DiffStream: Differential Output Testing for Stream Processing Programs

KONSTANTINOS KALLAS*, University of Pennsylvania, USA
FILIP NIKSIC*, University of Pennsylvania, USA
CALEB STANFORD*, University of Pennsylvania, USA
RAJEEV ALUR, University of Pennsylvania, USA

High performance architectures for processing distributed data streams, such as Flink, Spark Streaming, and Storm, are increasingly deployed in emerging data-driven computing systems. Exploiting the parallelism afforded by such platforms, while preserving the semantics of the desired computation, is prone to errors, and motivates the development of tools for specification, testing, and verification. We focus on the problem of differential output testing for distributed stream processing systems, that is, checking whether two implementations produce equivalent output streams in response to a given input stream. The notion of equivalence allows reordering of logically independent data items, and the main technical contribution of the paper is an optimal online algorithm for checking this equivalence. Our testing framework is implemented as a library called DiffStream in Flink. We present four case studies to illustrate how our framework can be used to (1) correctly identify bugs in a set of benchmark MapReduce programs, (2) facilitate the development of difficult-to-parallelize high performance applications, and (3) monitor an application for a long period of time with minimal performance overhead.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Information systems** → *Stream management*; • **Theory of computation** → *Online algorithms*.

Additional Key Words and Phrases: stream processing, differential testing, runtime verification

ACM Reference Format:
Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: Differential Output Testing for Stream Processing Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 153 (November 2020), 29 pages. https://doi.org/10.1145/3428221

1 INTRODUCTION

In recent years, a dramatic increase in real-time data processing needs has led to the popularity of high performance distributed stream processing systems, such as Flink [Carbone et al. 2015], Samza [Noghabi et al. 2017], Spark Streaming [Zaharia et al. 2013], Storm [Apache 2019], and Twitter Heron [Kulkarni et al. 2015]. While these systems achieve high performance through aggressive parallelization of data operators, programs written in such frameworks are nevertheless inherently concurrent and consequently prone to errors. As a result, there is a growing need for

*Equal contribution.

Authors' addresses: Konstantinos Kallas, University of Pennsylvania, Philadelphia, PA, 19104, USA, kallas@seas.upenn.edu; Filip Niksic, University of Pennsylvania, Philadelphia, PA, 19104, USA, fniksic@seas.upenn.edu; Caleb Stanford, University of Pennsylvania, Philadelphia, PA, 19104, USA, castan@cis.upenn.edu; Rajeev Alur, University of Pennsylvania, Philadelphia, PA, 19104, USA, alur@cis.upenn.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART153
https://doi.org/10.1145/3428221

tools that make it easier to write correct programs in such systems, through specification, testing, and verification.

In a broader context, the problem of ensuring correctness of distributed programs is long established, as can be seen by the significant amount of past and ongoing work on the correctness of distributed protocols (e.g., [Abdulla et al. 2014; Chand et al. 2016; Padon et al. 2016]), concurrent data structures (e.g., [Burckhardt et al. 2014; Herlihy and Wing 1990]), and distributed systems (e.g., [Hawblitzel et al. 2015; Ozkan et al. 2018; Wilcox et al. 2015]). In general, because this body of work addresses the challenges of verification of distributed protocols and low-level primitives, it targets experts who design and implement complex distributed systems. However, data-parallel programming frameworks, such as stream processing systems, aim to offer a simplified view of distributed programming where low level coordination and protocols are hidden from the programmer. This has the advantage of bringing distributed programming to a wider audience of end-users, and at the same time it requires new tools that *can be used by such end-users* (rather than just experts) to automatically test for correctness.

