataflow graph for the algorithm listed in Figure 4a. It contains the three coarse steps from Figure 3 that our analysis found: the first `smap` node retrieves the data via the context, the `hmr-map` function performs the computation and `output` uses again the context to emit the results. Note that streams inside the dataflow graph represent an opportunity for pipeline parallelism and independent nodes for task-level parallelism. A scheduler finally uses a thread pool to execute the graph.

## 4 REWRITING HMR'S MAPPER IN OHUA

This section presents the algorithms that we extracted from the original Hadoop MapReduce code and re-implemented in Ohua. We already listed a coarse-grained version for the map task in Figure 4a. In the following, we rewrite the steps before and after the `hmr-map` function call which include decompression, deserialization and their inverse operations. In HMR these steps are encapsulated in the `InputFormat` and `OutputFormat` classes that are instantiated in the `Context` of the map task. We focused our rewrite on the implementation for the `SequenceFile`[3] format which ships with HMR and is widely used. Figure 7a lists the algorithm for the map task. It applies the `data-ingress` algorithm to the raw bytes of a single key and its associated value as retrieved from disk (Lines 14–15). Note that this actually defines a computation for all key-value pairs retrieved via the network, i.e., data residing on remote disks, instead of from an in-memory data structure. The retrieved `line` and `content` objects are then input to the function call that runs the application-defined mapper (Line 16). The produced key-value pairs are finally input to the `data-egress` algorithm that handles one pair at a time (Lines 17–20). We define the `data-ingress` algorithm in Figure 7b. It decompresses the value bytes and returns the deserialized key and value. The `data-egress` algorithm in Figure 7c applies the inverse functions. It first serializes the objects in Lines 5 and 6 and then compresses the bytes at Line 7 before they are written to disk (Line 8). The rest of the stateful function calls concern HMR's status reporting and statistics, i.e., their execution order depends on their side-effects to the distributed file system. We use `seq` to encode these dependencies into the program.

We believe the Ohua code for the map task is concise. It only requires a few lines and omits implementation details. Instead, it allows the BDS developer to immediately understand the algorithms of the data processing core. Additionally, our rewrites use almost all of the existing code, i.e., we solely changed the composition of the data processing steps. Ohua's algorithm language allows to bind abstractions to variables. This establishes similar compositional flexibility that the iterator and the observer design patterns provide. We use it to define 4 programs with the following composition:

**Coarse (C)** uses the coarse-grained `Context` abstraction for the input and output side as shown in the algorithm of Figure 4a.

**Coarse-Input-Fine-Output (CIFO)** uses the `Context` for the input side and the fine-grained algorithm of Figure 7c for the output side.

**Fine-Input-Coarse-Output (FICO)** uses the fine-grained algorithm of Figure 7b for the input side and the `Context` for the output side.

**Fine (F)** uses the fine-grained algorithms for the input and output side.

Since Ohua inlines algorithm calls, this composition does not sacrifice parallelism. Figure 8 shows the dataflow graph for the Fine rewrite. It contains nodes for language constructs such as `seq` and `destruct` for destructuring tuples, arrays and lists[4]. The graph contains nested uses of `smap` and thus enough potential for a pipeline parallel execution. Additionally, deserialization, decompression and its inverse functions on the output side are independent for a key and its value and therefore can benefit from task-level parallelism.

## 5 EVALUATION

In this section, we evaluate the increase in throughput that can be achieved with our Ohua-based HMR rewrites. We first describe the setup for our experiments and then give breakdowns of the execution times of the individual stateful function calls of the program. Only these breakdowns allow to fully understand the speedup that can be achieved with Ohua's dataflow execution. Afterwards, we verify that Ohua achieves the maximal speedup possible for throughput and analyze the overheads of the Ohua-based execution.

