# Semantic Characterization of MapReduce Workloads

Zhihong Xu
University of Nebraska, Lincoln, NE
zxu@cse.unl.edu

Martin Hirzel
IBM Research, Yorktown Heights, NY
hirzel@us.ibm.com

Gregg Rothermel
University of Nebraska, Lincoln, NE
grother@cse.unl.edu

***Abstract*—MapReduce is a platform for analyzing large amounts of data on clusters of commodity machines. MapReduce is popular, in part thanks to its apparent simplicity. However, there are unstated requirements for the semantics of MapReduce applications that can affect their correctness and performance. MapReduce implementations do not check whether user code satisfies these requirements, leading to time-consuming debugging sessions, performance problems, and, worst of all, silently corrupt results. This paper makes these requirements explicit, framing them as semantic properties and assumed outcomes. It describes a black-box approach for testing for these properties, and uses the approach to characterize the semantics of 23 non-trivial MapReduce workloads. Surprisingly, we found that for most requirements, there is at least one workload that violates it. This means that MapReduce may be simple to use, but it is not as simple to use correctly. Based on our results, we provide insights to users on how to write higher-quality MapReduce code, and insights to system and language designers on ways to make their platforms more robust.**

## I. Introduction

Corporations collect large amounts of data during day-to-day operations, and analyzing this data yields insights that translate directly into monetary value. For many corporations, the tool of choice for performing such analyses is MapReduce, because it is simple, yet scales to large clusters of commodity machines. Massive parallel computing was formerly the domain of the HPC and DB communities, who are quick to point out that there is little new in MapReduce. But the reality remains that MapReduce, and particularly its free open-source implementation Hadoop, is widely adopted, and accounts for a large percentage of compute time on data centers of Google, Yahoo, Facebook, and others. In these contexts, MapReduce has been used to build search indices, analyze log files, extract information from large bodies of unstructured text, run graph algorithms on social networks, and perform many other data-intensive tasks.

Part of the appeal of MapReduce is that is is easy to adopt. To get started, a programmer merely writes two functions, Map and Reduce, and the platform takes care of parallelism, distribution, and fault tolerance. However, the platform makes several implicit assumptions about the semantics of user-written code. For instance, the Map function should be stateless and the Reduce function should avoid interference between different key partitions. If the user configures MapReduce to also use the Reduce function as a Combine function piggy-backed on the mapper, then the Reduce function needs to be associative. (We elaborate on these assumptions later in this paper.) If the user code violates these requirements, then depending on placement and scheduling decisions made at runtime, the program can yield arbitrary and potentially incorrect results.

MapReduce implementations such as Hadoop do not check semantic requirements on user code, because user code is written in general-purpose languages like Java or C++. One could argue that this situation is improved by higher-level programming languages that target MapReduce, such as Pig [20] or Jaql [2], because they satisfy the semantic requirements for their built-in operators. However, Pig, Jaql, and other languages of their sort still rely heavily on user code, and do not check that code for adherence to semantic requirements. While some static [13] and dynamic [19] analyses for MapReduce-like code provide a good starting point for doing this, to date that work provides only partial solutions.

Given that MapReduce relies on semantic properties of user code for correctness and performance, one would expect the foregoing issues to be well-known. However, it turns out that the community of MapReduce users and developers has little awareness of the problem. In fact, the required semantic properties are vague, and nobody knows how much user code out there satisfies them. Prior workload characterizations for MapReduce focus on performance, not semantics [4], [14]. Our paper fills this gap by providing a semantic workload characterization for MapReduce.

To provide a foundation for our approach, we formulate a set of semantic requirements for MapReduce in the form of six semantic properties: determinism, selectivity, statefulness, commutativity, partition isolation, and associativity. Then, we apply black-box test-case generation techniques to user code in an attempt to assess whether these properties hold. To do this, we gathered a suite of 23 non-trivial MapReduce workloads, some of which contain multiple Map and Reduce functions. The core of this paper is a detailed empirical study of the semantic properties of each Map or Reduce function in each of these workloads. The empirical study shows that most things that user-code can do wrong, it does do wrong in at least one workload. In other words, properties required for the correct execution of MapReduce are routinely violated. The lesson for the user is to be aware of these properties during development and testing. The lesson for the system or language designer is to avoid relying too heavily on unchecked semantic properties of user code. Instead, MapReduce systems and languages would do well to add automated support for making programs more robust.

This paper makes the following contributions:

- The first empirical study of semantic properties in MapReduce workloads, using 23 non-trivial applications.
- Formulations of semantic properties and assumptions for MapReduce.
- Interpretations of the empirical findings that can help users and language designers utilize MapReduce correctly, and provide better programming systems to support it.

978-1-4799-055-3/13/$31.00 ©2013 IEEE

The remainder of this paper is structured as follows. Section II provides background information on MapReduce applications, and defines several important semantic properties of Map and Reduce functions. Using these properties, Section III describes the assumptions about Map and Reduce functions used in Hadoop applications. Section IV presents the testing framework that we use to determine whether properties hold. Section V presents the details of our study design. Sections VI and VII analyze our study data and discuss implications of our results, respectively. Section VIII discusses related work, and Section IX concludes.

## II. Background

To characterize MapReduce workloads at a semantic level, we first need to understand the structure of these workloads, the signatures of user-defined functions in that structure, and the semantic properties with which said functions can be characterized.

### A. MapReduce Applications

Fig. 1 shows the general structure of a MapReduce application. The application begins by reading inputs from a distributed file system. The *map* stage consists of many data-parallel workers executing a user-defined Map function. This is followed by an optional *combine* stage. The combine stage, if present, is piggy-backed on the map stage, using cheap local communication. Next comes the *shuffle* stage, which forms a complete bipartite graph of network communication. The data sent across these links consists of key-value pairs, and the shuffle stage ensures that if two key-value pairs have the same key, they arrive at the same worker. Next comes a *sort* stage, which sorts all keys for a given partition. The sort stage is shown in gray in this figure, because whereas map, combine, and reduce are user-defined, sort is built-in. Finally, the *reduce* stage consists of many data-parallel workers executing a user-defined Reduce function, and the output is written back out to the distributed file system.