Unfortunately, there is limited work in testing for stream processing programs; in fact, the state of the art in practice is unit and integration testing [Vianna et al. 2019]. In order to bridge this gap and provide support for checking correctness to end-users, we focus on the problem of *differential testing* for distributed stream processing systems, that is, checking whether the outputs produced by two implementations in response to an input stream are equivalent. Differential testing [Evans and Savoia 2007; Groce et al. 2007; McKeeman 1998], allows for a simple specification of the correct program behavior, in contrast to more primitive testing techniques, where the specification is either very coarse (i.e. the application doesn't crash) or very limited (i.e. on a given input, the application should produce a specific output). More precisely, differential testing allows for a reference implementation to be the specification. This is especially useful in the context of distributed stream processing systems, since bugs introduced due to distribution can be caught by comparing a sequential and a distributed implementation. In addition, having a reference implementation as a specification allows testing with random inputs, since there is no need to specify the expected output.

We identify two critical challenges in testing stream processing programs. The first challenge is dealing with output events that are out-of-order due to parallelism. In particular, for differential testing, two implementations might produce events in different rates, asynchronously, and out-of-order. However, the order between specific output events might not affect the downstream consumer (e.g. events with timestamp less than t can arrive in any order, as long as they arrive before the watermark t), therefore requiring the notion of equivalent streams to be relaxed to allow for out-of-order events. In fact, lack of order is often desirable, since it enables parallelism. Importantly on the other hand, *not all* output events are unordered, because operators in streaming dataflow graphs (in contrast to in batch processing and MapReduce settings) often require some order to preserved (see Section 2). Because of this, not all operators simply decompose into commutative/associative aggregators, and prior solutions on testing [Chen et al. 2016; Csallner et al. 2011; Marynowski et al. 2012; Xu et al. 2013a] and static verification [Liu et al. 2014; Raychev et al. 2015] for MapReduce-like programs cannot be directly applied.

The second challenge is that stream processing systems, in contrast to batch processing systems, are designed to process input data that would not fit in memory. As best practice, it is recommended that applications written in these systems are tested under heavy load for long periods of time, to match the conditions that are expected after deployment [Vianna et al. 2019]. Achieving this requires that the testing framework itself is an online algorithm, in the sense that it processes output events as they arrive, and that the computational overhead is minimal.

To address these challenges, we propose a matching algorithm that incrementally compares two streams for equivalence. Following the approach of Mamouras et al. [2019], in our solution, ordering requirements between pairs of events are abstracted in a *dependence relation* that indicates when the ordering of two specific events is of significance to the output consumer. Given *any* dependence relation provided by the user, the algorithm determines, in an online fashion, whether the streams are equivalent up to the reorderings allowed by the dependence relation. We show that the algorithm is correct and that it reaches a verdict at the earliest possible time (Theorem 4.8). We also prove that the algorithm is optimal, in the sense that it keeps a minimal amount of space: any correct online algorithm must store at least as much information (Theorem 4.9).

We have implemented DiffStream, a differential testing library for Apache Flink that incorporates our algorithm. DiffStream is implemented in Java and can be used alongside existing testing frameworks such as JUnit [2019] or in stand-alone Flink programs. In order to evaluate the effectiveness and usability of the proposed testing framework, we have conducted a series of case studies.

First, we evaluate the effectiveness of the framework on a set of nondeterministic MapReduce programs from Xiao et al. [2014], adapted to the streaming setting. For some of these programs nondeterminism constitutes a bug, while for others it is acceptable, depending on input assumptions and application requirements. Using our framework, we demonstrate that tests can be written to successfully detect 5 out of 5 bugs (true positives), and to avoid flagging 5 out of 7 bug-free programs (false positives). This improves on previous work [Xu et al. 2013b], which would generally flag all nondeterministic programs as buggy, thus suffering from false positives.

Second, we design two specific use cases to illustrate the benefits of using DiffStream to design and implement parallel Flink applications. We consider a difficult-to-parallelize application which requires event-based windowing: we show that it is significantly more difficult (requiring twice as many lines of code) to effectively parallelize this application using Flink, and we show how our framework can be used to test and correctly implement such an application. We also evaluate the effort needed to write tests for an example computation with a subtle bug. The choice of specific programs we consider are explained in more detail in Section 2.