### 5.1 Experimental Setup

In order to evaluate the data processing core of the HMR map task, we executed a single HMR map task on a 12- core (24 hardware threads) Intel NUMA machine at 2.6 GHz with 2 CPU sockets and 128 GB of RAM. All experiments used a JVM (JDK 1.8) with G1 enabled, 10 GB initial and 30 GB maximal heap size. A study of our new data processing core on the overall throughput of a job executed across multiple machines is future. This is due to that fact that normal benchmarks such as WordCount and Sort do not apply because of their simple data formats and their low-profile reduce phases. We studied the internals of Hive and found that the query execution engine does not use HMR's `InputFormat` and `OutputFormat` interfaces for data (de-)serialization. Hive developers moved these aspects into the computational part of HMR's data processing core in order to use the same code base for executing queries on HMR, Spark and Tez[5]. As such, we focus our evaluation on the data processing core directly. Instead of the Hadoop

[3]https://wiki.apache.org/hadoop/SequenceFile

[4]The actual graph also contains constructs such as `scope` for scoped variable usage and `size` to get the number of items a `collect` must gather for a single result.

[5]https://tez.apache.org/

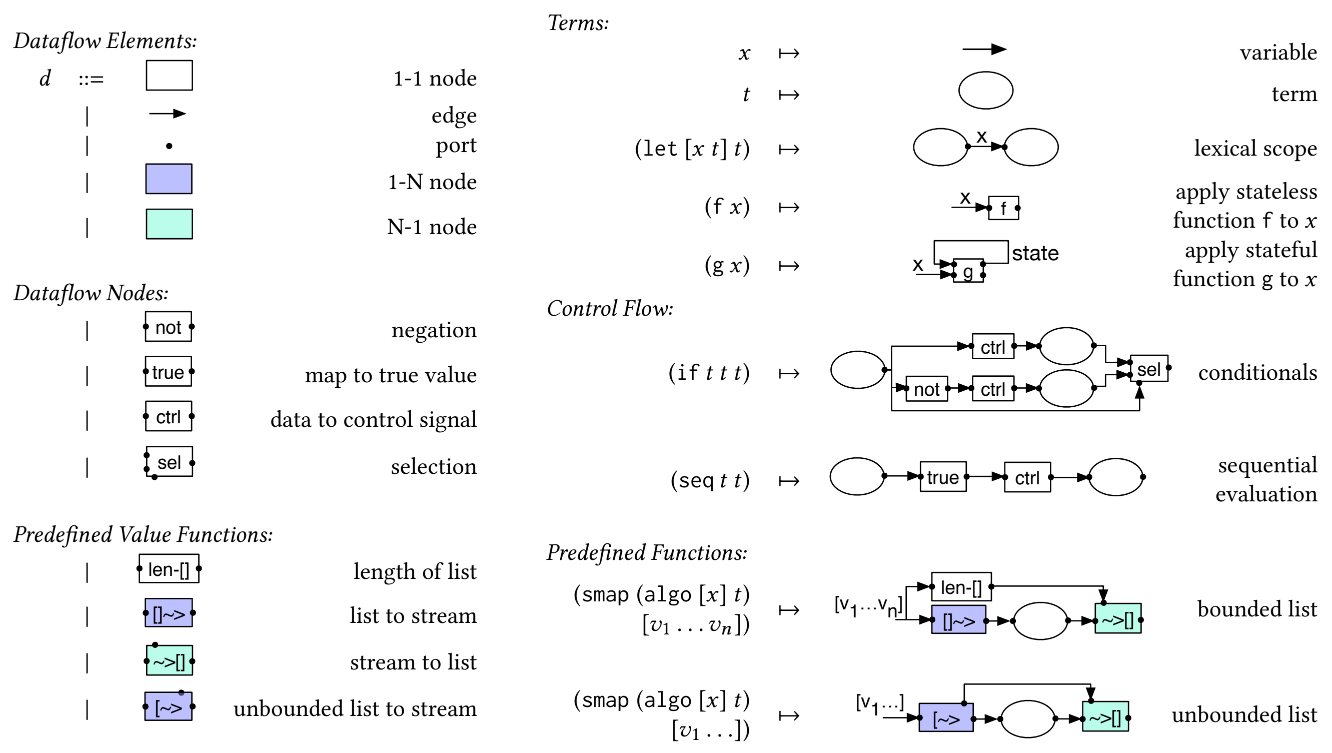

Dataflow Elements:
d ::=
1-1 node
edge
port
1-N node
N-1 node
Dataflow Nodes:
not
negation
true
map to true value
ctrl
data to control signal
sel
selection
Predefined Value Functions:
len-[]
length of list
[]~>
list to stream
~>[]
stream to list
[~>
unbounded list to stream
Terms:
x ↦ variable
t ↦ term
(let [x t] t) ↦ lexical scope
(f x) ↦ apply stateless function f to x
(g x) ↦ state apply stateful function g to x
Control Flow:
(if t t t) ↦ conditionals
(seq t t) ↦ sequential evaluation
Predefined Functions:
(smap (algo [x] t) [v1 ... vn]) ↦ bounded list
(smap (algo [x] t) [v1 ...]) ↦ unbounded list

**Figure 6: Definition of the dataflow representation (on the left-hand side) and the translation from the language terms to the dataflow representation (on the right-hand side).**

Distributed File System, we implemented our own Network File System (NFS) on Hadoop's file system abstraction to enforce data retrieval over the network. It retrieves a file chunk from local disk and transfers it over the network. This also happens during normal processing when the map task can not be executed on the node that hosts the input data chunk and hence enables complete code reuse on the input side. The requested file chunk for our experiments stores 1.4 GB of data. In order to achieve high bandwidth the NFS data provider transfers the file without copy overhead (`sendfile`) from the disk to the network while the map task executes on the same machine. To nevertheless study a portion of a real data analytics query, we retrieve and process randomly generated data for the `PART` table as defined in Figure 9 from the TPC-H benchmark [3]. Table records are stored in the widely used JSON format. We study the impact of Snappy and LZO compression in combination with either the default identity function in the mapper or a `map` function that applies a filter in a `WHERE` clause of an SQL query. For the latter we provide two versions: The first checks the conditions given in TPC-H query 19. The second implements a blacklist filter that tries to locate certain words in the `P_COMMENT` column.

## 5.2 Runtime Analysis

It has been observed that it is hard to analyze in detail the performance of current BDS, mainly due to a lack of support for instrumentation [17]. This lack of analysis support could explain why, for so many years, researchers stuck to the common believe that big data applications are network-bound. The modular structure of Ohua's dataflow graph makes such an instrumentation straight forward. We use it in the following to perform an in-depth analysis for the Fine rewrite. It allows us to understand the execution pattern of the data processing core and to analyze Ohua's runtime overheads. Afterwards, we study the throughput of our rewrites and the impact of maintaining state.

5.2.1 *Execution Pattern Analysis.* Figure 10 shows the total execution time of the stateful functions for the Fine rewrite. The stateful functions that facilitate constructs of the algorithm language are depicted on the left of each plot and prefixed with "ohua"[6]. The rest of the functions execute HMR functionality. The execution time of each function is broken down into the time spent in the Ohua operator framework code and the time spent in the actual stateful function. With this analysis, we can differentiate three aspects: 1) the overhead that comes with the algorithm language, 2) the runtime overhead of Ohua's operator framework and 3) the actual execution of the HMR functionality. The breakdown shows that most time is spent in (de)-serializing the value from and into JSON. Execution times vary slightly across the four configurations due to the fact that these functions create a lot of objects. Note that the execution time includes garbage collection. The more work other functions perform, the more likely it is that they are also interrupted by a garbage collection. Furthermore, the breakdown shows that Snappy requires much less CPU cycles than LZO, but LZO provides better compression rates. As a result, the configuration with