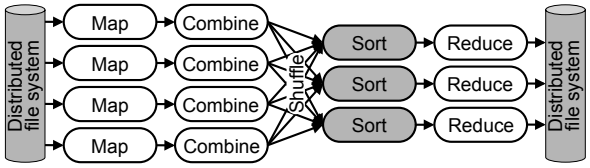


Fig. 1. MapReduce data flow graph

### B. Signatures for Map and Reduce

For the study described in this paper we used the Hadoop implementation of MapReduce. In Hadoop, the distributed file system is the Hadoop File System (HDFS), and the user-defined functions are written in Java. The rest of this paper sometimes uses "Hadoop" as synonymous with "MapReduce"; the results we obtain for Hadoop should generalize to other MapReduce systems.

Before listing semantic properties, we describe the general structure of Map and Reduce functions. When building a Hadoop application, the developer needs to write a Map class with a Map function, and a Reduce class with a Reduce function. The signature for a Map function is:

```
public void Map(KEYIN key, VALUEIN value,
                Context context)
```

As this signature shows, a Map function takes three parameters, key, value, and context. Their concrete types are defined by users. Usually, the input for the map stage is a file, and each line of this file becomes a key-value pair. We refer to each key-value pair as a *data item*. Every time a Map function is called for a data item, we call this a firing. A firing of the Map function submits zero or more output data items by invoking a callback on the context parameter:

```
context.write((KEYOUT) key, (VALUEOUT) value);
```

The signature for a Reduce function is:

```
public void Reduce(KEYIN key, Iterable<VALUEIN> values,
                   Context context)
```

The Reduce function also takes three parameters, two of which are the same as for the Map function. The signatures differ with respect to the second parameter. There is a list of values for Reduce functions while for Map functions, there is only a single value. As for the Map function, the concrete parameter types are defined by users.

We consider properties of Reduce functions at two levels. One is the *coarse-grained* level, where the input to a firing consists of a key and a list of values. In other words, the firing corresponds to the entire call to the Reduce function. The other level is the *fine-grained* level, where the input to a firing consists of a key and a single value. In other words, the fine-grained level breaks the Reduce function call into multiple separate firings, one for each value in the list. After each firing, no matter at what level, Reduce functions may submit output data items using the same callback as for Map functions: context.write((KEYOUT) key, (VALUEOUT) value).

There are several reasons for analyzing Reduce functions at two levels. On the one hand, the coarse-grained level may seem more natural, because it corresponds to a function call. On the other hand, the fine-grained level is more consistent with how Map functions are treated, because both handle a single key-value pair at a time. Analyzing Reduce functions at two levels also provides more information. For example, a given Reduce function may be stateful from one value to the next at the fine-grained level, but stateless from one list to the next at the coarse-grained level.

### C. Properties

This section defines six semantic properties of Map or Reduce functions. We selected these properties because they affect the correctness and the performance of MapReduce applications. As we shall see in Section III, the properties form the foundation for assumptions about MapReduce. Some of the properties (such as associativity) appear prominently in the MapReduce literature. Other properties (such as determinism) usually remain implicit in the literature, but their effect on application semantics is no less fundamental. We are not aware of additional properties that carry similar importance to the six we chose for this study for MapReduce.

*1) Determinism:* Determinism characterizes the repeatability of all firings of a function across entire runs of a MapReduce application. Given the same inputs, a deterministic function produces the same outputs on any run. A function that

generates different outputs for different runs given the same sequence of inputs to its firings is non-deterministic.

*2) Selectivity:* Selectivity characterizes the number of data items produced by a function per firing. If a function has selectivity $> 1$ for at least one firing, it is *prolific*. If a function always has selectivity $\leq 1$ for all firings and $< 1$ for at least one firing, it is *selective*. If a function always has selectivity 1 for all firings, it is *one-to-one*.

*3) Statefulness:* Statefulness characterizes the repeatability of individual firings of a function within a single run of a MapReduce application. Given the same input, a stateless function produces the same outputs on any firing. If a function's outputs are affected by historical inputs, it is stateful.

*4) Commutativity:* A function is commutative if a change in the order in which input data items are sent to it does not change its outputs. As shown in Fig. 1, MapReduce sorts the outputs of the map stage before sending them to the Reduce function. Thus, for Map functions, when assessing commutativity, we compare outputs after sorting them.

*5) Partition-isolation:* Partitioning uses the key parameter that is provided to the Map or Reduce function. In a partition-isolated function, output data items for firings with one key are not affected by firings with different keys. If firings with different keys can affect one another, the function is partition-interfering.

*6) Associativity:* We check associativity only for Reduce functions. Let k be a key, and let x and y be lists of values. A function R is associative if the following holds for all k, x, and y:

R(k, R(k, x) + R(k, y)) = R(k, x+y)

For simplicity, we assume that each function returns its outputs instead of sending them via a Context callback. If R is an associative Reduce function, users can also use R as a combiner, as shown in Fig. 1.

The six properties that we have presented are not altogether independent. Statelessness implies commutativity and partition-isolation. However, the reverse is not true: a stateful function may or may not also be commutative and partition-isolated. Therefore, the properties are worth establishing separately.

## III. Semantic Assumptions

This section presents the assumptions regarding the foregoing properties that underlie the usage of Map and Reduce functions in Hadoop applications.

### A. Assumptions about Map Functions

The Map function is expected to be **deterministic**. Determinism is useful for testing and debugging, since a deterministic application can be run any number of times with the same output, and the actual output of a run can easily be compared to a stored expected output. Furthermore, MapReduce comes with a fault-tolerance mechanism that recovers from a failed worker machine by repeating its tasks on a different machine. The implicit assumption behind this mechanism is that tasks yield the same results the second time around.

The **selectivity** of the Map function is expected to be **one-to-one**. This is not required for correctness – the framework is general enough to safely handle selective or prolific Map functions. However, it is still useful to know about the selectivity of Map functions, because this has performance implications: it determines the amount of network communication in the shuffle, and the load on the reducers. In functional programming languages, Map is a higher-order function also known as *apply-to-all*. In that paradigm, Map is one-to-one.

The parallelization scheme for MapReduce requires the Map function to be **stateless**. If the Map function is stateless, it is safe to chop the input after any key-value pair, and assign chunks of inputs to any map workers, without affecting the result. If the Map function is stateful, the user must instead carefully arrange which key-value pairs to store in which input files. Such data invariants make applications brittle.

The Map function is expected to be **commutative** modulo the ordering of its outputs. In other words, the order of an input sequence may affect the order, but not the values, of an output sequence. Commutativity is important for parallelization. If the Map function is commutative, then the parallel scheduler for MapReduce can assign input chunks to map workers in any order.