Finally, we demonstrate that the matching algorithm is efficient in practice, and can be used in an online monitoring setting, by monitoring two implementations of the Yahoo Streaming Benchmark [Chintapalli et al. 2016] over the span of two hours and measuring the impact on performance. The overhead of testing is a modest 5% reduction in maximum possible throughput, and the memory usage remains stable at less than 500 unmatched items out of 30K items per second, reflecting the theoretical optimality of the algorithm for this particular application.

In total, the main contributions of this work are:

- A new testing methodology for specifying ordering requirements in stream processing programs. (Section 3)
- An optimal online matching algorithm for differential testing of stream processing programs which uniformly handles data with differing ordering requirements. (Section 4)
- DiffStream, a differential testing library for testing Apache Flink applications based on the online matching algorithm, together with a series of case studies to evaluate its usability and effectiveness. DiffStream is available as an open source repository on GitHub.[1] (Section 5)

2 OVERVIEW

2.1 Motivating Examples

Programs written in distributed stream processing frameworks exhibit implicit parallelism, which can lead to subtle bugs. Programs in such frameworks are usually written as *dataflow graphs*, where

[1] https://github.com/fniksic/diffstream

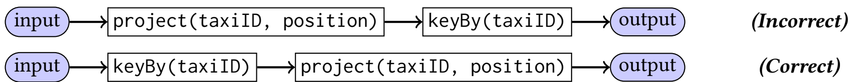


Fig. 1. A subtle consequence of implicit parallelism over an input stream containing taxi location data.

the edges are data streams and the nodes are streaming operators or transformations. Common operators include stateless transformations (*map*), and operations that group events based on the value of some field (*key-by*). For example, suppose that we have a single input stream which contains information about rides of a taxi service: each input event (`id`, `pos`, `meta`) consists of a taxi identifier, the taxi position, and some metadata. In the first stage, we want to discard the metadata component (*map*) and partition the data by taxi ID (*key-by*). In the second stage, we want to aggregate this data to report the total distance traveled by each taxi. Notice that the second stage is order-dependent (events for each taxi need to arrive in order), so it is important that the first stage does not disrupt the ordering of events for a particular taxi ID.

To make a program for the first stage of this computation in a distributed stream processing framework such as Flink or Storm, we need to build a dataflow graph representing a sequence of transformations on data streams. A first (natural) attempt to write the program is given in Fig. 1 (top). Here, the `project` node projects the data to only the fields we are interested in; in this case, `taxiID` and `position`. And `keyBy` (also known as "group by" in SQL-like languages, or the concept of a "stream grouping" in Storm) partitions the data stream into substreams by `taxiID`. Although written as an operator, here `keyBy` can be thought of as modifying the stream to give it a certain property (namely, if it is parallelized, streams should be grouped by the given key).

The first attempt is incorrect, however, because it fails to preserve the order of data for a particular key (taxi ID), which is required for the second stage of the computation. The problem is that dataflow graph operators are implicitly parallelized—here, the stateless map `project` is internally replicated into several copies, and the events of the input stream are divided among the copies. Because input events of the same key may get split across substreams, when the operator `keyBy` reassigns each item to a new partition based on its key, if items of a particular key were previously split up, then they might get reassembled in the wrong order.

This issue can be addressed by ensuring that parallelization is done only on the basis of `taxiID` *from the beginning of the pipeline*. This can typically be accomplished by simply by reversing the `project` and `keyBy` transformations, as in Fig. 1 (bottom). (For example, this is done explicitly in Flink, and the concept is the same in Storm, except that instead of an explicit `keyBy` operator we implicitly construct it by setting the input stream to be grouped by key.) Although the two programs are equivalent when the `project` operation is not parallelized, the second lacks the undesirable behavior in the presence of parallelism: assuming the `project` operation has the same level of parallelism as `keyBy`, most systems will continue to use the same partition of the stream to compute the projection, so data for each key will be kept in-order. In particular, this works in any framework which guarantees that the same key-based partitioning is used between stages.