[6]Ohua implements language constructs as stateful functions as well.

```
(defn map-task
  [; decomposed input context
   ^SequenceFile$Reader reader ^Deserializer key-deserializer
   ^Deserializer val-deserializer
   ^Mapper mapper ; app-supplied map implementation
   ; decomposed output context
   ^Serializer key-serializer ^Serializer val-serializer
   ^Progressable reporter ^List stats
   ^FSDataOutputStream out ^Counter map-out-records
   ^Counter file-out-bytes ^CompressionCodec codec]
  (let [records (new NetworkDataIterator reader)]
    (ohua
      (smap
       (algo map-task-algo [[key-buf value-buf]]
        (let [[line content] (data-ingress key-buf value-buf
                                   key-deserializer val-deserializer)
                kv-pairs (hmr-map line content mapper)]
          (smap
           (algo kv-algo [kv-pair]
            (data-egress kv-pair reporter stats
                         key-serializer val-serializer codec
                         out map-out-records file-out-bytes))
            kv-pairs)))
       records))
```

**(a) Algorithm for the map task.**

```
(defalgo data-ingress [serialized-key serialized-val
                       key-deserializer val-deserializer]
  (let [key-in (deserialize serialized-key
                            key-deserializer)
        val-in (deserialize (decompress serialized-val)
                            val-deserializer)]
       (array key-in val-in)))
```

**(b) Algorithm for the data ingestion.**

```
(defalgo data-egress [kv-pair reporter stats key-serializer
                      val-serializer codec out
                      map-out-records file-out-bytes]
  (let [[k v] kv-pair
        bytes-before (seq (report-progress reporter)
                          (fs-stats-out stats))
        key-ser (seq bytes-before
                     (serialize k key-serializer))
        val-ser (seq bytes-before
                     (serialize v val-serializer))
        val-compressed (compress val-ser codec)
        bytes-after (seq (write key-ser val-compressed out)
                         (fs-stats-out stats))]
       (update-counters bytes-after bytes-before
                        map-out-records file-out-bytes)))
```

**(c) Algorithm for the data emission.**

**Figure 7: The modular rewrite for the HMR data processing core of the map task allows to construct four different scenarios. It facilitates to exchange the coarse-grained implementations of the `data-ingress` (see Figure 4) and `data-egress` algorithms for the fine-grained ones (see Figure 7b and 7c).**

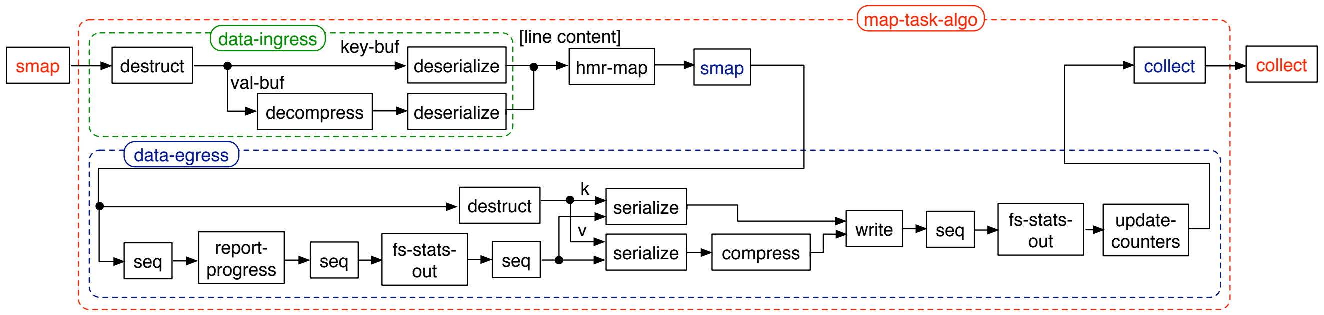


**Figure 8: The dataflow graph for the Fine rewrite expresses the inherent task-level parallelism in the decompression/deserialization of key and values. The same accounts for the inverse operations in the `data-egress` algorithm. The overall program uses two nested `smap` applications which directly translate into pipeline parallelism at runtime.**

```
CREATE TABLE PART ( P_PARTKEY     SERIAL PRIMARY KEY,
  P_NAME        VARCHAR(55),  P_MFGR      CHAR(25),
  P_BRAND       CHAR(10),     P_TYPE      VARCHAR(25),
  P_SIZE        INTEGER,      P_CONTAINER CHAR(10),
  P_RETAILPRICE DECIMAL,      P_COMMENT   VARCHAR(23) );
```