Similarly, the Map function should be **partition-isolated**. Partition isolation supports parallelization, because it implies that any input chunk can be assigned to any map worker without concern for interference by data items with different keys.

### B. Assumptions about Reduce Functions – Coarse-Grained Level

The Reduce function is assumed to be **deterministic** for the same reasons the Map function is. Determinism helps with testing, debugging, and fault-tolerance, and overall leads to cleaner semantics.

At the coarse-grained level, for each firing on a key and list of values, the Reduce function is expected to yield exactly one key-value pair as output. In other words, its **selectivity** should be **one-to-one**. Just like the selectivity of the Map function, the selectivity of the Reduce function does not affect correctness, but is relevant to performance. It affects the output disk bandwidth and footprint, and the data size for downstream consumers (looking beyond where Fig. 1 ends). In functional programming languages, Reduce is a higher-order function also known as *fold* or *aggregate*. In that paradigm, it turns one list of input values into one output value. This has been dubbed the *iterator-based* approach to aggregation [24].

The Reduce function should be **stateless** at the coarse-grained level. This may appear counter-intuitive at first glance, because reduction typically involves stateful aggregation across values. However, that reduction state is necessary only between values within a key, while the coarse-grained level considers each key with its entire list of values for a firing. Stateless reduction at the coarse-grained level is needed for the parallelization scheme to be correct; it enables the MapReduce scheduler to assign any task to any reduce worker.

At the coarse-grained level, the Reduce function should be **commutative** modulo output order; that is, the order in

which pairs of keys and lists of inputs are submitted to the Reduce function must not affect output values, only output order. At the coarse-grained level, we are not concerned with the order of values within each list. Commutativity is implied by statelessness, but even in the stateful case it has benefits, because it allows the parallelization scheme to assign tasks to reduce workers in any order.

The Reduce function should be **partition-isolated**: no state from one key interferes with processing of another key. Partition isolation is needed for parallelization; it allows the parallel scheduler to freely assign any key to any reduce worker.

The Reduce function should be **associative** if and only if the Reduce function is also used as the combiner (see Fig. 1). If the Reduce function is used as a combiner but is not associative this is incorrect, because it implies that the result depends on the chunking and worker assignment. On the other hand, if the Reduce function is associative but is not used as a combiner this is fine for correctness, and affects only performance. By not using a combiner, the application loses an optimization opportunity, since all data flows over the shuffle before reduction.

### C. Assumptions about Reduce Functions – Fine-Grained Level

The Reduce function should be **deterministic** at the fine-grained level, just as at the coarse-grained level.

At the fine-grained level, the Reduce function should be **selective**. In other words, when the input for a key is presented one value at a time, then only the last value for that key produces an output. As with all selectivities in MapReduce, this does not affect correctness, only performance. The fine-grained level is the perspective of a function called at each step during the reduction. MapReduce does not require users to incrementalize their functions in this way, but this *accumulator-based* approach to aggregation is common in other systems, where it can be used for optimizations [24].

At the fine-grained level, the Reduce function is expected to be **stateful**. The result for a key is expected to depend on all values in the list for that key, which means that when each value is handled by a separate firing, the Reduce function must remember some state between firings. A Reduce function that is stateless at the fine-grained level is not necessarily incorrect, but that case is so uncommon that it is suspicious: it may indicate that the system is not being used in the way in which it was designed. For instance, such functionality might be better situated in the Map function, thus saving the overhead of a shuffle and an extra computation stage.

Assumptions for **commutativity** at the fine-grained level are subtle. By default, the sort stage shown in Fig. 1 sorts only the keys within a partition, not the values within a key [7]. That implies an assumption that Reduce be commutative at the fine-grained level, because otherwise the results are scheduling-dependent. However, a common practice of Hadoop users is to request a *secondary* sort of the values within a key. When the secondary sort is present, Reduce need not be commutative at the fine-grained level. The inventors of MapReduce could have chosen to make their system more resilient by making the secondary sort the default.

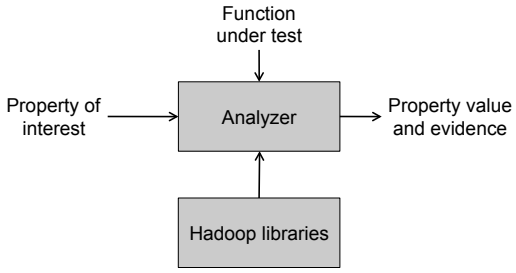


Fig. 2. Testing framework

The MapReduce system guarantees **partition-isolation** by construction at the fine-grained level, because the Reduce function handles all values for a given key together as a single list. Also, there is no separate assumption about **associativity** at the fine-grained level, because the definition of associativity directly uses multiple lists of values for the same key. Hence, we do not analyze either partition-isolation or associativity at the fine-grained level.

## IV. Dynamic Analysis

We now describe the testing framework that we use to determine whether or not properties hold. We describe how to instrument a function-under-test (FUT) to form a test program for our analysis. Next, we describe how each property is tested. Finally, we discuss test input generation techniques. In related work [23], we built a similar testing framework for a different domain (streaming analytics using SPL [12] instead of batch analytics using MapReduce [7]).

### A. Testing Framework

Figure 2 shows our testing framework. The main component is the Analyzer, which reads the property that users want to test, along with an FUT, which is a user-written Map or Reduce function. The Analyzer instruments the FUT such that when it executes, it generates information that can be checked later. Then the Analyzer generates inputs relevant to the property and executes the function with the inputs using the Hadoop libraries. Instrumentation results are sent back to the Analyzer, which checks whether the property holds. If the property is violated, the Analyzer presents users with evidence indicating the violation. If the property has not yet been shown to be violated, the Analyzer generates additional test inputs until it reaches a technique time limit, and on reaching that time limit, it reports to the user that the property "potentially" holds. The Analyzer can also report statistics on the extent of the evidence provided for a potentially holding property.

While we use dynamic analysis to check the properties, an alternative approach would have been to use static analysis. Both approaches are valid, and yield different trade-offs. Dynamic analysis can observe only the behavior of specific test runs, whereas static analysis can generalize over all possible runs. Conversely, static analysis may be imprecise and consider situations that are infeasible in practice. Since the main contribution of this paper is the empirical study, rather than the analysis approach, we chose to be pragmatic and select dynamic analysis.