We have seen that even simple programs can exhibit counterintuitive behavior. In practice, programs written to exploit parallelism are often much more complex. To illustrate this, consider a single input stream consisting of very large documents, where we want to assign a topic to each document. The documents are streamed word by word and delineated by end-of-file markers. The topic of each word is specified in a precomputed database, and the topic of a document is defined to be the most frequent topic among words in that document.

In this second example querying the database is a costly operation, so it is desirable to parallelize by partitioning the words within each document into substreams. However, the challenge is to

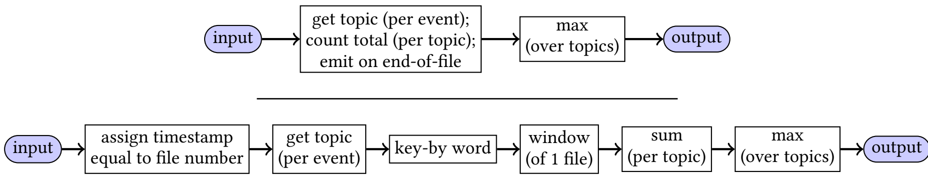


Fig. 2. A difficult-to-parallelize sequential program (top), and the correct parallel version (bottom) over an input set of documents which arrive concatenated in a single stream.

do so in a way that allows for the end-of-file markers to act as *barriers*, so that we re-group and output the summary at the end of each document. Although a sequential solution for this problem is easy, the simplest solution we have found in Flink that exploits parallelism uses about twice as many lines of code (Fig. 2). The source of the complexity is that we must first use the end-of-file events to assign a unique timestamp to each document (ignoring the usual timestamps on events used by Flink). After these timestamps are assigned, only then is it safe to parallelize, because windowing by timestamp later recovers the original file (set of events with a given timestamp). We also consulted with Flink users on the Flink mailing list, and we were not able to come up with a simpler solution. This example is explored in detail in Section 5.2. The additional complexity in developing the parallel solution, which requires changing the dataflow structure and not simply tuning some parameter, further motivates the need for differential testing.

2.2 Solution Architecture

These examples motivate the need for some form of testing to determine the correctness of distributed stream processing applications. We propose *differential testing* of the sequential and parallel versions. As the parallel solution might be much more involved, this helps validate that parallelization was done correctly and did not introduce bugs.

In the example of Fig. 1, the programmer begins with either the correct program P_1 (bottom), or the incorrect program P_1' (top), and wishes to test it for correctness. To do so, they write a correct reference implementation P_2; this can be done by explicitly disallowing parallelism. Most frameworks allow the level of parallelism to be customized; e.g. in Flink, it can be disabled by calling `.setParallelism(1)` on the stream. The program P_1 or P_2 is then viewed as a black-box reactive system: a function from its input streams to a single *output stream* of events that are produced by the program in response to input events.

However, the specification of P_1 and P_2 alone is not enough, because we need to know whether the output data produced by either program should be considered unordered, ordered, or a mixture of both. A naive differential testing algorithm might assume that output streams are out-of-order, checking for multiset equivalence after both programs finish; but in this case, the two possible programs P_1 will both be equivalent to P_2. Alternatively, it might assume that output streams are in-order; but in this case, neither P_1 nor P_1' will be equivalent to P_2, because data for different taxi IDs will be out of order in the parallel solution. To solve this, the programmer additionally specifies a *dependence relation*: given two events of the output stream, it returns *true* if the order between them should be considered significant. For this example, output events are dependent if they have the same taxi ID. In general, the dependence relation can be used to describe a flexible combination of ordered, unordered, or partially ordered data.

The end-to-end testing architecture is shown in Fig. 3. In summary, the programmer provides: (1) a program (i.e., streaming dataflow graph) P_1 which they wish to test for correctness; (2) a correct reference implementation P_2; (3) a *dependence relation* which tells the tester which events

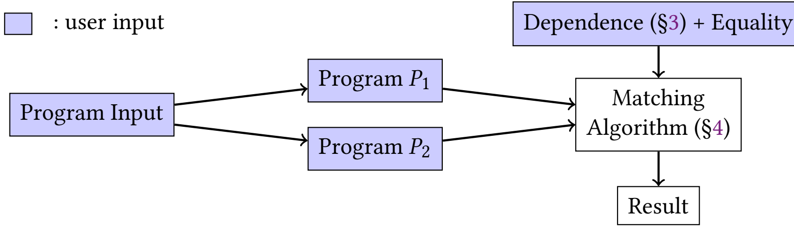


Fig. 3. Architecture for our testing framework.