**Figure 9: TPC-H table for parts.**

LZO provides a higher potential to speedup the pipeline parallel execution than the one with Snappy because more processing can be performed in parallel. This potential further increases when the mapper performs actual work such as evaluating the conditions of TPC-H query 19 or our blacklist filter.

*5.2.2 Throughput.* Figure 11 depicts the throughput speedup for our four rewrites across each of the configurations. In all four configurations the Fine rewrite achieves the highest speedup with 3.5× for the LZO + Blacklist Filter configuration. Naturally, no configuration reaches the maximum theoretical speedup due to the overhead of the Ohua framework and the load profile of the different functions (cf. Figure 10). This is particularly notable in the case of Snappy + Identity, with a speedup below a maximum of around 2×[7]. In this configuration the overhead of the Ohua framework has

[7]Intuitively, the maximum speedup is calculated by determining the most appropriate pipeline, i.e., number of stages and stage balancing (load in Figure 10). For Snappy + Identity, with clearly two dominant functions of comparable load (deserialize-2, serialize-1), a two stage configuration would deliver around 2× speedup.

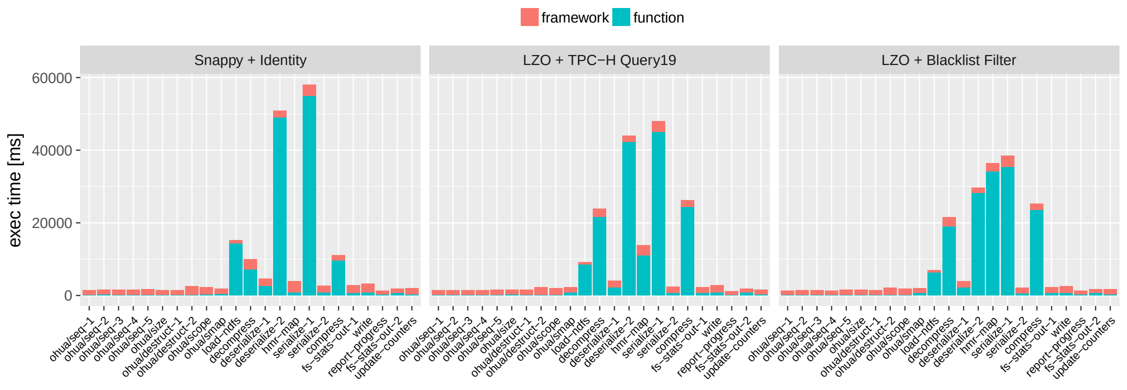


**Figure 10: Execution breakdowns for executions of the Fine rewrite.**

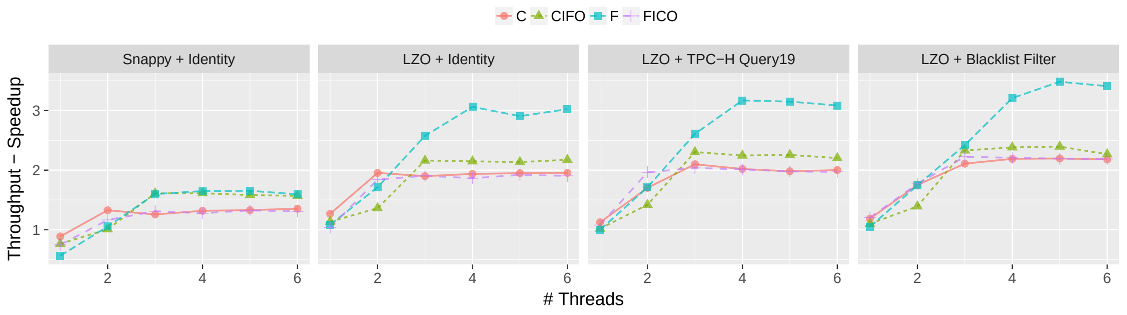


**Figure 11: Speedups in throughput of the rewrites in the four configurations.**

a higher impact on the throughput. The overall function execution is less than in other workloads while the overheads are the same in all configurations. The resulting performance penalty is roughly 0.4×, leading to maximum achievable speedup of 1.6×. In the other configurations this penalty is negligible with speedups around 0.1× below the maximum.

*5.2.3 Cost.* The COST metric refers to the number of additional cores that are necessary in order to achieve the same performance as the original implementation [14]. Figure 12 shows the COST results for the four configurations and compares a stateless implementation (as suggested by the internal HMR API) to a stateful one. Note that the original (coarse) parts are stateful. The graphs provide an important argument in favor for stateful computations and thus Ohua's programming model. The two functions with state are `decompress` and `compress` which both use a dictionary to speed up the discovery of words in the bytes. Removing this state can degrade the performance by as much as an order of magnitude, as in the case of our last two compute-heavy configurations.