TABLE I. WORKLOAD INFORMATION

| Application | Source | Lines of code | Functions map | Functions red. | Functionality |
|---|---|---|---|---|---|
| **MultiFileWordCount** | Hadoop | 178 | 1 | 1 | Count words from several files. |
| **QuasiMonteCarlo** | Hadoop | 208 | 1 | 1 | Estimate $\pi$ using Monte-Carlo method. |
| **RandomTextWriter** | Hadoop | 651 | 1 | 0 | Write 10GB of random textual data per node. |
| **RandomWriter** | Hadoop | 185 | 1 | 1 | Write 10GB of random data per node. |
| **SecondarySort** | Hadoop | 161 | 1 | 1 | Sort by two keys, a primary and a secondary. |
| **Sort** | Hadoop | 143 | 1 | 1 | Sort the data written by the random writer. |
| **WordCount** | Hadoop | 60 | 1 | 1 | Count the words in the input files. |
| **Anagrams** | GitHub | 79 | 1 | 1 | Find all the anagrams in the given data corpus. |
| **ApacheLogAnalyzer** | GitHub | 224 | 1 | 1 | Extract statistics from Apache access log file. |
| **CustomKey** | GitHub | 255 | 1 | 1 | Count integer pairs in a file. |
| **CVSPairThreshold** | GitHub | 80 | 1 | 1 | Read a file, looking at pairings above a certain threshold. |
| **Dictionary** | GitHub | 92 | 1 | 1 | Concatenate different translations for the same word. |
| **FacebookBuzzCount** | GitHub | 107 | 1 | 1 | Count "likes" on FaceBook for each college. |
| **Geolocation** | GitHub | 91 | 1 | 1 | Group articles in Wikipedia by their GEO location. |
| **ReduceSideJoin** | GitHub | 119 | 2 | 1 | Simple example of reduce side join. |
| **ScoreFriends** | GitHub | 273 | 4 | 2 | Pipeline of multiple MapReduce jobs that starts with a score and a list of friends for each user, and sorts each list by decreasing scores. |
| **UserAccessCount** | GitHub | 69 | 1 | 1 | Find user access information from log files. |
| **FarmerMarket** | YouTube | 99 | 1 | 1 | How good are a city's farmer's markets? |
| **Canopy** | Mahout | 170,913 | 3 | 1 | Group objects into clusters. |
| **Dirichlet** | Mahout | 170,913 | 2 | 1 | Performs Bayesian mixture modeling. |
| **FuzzyKMeans** | Mahout | 170,913 | 4 | 2 | Discover soft clusters where a particular point can belong to more than one cluster with certain probability. |
| **KMeans** | Mahout | 170,913 | 3 | 1 | Group objects into clusters. |
| **MeanShift** | Mahout | 170,913 | 4 | 1 | Mean Shift clustering. |

### B. Testing Each Property

*1) Determinism:* The Analyzer generates an input and runs it twice. Then the Analyzer compares the outputs from the two executions. If they differ, the Analyzer concludes that this FUT is definitely non-deterministic, returning the input as evidence. Otherwise, it tries again with more inputs. If it reaches the time limit, the FUT is probably deterministic.

*2) Selectivity:* The Analyzer generates an input, and checks the output after each firing. If the output contains more than one data item, the FUT is definitely prolific. Otherwise, the analyzer keeps generating inputs until the time limit is reached. At this point, if any firing had produced 0 data items, the FUT is probably selective, otherwise, it is probably one-to-one.

*3) Statefulness:* For simplicity, the Analyzer tests statefulness with an input in which all data items are the same. It checks the output after each firing to see if it differs from the output from the previous firing. If it is different, the FUT is definitely stateful. Otherwise, the Analyzer keeps trying new inputs until reaching the time limit. It then concludes that the FUT is probably stateless.

*4) Commutativity:* The Analyzer generates an input for the FUT. It executes the FUT with different permutations of the input, one by one. The Analyzer checks the output after each permutation to see if it differs from the output of the previous permutation. If there is any difference, the FUT is definitely non-commutative. Otherwise, the Analyzer tries again with another input and its permutations. If the time limit is reached without encountering evidence for non-commutativity, the function is probably commutative.

*5) Partition-isolation:* The Analyzer generates an input in which all data items have the same key. It runs the FUT on this input and saves the result. Then, it randomly inserts data items into the input that have a different key. It runs the FUT again and compares the results. If the results differ, the pair of inputs is evidence for partition-interference. Otherwise, the Analyzer keeps trying new inputs until reaching the time limit. It then concludes that the FUT probably satisfies partition-isolation.

*6) Associativity:* We check associativity only for Reduce functions. The Analyzer generates an input consisting of a list $I$ of data items with the same key. It splits the list into two parts, $I_1$ and $I_2$, such that $I_1 + I_2 = I$. Then, it runs the FUT four times, to obtain four results: $O = FUT(I)$, $O_1 = FUT(I_1)$, $O_2 = FUT(I_2)$, and $O_C = FUT(O_1 + O_2)$. If $O \neq O_C$, the function is definitely not associative. Otherwise, the Analyzer tries a different input. If the Analyzer reaches the time limit, it reports that the FUT is probably associative.

### C. Test Case Generation

Our testing framework relies on test case generation techniques, many of which could be viable in our context. For example, search-based test case generation techniques [1], [21] and constraint-based test case generation techniques [3], [5], [11], [16] yield successful results in many scenarios. This work, however, uses a simpler approach, involving random test case generation [9]. In random test case generation, for each input to the FUT, values for that input are randomly chosen within the range of possible values for that input given its type. We chose this approach because it is easy to apply, widely used, and works well in our case.

## V. EMPIRICAL STUDY

To evaluate whether MapReduce applications meet the paradigm's design requirements, we conducted an empirical