(a) Specification in DiffStream

(b) Dependence visualized as a graph

Fig. 4. Example specification in DiffStream for the taxi example. Taxi events with the same `taxiID` are dependent.

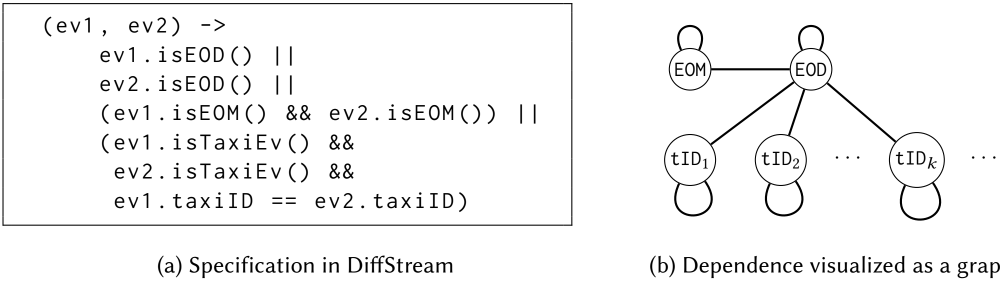
in the output stream may be out-of-order; (4) if needed, overriding the definition of equality for output stream events (for example, this can be useful if the output items may contain timestamps or metadata that is not relevant for the correctness of the computation); and (5) optionally, a custom generator of input data streams, or a custom input stream—otherwise, the default generator is used to generate random input streams. The two programs are then connected to our differential testing algorithm, which consumes the output data, monitors whether the output streams so far are equivalent, and reports a mismatch in the outputs as soon as possible.

3 SPECIFYING ORDERING REQUIREMENTS

In this section we describe how the programmer writes specifications in DiffStream. Let's look back at the taxi example from Section 2.1. The second stage of the program computes the total distance traveled by each taxi by computing the distance between the current and the previous location, and adding that to a sum. For this computation to return correct results, location events for each taxi should arrive in order in its input—a requirement that must be checked if we want to test the first stage of the program. We propose expressing this ordering requirement using a *dependence relation D*. The concept of dependence relations was first introduced in research on concurrency theory, where it was used to define Mazurkiewicz traces, i.e. partially ordered sequences of events in distributed systems [Mazurkiewicz 1986], and has previously been used to give semantics for stream processing programs [Mamouras et al. 2019].

A dependence relation is a symmetric binary relation on events of a stream with the following semantics. If $x \mathrel{D} y$, then the order of x and y in a stream is significant and reordering them gives us two streams that are not equivalent. This could be the case if the consumer of an output stream produces different results depending on the order of x and y. Thus, the dependence relation can be thought of as encoding the pairwise ordering requirements of the downstream consumer.

It is often helpful to visualize dependence relations as unordered graphs, where nodes are equivalence classes of the dependence relation. For the taxi example, the dependence relation is visualized in Fig. 4b, and it indicates that events with the same taxi identifier are dependent. In DiffStream, dependence relations can be specified using a Boolean function on a pair of events.



(a) Specification in DiffStream

(b) Dependence visualized as a graph

Fig. 5. Example specification in DiffStream for the extended taxi example. Taxi events with the same `taxiID` are dependent and all events are dependent with end-of-day (EOD) events.

```
(ev1, ev2) -> distance(ev1.loc, ev2.loc) < 1
```

Fig. 6. Example specification in DiffStream where events are dependent if their locations are close.

```
(ev1, ev2) -> (ev1.isPunctuation() &&
                ev2.timestamp < ev1.timestamp) ||
               (ev2.isPunctuation() &&
                ev1.timestamp < ev2.timestamp)
```

Fig. 7. Example specification in DiffStream where punctuation events, used to enforce progress, depend on other events only if the punctuation timestamp is larger.