## 6 RELATED WORK

We could not find any work that tries to make current big data systems scale with new hardware by introducing parallelism into their data processing cores. To the best of our knowledge there is no existing approach to do so using an implicitly parallel language. For example, Weld [18] applies compiler optimizations such as loop tiling and vectorization to speed up data analytics across different frameworks. It requires to re-implement main functions and operators to make them Weld-aware. Weld works on the application level rather than the data processing core of BDS. Crail [21] leverages RDMA and storage tiering to speed up the I/O functions (`load-ndfs` and `write` in the HMR data processing core). This works for sorting data but fails to scale as soon as additional analytics take place which require complex data types, their associated serialization, compression and perform additional operations on the data. Researchers also made the case of a single global cluster-wide address space to further speed up serialization and deserialization [12]. The Apache Spark project Tuncsten tries to remove garbage collection penalties via their own off-heap memory management similar to Flink [1]. However, all these approaches optimize only isolated parts of the data processing core, i.e., they shrink one bar in the execution breakdown graphs of Figure 10 just for another to become the bottleneck. They speed up individual parts instead of making the data processing core scale via an impacting structural change as researchers concluded [24]. In this paper, we do exactly this. We re-implemented the algorithms of the data processing core using an implicitly parallel language that can automatically exploit pipeline and task-level parallelism to scale the computation without increasing code complexity. Note that there exist frameworks such as Phoenix [19] or Metis [13] that scale analytics applications for in-memory data on multi- and many-core architectures. Although

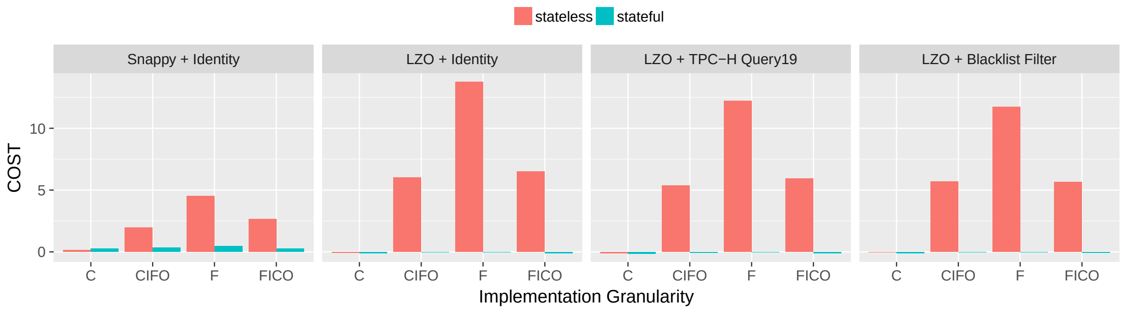


**Figure 12: COST of the four rewrites when implemented with or without state.**

Ohua can be used for that as well, this is not the topic of this paper. In this paper, we give a redesign of the data processing cores of the most-widely-used big data systems in the field today.