TABLE II. PROPERTY TESTING RESULTS FOR MAP FUNCTIONS

| Application | Class containing Map function | Deterministic? | Selectivity | Stateful? | Commutative? | Partition isolated? |
|---|---|---|---|---|---|---|
| **MultiFileWordCount** | MapClass | D | prolific | - | C | P |
| **QuasiMonteCarlo** | QmcMapper | D | prolific | - | C | P |
| **RandomTextWriter** | RandomTextMapper | - | prolific | S | - | - |
| **RandomWriter** | RandomMapper | - | prolific | S | - | - |
| **SecondarySort** | MapClass | D | 1 : 1 | - | C | P |
| **Sort** | Mapper | D | 1 : 1 | - | C | P |
| **WordCount** | TokenizerMapper | D | prolific | - | C | P |
| **Anagrams** | AnagramMapper | D | 1 : 1 | - | C | P |
| **ApachLogAnalyzer** | StatisticsMapper | D | selective | - | C | P |
| **CustomKey** | CustomIntPairMapper | D | selective | - | C | P |
| **CVSPairThreshold** | Map | D | prolific | - | C | P |
| **Dictionary** | AnagramMapper | D | 1 : 1 | - | C | P |
| **FacebookBuzzCount** | FacebookMapper | D | prolific | - | C | P |
| **Geolocation** | GeoLocationMapper | D | selective | - | C | P |
| **ReduceSideJoin** | ReduceSideJoinUserNameMapper | D | 1 : 1 | - | C | P |
| | ReduceSideJoinUserLogsMapper | D | 1 : 1 | - | C | P |
| **ScoreFriends** | EmitFriendsMapper | D | prolific | - | C | P |
| | ReduceSideJoin_1_UserScoreMapper | D | selective | - | C | P |
| | ReduceSideJoin_1_FriendsUserMapper | D | selective | - | C | P |
| | SecondarySort_2_UserFriendMapper | D | selective | - | C | P |
| **UserAccessCount** | Map | D | 1 : 1 | - | C | P |
| **FarmerMarket** | MapClass | D | 1 : 1 | - | C | P |
| **Canopy** | InputMapper | D | selective | - | C | P |
| | ClusterClassificationMapper | D | selective | - | C | P |
| **Dirichlet** | InputMapper | D | selective | - | C | P |
| **FuzzyKMeans** | InputMapper | D | selective | - | C | P |
| | ClusterClassificationMapper | D | selective | - | C | P |
| **KMeans** | InputMapper | D | selective | - | C | P |
| | ClusterClassificationMapper | - | selective | S | - | - |
| **MeanShift** | InputMapper | D | selective | S | - | - |
| | MeanShiftCanopyCreatorMapper | D | 1 : 1 | S | - | - |
| | MeanShiftCanopyClusterMapper | D | 1 : 1 | - | C | P |

study. As objects of study we chose 23 MapReduce applications, which were composed of 38 Map functions and 24 Reduce functions. Table I provides information on these applications. The first column lists the name of the application, and the second column lists its source. The next two columns list the numbers of lines of code (counted using `cloc` version 1.56), and the numbers of Map and Reduce functions in each application. The rightmost column provides a brief description of the functionality of each application.

These object programs come from three sources. Seven are provided as samples with the Hadoop package – in this case, Hadoop version 0.21. Eleven come from an online code source site (GitHub) and from tutorials found on YouTube. The remaining five are from Apache Mahout version 0.8. Mahout implements core machine learning algorithms such as clustering, classification, and batch-based collaborative filtering on top of Apache Hadoop using the MapReduce paradigm [17].

Most of the object programs we use are self-contained; they range in size from 60 to 651 lines of code. The Mahout applications make extensive calls to the Mahout library, from which they are difficult to separate. Therefore, for the Mahout programs, we included the entire library (170,913 lines of code) in the line count.

As mentioned in Section IV, the Analyzer uses a time limit, and it can be set to any time. We used five minutes, which worked well for our experiments and corresponds to roughly 100 tests. We manually checked all analysis results and found only one case where the answer was incorrect: for some Map functions, input files need empty lines to expose the fact that these functions are selective. However, the files generated by our Analyzer contain no empty lines, so it misclassifies functions whose selectivity depends on empty lines as one-to-one. We report the manually corrected results. This manual step could have been avoided by letting the Analyzer generate empty lines in the test files.

## VI. RESULTS

This section focuses on objective results, deferring interpretations of those results to Section VII.

Table II displays the results of applying our analysis approach to Map functions in our object programs. Whereas Table I lists 38 Map functions, Table II lists only 32 Map functions. The other six functions are omitted because they do not generate any output, and therefore their semantic properties cannot be tested for. Rather than using the context.write API described in Section II, these Map functions store intermediate results in attributes of their Map classes. Then, at the end of the map stage, the Map classes use a Cleanup function to finish the computation and produce an output to be passed on to the reduce stage.

TABLE III. PROPERTY TESTING RESULTS FOR REDUCE FUNCTIONS AT THE COARSE LEVEL

| Application | Class containing Reduce function | Deterministic? | Selectivity | Stateful? | Commutative? | Partition isolated? | Associative? | Combiner? |
|---|---|---|---|---|---|---|---|---|
| **MultiFileWordCount** | IntSumReducer | D | 1 : 1 | - | C | P | A | Y |
| **RandomWriter** | Reducer | D | prolific | - | C | P | A | N |
| **SecondarySort** | Reduce | D | prolific | - | C | P | - | N |
| **Sort** | MyReducer | D | prolific | - | C | P | A | N |
| **WordCount** | IntSumReducer | D | 1 : 1 | - | C | P | A | Y |
| **Anagrams** | AnagramReducer | D | selective | - | C | P | - | Y |
| **ApachLogAnalyzer** | StatisticsReducer | D | 1 : 1 | - | C | P | - | N |
| **CustomKey** | CustomIntPairReducer | D | 1 : 1 | - | C | P | A | N |
| **CVSPairThreshold** | Reduce | D | selective | - | C | P | A | Y |
| **Dictionary** | AnagramReducer | D | selective | - | C | P | - | N |
| **FacebookBuzzCount** | IntSumReducer | D | 1 : 1 | - | C | P | A | Y |
| **Geolocation** | GeoLocationReducer | D | 1 : 1 | - | C | P | - | N |
| **ReduceSideJoin** | ReduceSideReducer | D | prolific | - | C | P | - | N |
| **ScoreFriends** | ReduceSide_1_Reducer | D | prolific | - | C | P | - | N |
| | SecondarySort_2_Reducer | D | prolific | - | C | P | A | N |
| **UserAccessCount** | Reduce | D | selective | - | C | P | A | N |
| **FarmerMarket** | Reduce | D | selective | - | C | P | A | N |
| **Canopy** | CanopyReducer | D | prolific | - | C | P | - | N |
| **Dirichlet** | CIReducer | D | 1 : 1 | - | C | P | - | N |
| **Kmeans** | CIReducer | D | 1 : 1 | - | C | P | - | N |
| **FuzzyKMeans** | CanopyReducer | D | prolific | - | C | P | - | N |
| | CIReducer | D | 1 : 1 | - | C | P | - | N |
| **MeanShift** | CanopyReducer | D | prolific | - | C | P | A | N |

As the table shows, of the 32 Map functions, three are non-deterministic. Eight functions are prolific, ten are one-to-one, and 14 are selective. In addition to the three non-deterministic functions, we found two other stateful functions. All stateful Map functions in our set of object programs were also non-commutative and partition-interfering.