These functions should be pure and should only depend on the fields of the two events. The DiffStream specification of the dependence relation from Fig. 4b is shown in Fig. 4a.

Now let's consider an extension of the above example where the downstream consumer computes the total distance traveled by each taxi *per day*, and also computes the average daily distance by each taxi every month. To make this possible, the output of the program under test is now extended with special EOD (*end-of-day*) and EOM (*end-of-month*) events. The ordering requirements on this output, while more subtle, can still be precisely specified using a dependence relation. For example, EOD events are dependent with taxi events since all events of a specific day have to occur before the EOD event of that day for the total daily distance to be correctly computed. On the other hand, EOM events do not have to be dependent with taxi events since daily distances are computed on EOD events. Therefore, an EOM event can occur anywhere between the last EOD event of the month and the first EOD event of the next month. The DiffStream specification of the dependence relation and its visualization are both shown in Fig. 5.

Several frequently occurring dependence relations can be specified using a combination of the predicates seen in the above examples. This includes predicates that check if an event is of a specific type (e.g. `isEOD()`, `isTaxiEv()`), and predicates that check a field (possibly denoting a key or identifier) of the two events for equality (e.g. `ev1.taxiID == ev2.taxiID`). However, it is conceivable that the dependence of two events is determined based on a complex predicate on their fields. Figure 6 shows an example where the proximity of two taxi location events determines if they are dependent.

Another interesting dependence relation occurs in cases where output streams contain punctuation events. Punctuations are periodic events that contain a timestamp and indicate that all events

Algorithm DiffStream Checking equivalence of two streams

Input: Equality relation $\equiv$, dependence relation D
Input: Connected stream s with $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$
Require: Relations $\equiv$ and D are compatible

1: **function** StreamsEquivalent(s)
2: $u_1, u_2 \leftarrow$ empty logically ordered sets
3: Ghost state: $p_1, p_2 \leftarrow$ empty logically ordered sets
4: Ghost state: $f \leftarrow$ empty function $p_1 \to p_2$
5: **for** (x, i) in s **do**
6: $j \leftarrow 3 - i$
7: **if** x is minimal in u_i and $\exists y \in \min u_j : x \equiv y$ **then**
8: $u_j \leftarrow u_j \setminus \{y\}$
9: $p_i \leftarrow p_i \cup \{x\}; p_j \leftarrow p_j \cup \{y\}$
10: $f \leftarrow f[x \mapsto y]$ **if** $i = 1$ **else** $f[y \mapsto x]$
11: **else if** $\exists y \in u_j : x \text{ D } y$ **then**
12: **return false**
13: **else**
14: $u_i \leftarrow u_i \cup \{x\}$
15: **return** $(u_1 = \emptyset$ and $u_2 = \emptyset)$

up to that timestamp, i.e. all events `ev` such that `ev.timestamp < punc.timestamp`, have *most likely* already occurred. Punctuation events allow programs to make progress, completing any computation that was waiting for events with earlier timestamps. However, since events could be arbitrarily delayed, some of them could arrive after the punctuation. Consider as an example a taxi that briefly disconnects from the network and sends the events produced while disconnected after it reconnects with the network. These events are usually processed with a custom out-of-order handler, or are completely dropped. Therefore, punctuation events are dependent with events that have an earlier timestamp, since reordering them alters the result of the computation, while they are independent of events with later timestamps. This can be specified in DiffStream as shown in Fig. 7.

4 ALGORITHM

In this section we present Algorithm DiffStream, our algorithm for checking equivalence of two streams. As described in Section 1, the algorithm has two main features: (i) it can check for equivalence up to any reordering dictated by a given dependence relation, and (ii) it is online—it processes elements of the stream one at a time. We prove that our algorithm is correct, and we show that it is optimal in the amount of state it stores during execution.