## 7 CONCLUSION AND FUTURE DIRECTIONS

The data processing core of current big data systems do not scale with improved network performance. An analysis of Hadoop MapReduce, Spark and Flink found that the implementations use either the iterator or the observer design pattern. Both provide the opportunity for a semantically equivalent pipeline parallel execution. We replaced these patterns in Hadoop MapReduce using Ohua, an implicitly parallel language. The resulting Ohua program heavily reuses the existing code, while being more concise. Furthermore, our evaluation shows that our rewrites provide speedups of up to 3.5x exploiting pipeline as well as task-level parallelism. In the future, we want to integrate a compiler-based approach that finds stateful functions that are applicable to a data parallel execution. This would enable us to make the data processing cores fully scalable independent of the size of the data processing pipeline.

## ACKNOWLEDGMENTS

The authors thank Andrés Goens for valuable discussions and the anonymous reviewers for their helpful comments. This work is supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden (CfAED)" and the Collaborative Research Center "Highly Adaptive Energy-efficient Computing (HAEC)".

## REFERENCES

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015).
[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data *(OSDI '06)*. USENIX Association, Berkeley, CA, USA.
[3] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.
[4] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters *(OSDI'04)*. USENIX Association, Berkeley, CA, USA.
[5] The Apache Software Foundation. 2017. Apache HBase. https://hbase.apache.org/. (2017). Accessed: 2017-03-22.
[6] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.* 2, 2 (Aug. 2009).
[7] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative Machine Learning on MapReduce *(ICDE '11)*. IEEE Computer Society, Washington, DC, USA.
[8] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System *(SIGMOD '90)*. ACM, New York, NY, USA.
[9] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4 (March 2014).
[10] Peng Jiang and X Shirley Liu. 2015. Big data mining yields novel insights on cancer. *Nat Genet* 47, 2 (02 2015), 103–104. http://dx.doi.org/10.1038/ng.3205
[11] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics *(ISCA '15)*. ACM, New York, NY, USA.
[12] Alexey Khrabrov and Eyal De Lara. 2016. Accelerating Complex Data Transfer for Cluster Computing *(HotCloud'16)*. USENIX Association, Berkeley, CA, USA.
[13] Yandong Mao, Robert Morris, and M. Frans Kaashoek. 2010. Optimizing MapReduce for Multicore Architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep* (2010).
[14] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost? *(HOTOS'15)*. USENIX Association, Berkeley, CA, USA.
[15] Erik Meijer. 2010. Subject/Observer is Dual to Iterator. (2010). http://www.cs.stanford.edu/pldi10/fit.html 2010 Conference on Programming Language Design and Implementation (PLDI), Fun Ideas and Thoughts Session.
[16] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing *(SIGMOD '08)*. ACM, New York, NY, USA.
[17] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks *(NSDI'15)*. USENIX Association, Berkeley, CA, USA.
[18] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.
[19] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems *(HPCA '07)*. IEEE Computer Society, Washington, DC, USA.
[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA.
[21] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017).
[22] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009).
[23] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From Functional Programs to Pipelined Dataflow Circuits *(CC 2017)*. ACM, New York, USA.
[24] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On the [Ir]Relevance of Network Performance for Data Processing *(HotCloud'16)*. USENIX Association, Berkeley, CA, USA.
[25] 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 *(NSDI'12)*. USENIX Association, Berkeley, CA, USA.