Table III displays analysis results obtained on Reduce functions at the coarse-grained level. Whereas Table I shows 24 Reduce functions, Table III shows only 23 functions. One is omitted because (as above) it produces no output and cannot be tested for semantic properties. Instead, it stores intermediate results in attributes of the Reduce class, which then calculates the ultimate output from the Cleanup function.

As the table shows, all Reduce functions in our object programs are deterministic. At the coarse-grained level, nine Reduce functions are prolific, nine are one-to-one, and five are selective. All Reduce functions are stateless, commutative, and partition-isolated. There are 11 Reduce functions that are associative. In seven cases, the Reduce functions are associative, but the applications do not use a combiner stage. There is one case in which the Reduce function is not associative, but the application uses it in a combiner stage.

Table IV displays analysis results obtained on Reduce functions at the fine-grained level. As discussed in Section IV, we check only four properties at this level: determinism, selectivity, statefulness, and commutativity. All Reduce functions are deterministic at the fine-grained level. Only one function is prolific at the fine-grained level, six are one-to-one, and 16 are selective. Four functions are stateless at the fine-grained level, while the rest are all stateful. Eight of the functions are commutative, and the others are non-commutative.

## VII. DISCUSSION AND IMPLICATION

We now provide additional insights into the results of our study, and comment on its implications.

### A. Assumptions vs. Realities for Map Functions

This section considers each of the assumptions applicable to Map functions.

*1) Determinism:* Map functions are assumed to be deterministic. However, our analysis discovered three Map functions that are non-deterministic. Two of these are from examples in the Hadoop package. These two Map functions use random-number generators to produce synthetic test data for experimentation when no real-world data is available. The third non-deterministic Map function is in the Mahout package. This Map function does not use random-number generators directly; instead, it uses an attribute of the Map class that is initialized with a random number before the Map function is first called. Specifically, this happens in the context of a clustering algorithm, which begins with random clusters that are later refined based on actual input data. In all three non-deterministic Map functions, the non-determinism affects the output not just of the Map stage, but of the entire application.

All three of these Map functions have been implemented to be non-deterministic on purpose. This demonstrates that there exist cases, such as for data-generation or for seeding a machine-learning algorithm, where users consciously decide that they do not need determinism. An implication for users is that in such cases, the applications become more difficult to test and debug, because their output is not reproducible. Even if each local random-number generator were deterministic, the overall result would also depend on task assignments, scheduling, and even fault recovery: after a worker fails, the system schedules the task to run again, but the results are all

TABLE IV. PROPERTY TESTING RESULTS FOR REDUCE FUNCTIONS AT THE FINE LEVEL

| Application | Class containing Reduce function | Deterministic? | Selectivity | Stateful? | Commutative? |
|---|---|---|---|---|---|
| **MultiFileWordCount** | IntSumReducer | D | selective | S | C |
| **RandomWriter** | Reducer | D | 1 : 1 | - | - |
| **SecondarySort** | Reduce | D | prolific | - | - |
| **Sort** | MyReduce | D | 1 : 1 | - | - |
| **WordCount** | IntSumReducer | D | selective | S | C |
| **Anagrams** | AnagramReduce | D | selective | S | - |
| **ApachLogAnalyzer** | StatisticsReducer | D | selective | S | C |
| **CustomKey** | CustomIntPairReducer | D | selective | S | C |
| **CVSPairThreshold** | Reduce | D | selective | S | C |
| **Dictionary** | AnagramReducer | D | selective | S | - |
| **FacebookBuzzCount** | IntSumReducer | D | selective | S | C |
| **Geolocation** | GeoLocationReducer | D | selective | S | - |
| **ReduceSideJoin** | ReduceSideReducer | D | selective | S | - |
| **ScoreFriends** | ReduceSide_1_Reducer | D | selective | S | - |
| | SecondarySort_2_Reducer | D | 1 : 1 | - | - |
| **UserAccessCount** | Reduce | D | selective | S | C |
| **FarmerMarket** | Reduce | D | selective | S | C |
| **Canopy** | CanopyReducer | D | 1 : 1 | S | - |
| **Dirichlet** | CIReducer | D | selective | S | - |
| **Kmeans** | CIReducer | D | selective | S | - |
| **FuzzyKMeans** | CanopyReducer | D | 1 : 1 | S | - |
| | CIReducer | D | selective | S | - |
| **MeanShift** | CanopyReducer | D | 1 : 1 | S | - |

but certain to contain different random numbers. That said, in most cases, the user-written function is deterministic, and it is useful for the MapReduce system to strive for determinism at the application level too.

*2) Selectivity:* MapReduce is inspired by functional languages with higher-order functions, where the Map function is traditionally one-to-one. However, only 10 of the 32 Map functions we tested are one-to-one. There are 14 cases in which Map functions are selective. In these cases, when an input item is empty, the Map function generates no output. In other words, instead of representing missing values by some representation of null, these Map functions simply omit missing values. There are also eight cases in which Map functions are prolific. In six of these eight cases, the functions are prolific because of the format of the input files utilized. If there is more than one element in one line of an input file, these Map functions split them and send them to output one by one. In other words, these functions produce multiple output data items for a single input data item. The remaining two prolific functions are the non-deterministic synthetic input generators from the Hadoop package. These input generators are configured with the number of bytes written to output per firing. The default setting is 1024 bytes, so in each firing, data is written to context until the limit is reached. This makes these Map functions prolific. By tweaking this setting, which is stored in an attribute of the Map class, the same Map functions can also be made selective or one-to-one.

As mentioned in Section III, selectivity does not affect the correctness of applications, but it does affect performance. Based on the roots of MapReduce in functional languages, one would in theory expect the output of the map stage to have the same size as its input. In practice, however, Map functions are often not one-to-one. If the Map function is selective, then the combine stage is less important, because the main purpose of the combine stage is to reduce the volume of data to be sent over the shuffle. On the other hand, if the Map function is prolific, the combine stage, and possibly other optimizations as well, are more important. For instance, if the data volume after the map stage is large, it may help to introduce pipeline parallelism between the map and reduce stages, as proposed by MapReduce Online [6].