4.1 Background

Before getting to the algorithm itself, we need to introduce some terminology. A *stream* s is a bounded or unbounded sequence of elements: $s = \langle x_1, x_2, \ldots \rangle$. We write $x \in s$ to denote that x is an element of s, we write $s[n]$ for the nth element of s, and we write $s[{:}n]$ for the bounded substream of elements up to and including the nth element.

We follow the convention that all elements of a stream (denoted with x, y, etc.) are distinct. This is so that we can unambiguously refer to the location of x in the stream s, and for example, say

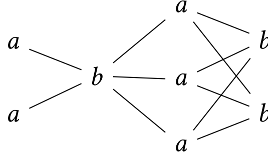


Fig. 8. The logical order of the stream from Example 4.1. Vertically aligned elements are logically unordered, and for two elements that are not aligned, the left one logically precedes the right one. The two leftmost elements are minimal.

which of x and y occurs earlier. We use $x \equiv y$ to refer to equality of the *underlying values*, rather than the elements as positioned in the stream.

Two streams s_1, s_2 are given as input to the algorithm as a *connected stream* s, which is a stream obtained by arbitrarily interleaving the elements of s_1 and s_2. More precisely, the elements of the connected stream s are of the form (x, i) such that $i \in \{1, 2\}$ and $x \in s_i$. We can recover the original streams by using *projections* π_1 and π_2: $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$. Conversely, given a stream s, we can form connected streams using *injections* ι_1 and ι_2: $\iota_1(s)$ is obtained by mapping each $x \in s$ to $(x, 1)$, and analogously, $\iota_2(s)$ is obtained by mapping each $x \in s$ to $(x, 2)$. Thus, $\iota_1(s)$ and $\iota_2(s)$ are characterized by $\pi_1(\iota_1(s)) = \pi_2(\iota_2(s)) = s$ and $\pi_1(\iota_2(s)) = \pi_2(\iota_1(s)) = \emptyset$. The motivation for connected streams comes from the fact that the streams s_1 and s_2 are produced by the stream processing system asynchronously.

Next, we need to describe what it means for two streams to be equivalent. Our notion of equivalence relies on two user-specified relations on the elements of the streams: an *equality relation*, denoted by $\equiv$, and a *dependence relation*, denoted by D. The equality relation is provided by the user (e.g., in Java by overriding the method equals()) and is required to be an equivalence relation, that is, it should be reflexive, transitive, and symmetric. For elements x and y, we write $x \equiv y$ instead of $x = y$ for the equality relation to emphasize that it refers to equality on the underlying values, rather than equality of stream elements (as we assume stream elements at different positions are distinct). The dependence relation is required to be symmetric, that is, for elements x and y, $x \mathrel{\mathrm{D}} y$ implies $y \mathrel{\mathrm{D}} x$. Finally, the equality and the dependence are required to be *compatible*: if $x \mathrel{\mathrm{D}} y$ and $x \equiv x'$, then $x' \mathrel{\mathrm{D}} y$. The three requirements—the equality being an equivalence relation, the dependence being a symmetric relation, and the equality and dependence being compatible—need to be ensured by the user.

Given a stream s, a dependence relation D gives rise to a *logical order* on the elements in s: for elements $x, y \in s$, x logically precedes y, denoted by $x < y$, if x precedes y in the stream and either x and y are dependent or they are transitively dependent—there are intermediate elements $x_1, \ldots, x_n \in s$ given in their order of occurrence in s such that $x \mathrel{\mathrm{D}} x_1 \mathrel{\mathrm{D}} \ldots \mathrel{\mathrm{D}} x_n \mathrel{\mathrm{D}} y$. It can be shown that the logical order is irreflexive and transitive, that is, it is a strict (irreflexive) partial order on the elements of the stream s. Recall that this makes sense because by convention all the elements are distinct, even though the underlying values may be equivalent according to the equality relation $\equiv$.