*3) Statefulness:* Map functions are assumed to be stateless. However, we found five Map functions that are stateful. Three of these are stateful due to non-determinism as discussed above. The other two have an attribute in their Map classes with a counter. Each firing increments the counter, and each output data item includes the counter as part of its value. Therefore, these two functions are not only stateful, but their statefulness is visible in the output of the map stage. This means that the counts affect the sort stage and the reduce stage.

MapReduce always parallelizes the Map function and distributes it across multiple workers. However, this is not safe to do for stateful Map functions, in the sense that the result of a parallel run will differ from the result of a sequential run. In fact, the computation is not repeatable from one parallel run to the next, because it depends on scheduling. The lesson for users is that something as innocuous as a counter can jeopardize the determinism guarantees of an entire system, and should therefore be avoided. The lesson for language and system designers is that a trust-but-verify approach may be useful: let users decide whether they want to write stateful functions, but perform some analysis to warn about inadvertent cases.

*4) Commutativity:* Map functions are assumed to be commutative, but our analysis found five non-commutative Map functions. These are the same five functions that were found to be stateful, including three non-deterministic functions and two functions that maintain state with a counter. Map functions

need to be stateless for safe parallelization. In the stateful case, commutativity could at least partially shield the application from unsafe parallelization, because the result would at least be robust relative to input order. However, this second line of defense did not occur in any of the workloads we looked at.

*5) Partition-isolation:* Map functions are assumed to be partition-isolated. The Map functions that are stateful and non-commutative are also partition-interfering. The same reasons apply here; the randomness and counter information cause the output of one key to be affected by different keys. As discussed under commutativity above, MapReduce parallelization is fully safe only for stateless Map functions. Like commutativity, partition-isolation can also serve as a second line of defense, because it shields the application from some scheduler decisions. However, we did not observe this effect in practice.

### *B. Assumptions vs. Realities for Reduce Functions, at the Coarse-Grained Level*

This section considers each of the assumptions applicable to Reduce functions at the coarse-grained level.

*1) Determinism:* Reduce functions are assumed to be deterministic at the coarse-grained level and in all cases observed in our experiment they are.

*2) Selectivity:* Reduce functions are assumed to be one-to-one at the coarse-grained level. However, we found nine prolific Reduce functions. In most of these cases, the functions generate one output item for each data item in the values list. In one case, the Reduce function always generates a split line before generating other outputs when it is called for a key. We also found five selective Reduce functions: for example, one of them computes a sum, but only outputs it if it is non-zero.

Given that prolific and selective Reduce functions do exist, the system needs to take them into consideration for performance. Our findings validate the design choice made by the inventors of MapReduce to put the output in a distributed file system, because it is not always reduced to a small amount of data. Also, if the Reduce function is prolific, it can be counter-productive to use it as a combiner, because a combiner tries to reduce traffic on the shuffle, and does not serve that purpose when prolific.

*3) Statefulness:* Our study confirms the assumption that Reduce functions are stateless at the coarse-grained level.

*4) Commutativity:* Our study confirms the assumption that Reduce functions commute at the coarse-grained level.

*5) Partition-isolation:* Our study confirms the assumption that Reduce functions satisfy partition-isolation at the coarse-grained level.

*6) Associativity:* Our applications include 11 Reduce functions that are associative, and 12 that are not. For those that are associative, seven involve applications that do not use a combiner. Of the non-associative Reduce functions, only one is used as a combiner. We investigated this case and found that the use of the non-associative Reduce function in the combiner does not affect the behavior of the application significantly; it affects only the position and the number of commas in its output, which are used to split the real output data. One application, CVSPairThreshold, uses both an associative Reduce function and a combiner, but the combiner employs a different function than Reduce. The combiner only summarizes the information for a key, while the reducer also does that and in addition controls the final output by applying a threshold to the summarized information.

The foregoing discussion has two implications. First, if the Reduce function is associative, users should have a combiner stage using the same Reduce function to help improve performance, because the Hadoop system is designed to include a combiner stage. Second, when functions are not associative, users cannot use the same Reduce functions in the combiner stage. In the case we observed, even though the use of a non-associative Reduce function in the combiner stage did not affect the results significantly, it still means that results are not precisely reproducible.

### *C. Assumptions vs. Realities for Reduce Functions, at the Fine-Grained Level*

This section considers each of the assumptions applicable to Reduce functions at the fine-grained level.

*1) Determinism:* As at the coarse-grained level, Reduce functions were all observed to be deterministic at the fine-grained level.

*2) Selectivity:* Reduce functions are assumed to be selective at the fine-grained level. However, we found six Reduce functions that are one-to-one and one that is prolific. Those that are one-to-one generate one output item for each input item in the list of values. The one case that is prolific at the fine-grained level is also prolific at the coarse-grained level. As mentioned above, this Reduce function generates a split line when it is called. Therefore, for the first data item, there are two output data items, which render this Reduce function prolific. Comparing the results of checking selectivity at the two levels, if a Reduce function is one-to-one or prolific at the fine-grained level, it must be prolific at the coarse-grained level. If it is selective at the fine-grained level, it could be either one-to-one or selective at the coarse-grained level.

Not all Reduce functions are selective at the fine-grained level. By comparing selectivity at both the coarse-grained and the fine-grained levels, we obtain a more differentiated view than if we were to look at only one or the other. The fine-grained results reinforce the conclusions from earlier that one should not expect the reduce stage to always decrease the amount of data. Systems and applications must plan accordingly to handle possibly large outputs.

*3) Statefulness:* Reduce functions are assumed to be stateful at the fine-grained level. However, we found four Reduce functions that are stateless. Two of these simply implement an identity function: each element of the list of values is copied unmodified into the output data item, together with the corresponding key. At first glance, an identity reducer may seem useless, but keep in mind that MapReduce also performs a sort stage in the reduce workers, and using an identity reducer while specifying a secondary sort key is a common trick for obtaining just the sorting. The other two stateless Reduce functions perform simple transformations one value at a time, but without remembering state across values. No historical information is used, so these functions are stateless. Comparing

the selectivity results with the statefulness results, we see that if a Reduce function is selective, it is probably stateful. This type of Reduce function summarizes the information on all data items in the list of values and writes the final result to output.

*4) Commutativity:* There are no assumptions made about the commutativity of Reduce functions at the fine-grained level. Stateless functions are commutative by default. There are eight Reduce functions that are stateful and commutative; these summarize information from input items. However, the other 11 stateful functions are not commutative. The order in which inputs are received is important in these cases, because they concatenate some information from each data item together and send the concatenated content out. This suggests it would be a good habit for users to enable Hadoop's secondary sorting (of values within a key). Alternatively, designers of new systems could make this behavior the default.

## VIII. Related Work

Dean and Ghemawat's [7] seminal paper on MapReduce mentions some semantic properties of user-defined functions. The paper points out that the overall application is deterministic if the Map and Reduce functions are stateless. It describes how the system takes care of partitioning and sorting; as we discuss, doing so relaxes the requirements for partition-independence and commutativity. The paper also mentions that if Reduce is associative, it can be used as a combiner. Our paper clarifies these properties and assumptions, and characterizes to what extent they hold in workloads.

Lämmel [15] provides an essay on how to emulate MapReduce in the Haskell functional programming language using higher-order functions (HOFs). Among other things, Lämmel reiterates the role of associativity, and points out that if the Reduce function is commutative, there are additional opportunities for optimization. Instead of rephrasing the inner workings of MapReduce in Haskell, we characterize MapReduce workloads implemented in Hadoop. Our paper explores which semantic assumptions hold in practice.

Kavulya et al. [14] characterize traces of Hadoop jobs running on a research cluster and Chen et al. [4] characterize traces of Hadoop jobs running on production clusters. These papers study empirical data on job completion times, data set sizes, and burstiness of a job mix. Unlike these papers, we look at code, not traces, enabling us to explore more high-level semantic properties. However, even at the level of traces, Chen et al. are able to characterize selectivity, finding that in some jobs, the Map function aggregates data and is selective, and in other jobs, the Reduce function expands data and is prolific. Our semantic characterization confirms and explains these surprising results about selectivity, as well as exploring several additional properties.

Olston et al. describe a dynamic analysis [19] that generates examples for Pig Latin [20], a programming language for MapReduce. These examples show sample data at each stage of a dataflow graph to help humans understand the behavior of their application. Instead of concrete low-level example data, we find high-level semantic properties. Furthermore, while their paper mostly focuses on describing a tool for workload characterization, our central contribution is an empirical study that characterizes several real-world workloads.

Our own related work [23] describes a dynamic analysis for dataflow operators and applies it to the SPL streaming language [12]. Unlike the present paper, the prior paper does not consider MapReduce programs, and it does not provide a workload characterization. Instead, it formalizes properties in first-order logic, and explores a variety of test case generation techniques that can be used to check those properties on SPL operators.

Hueske et al. [13] invented a static program analysis for user-written code in Stratosphere, a MapReduce-like system. They focus on read-sets and write-sets, which are useful for operator reordering, and on selectivity, which we also explore. On the other hand, they do not analyze determinism, statefulness, or partition-isolation, and instead silently assume that those properties hold. Our paper differs in that we use dynamic analysis instead of static analysis, work on MapReduce instead of Stratosphere, and cover a broader set of semantic properties.

There is a growing literature on semantic workload characterization for widely-adopted languages and features. These papers formed part of the inspiration for our work, since they have benefited the research community by identifying pain-points in current designs. They encompass characterizations of workloads for the C preprocessor [10], Java [8], JavaScript [22], and R [18]. Like our work, they focus on semantics, but unlike our work, they do not address MapReduce.

## IX. Conclusions

This paper characterizes the semantics of 23 MapReduce applications. The characterization checks commonly-held assumptions of what properties user-written Map and Reduce functions should satisfy. For example, one might expect that Reduce reduces the data volume, by summarizing multiple input data items into a single output data item. Likewise, Map is assumed to be stateless, since otherwise, the parallel and fault-tolerant MapReduce execution yields non-deterministic outputs. We find that these assumptions are commonly violated in practice. In some cases, a violated assumption constitutes a bug, and we offer suggestions for fixing it. In other cases, a violated assumption is arguably less important or even intentional. We discuss implications from the perspective of users, system implementers, and language inventors.

Characterization of workloads is one use case for our testing framework, and this paper both illustrates how such characterizations can be obtained and provides data on a substantial set of applications. A second use case for our testing framework, explored initially in [23], involves the testing of Map and Reduce functions by engineers who create them. In this use case, engineers can utilize test cases that they create, or test cases generated by approaches such as the one we utilize in our study, to assess whether their operators possess necessary properties for the given situations in which they are employed. In future work we intend to continue to study this approach.

## Acknowledgements

This work was supported in part by the AFOSR through award FA9550-09-1-0129 to the University of Nebraska -

Lincoln. Kun-Lung Wu contributed to early stages of the work. Portions of this work were performed by the first author during the course of an internship at IBM Research. We thank Avraham Shinnar for insightful discussions about some of the more esoteric Hadoop features.

## REFERENCES

[1] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 108–118, 2004.

[2] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Very Large Data Bases (VLDB)*, pages 1272–1283, 2011.

[3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Computer and Communications Security (CCS)*, pages 322–335, 2006.

[4] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. In *Very Large Data Bases (VLDB)*, pages 1802–1813, 2012.

[5] Lori A. Clarke. A system to generate test data and symbolically execute programs. *Transactions on Software Engineering (TSE)*, 2(3):215–222, 1976.

[6] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2010.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[8] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, 2003.

[9] Joe W. Duran and Simeon Ntafos. A report on random testing. In *International Conference on Software Engineering (ICSE)*, pages 179–183, 1981.

[10] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *Transactions on Software Engineering (TSE)*, 28(12):1146–1170, 2002.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[12] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development (IBMRD)*, 57(3/4):7:1–7:11, 2013.

[13] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. In *Very Large Data Bases (VLDB)*, pages 1256–1267, 2012.

[14] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production MapReduce cluster. In *Cluster, Cloud, and Grid Computing (CCGrid)*, pages 94–103, 2010.

[15] Ralf Lämmel. Google's MapReduce programming model — revisited. *Science of Computer Programming (SCP)*, 70(1):1–30, 2007.

[16] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON. In *Embedded Software (EMSOFT)*, pages 299–306, 2005.

[17] Mahout official website. http://mahout.apache.org/.

[18] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 104–131, 2012.

[19] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *International Conference on Management of Data (SIGMOD)*, pages 245–256, 2009.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[21] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability (JSTVR)*, 9(4):263–282, 1999.

[22] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[23] Zhihong Xu, Martin Hirzel, Gregg Rothermel, and Kun-Lung Wu. Testing properties of dataflow program operators. In *Conference on Automated Software Engineering (ASE)*, 2013.

[24] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.