Example 4.1. Consider a stream $s = \langle a, a, b, a, a, a, b, b \rangle$. The equality relation $\equiv$ is given by $a \equiv a$ and $b \equiv b$, and the dependence relation D is given by $a \mathrel{\mathrm{D}} b$ (and $b \mathrel{\mathrm{D}} a$). The logical order arising from D is shown in Fig. 8. The logical orderings between elements include $s[1] < s[3]$, $s[3] < s[4]$, and $s[5] < s[7]$. Also $s[4] \parallel s[5]$, $s[4] \parallel s[6]$, and $s[5] \parallel s[6]$. Note that $s[1] < s[4]$ even though $s[1] \not\mathrel{\mathrm{D}} s[4]$ (both elements are a). This is because they both depend on $s[3] = b$, which is in between.

Given two streams s and s', an equality relation $\equiv$, and a dependence relation D, we say that s and s' are *equivalent* if they give rise to the same logical order. More precisely, we say they are equivalent if there exists a bijective mapping $f\colon s \to s'$, called a *matching*, that matches equal elements and preserves the logical order, that is, for every $x, y \in s$, $f(x) \equiv x$ and $f(x) < f(y)$ if and only if $x < y$. In case the streams are equivalent, we write $s \equiv_{\mathrm{D}} s'$, or simply $s \equiv s'$ if the dependence relation is clear from the context. We call two streams that are not equivalent *distinguishable*.

If the two streams s and s' are bounded, one way to think about them being equivalent is as follows: we can get from s to s' in finitely many steps by either swapping two adjacent logically unordered elements or by replacing an element with another equal element. In particular, bounded equivalent streams have the same length.

Example 4.2. Streams $s_1 = \langle a, c, b\rangle$ and $s_2 = \langle c, a, b\rangle$ are equivalent with respect to a dependence relation given by $a \mathrel{\mathrm{D}} b$ and $c \mathrel{\mathrm{D}} b$. A (unique) matching is given by $f\colon s_1[1] \mapsto s_2[2]$, $s_1[2] \mapsto s_2[1]$, and $s_1[3] \mapsto s_2[3]$. Note that the same streams are not equivalent with respect to a dependence relation where additionally $a \mathrel{\mathrm{D}} c$.

When it comes to unbounded streams, it may be impossible to algorithmically decide whether they are equivalent or not. For example, consider $s_1 = \langle a, a, a, \ldots\rangle$ and $s_2 = \langle b, b, b, \ldots\rangle$ with $a \not\mathrel{\mathrm{D}} b$. Clearly, $s_1 \not\equiv s_2$, but an algorithm processing a connected stream s with $\pi_1(s) = s_1$ and $\pi_2(s) = s_2$ one element at a time can never reach a conclusion: perhaps eventually b's will start arriving on the first stream, and a's will start arriving on the second stream. However, there are situations when an algorithm can reach a definite decision even if the streams are unbounded. We say that a connected stream s is *finitely distinguishable* if there is a position n such that for every continuation s' of $s[:n]$, the projected streams $\pi_1(s[:n] \cdot s')$ and $\pi_2(s[:n] \cdot s')$ are distinguishable.

Example 4.3. If s is a connected stream such that $\pi_1(s) = \langle a, a, b\rangle$ and $\pi_2(s) = \langle a, b\rangle$, and $a \mathrel{\mathrm{D}} b$, then for no continuation of s will the two projections ever be equivalent. Thus, s is finitely distinguishable.

Given a partial order p, we say that an element $x \in p$ is *minimal* if no other element is less than x. There can be multiple minimal elements in p; we denote the set of minimal elements in p by $\min p$. Given two partial orders p and q such that $p \subseteq q$, we say that p is a *prefix* of q if for every element $x \in p$, p also contains all the elements that are less than x in q.

4.2 Algorithm Description

We now give a general specification of an online equivalence-checking algorithm. The algorithm's inputs are an equality relation $\equiv$, a dependence relation D, and a connected stream s with the projections $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$. We require the equality and the dependence relations to be compatible. The algorithm provides a function StreamsEquivalent that returns return **true** or **false** to report whether or not $s_1 \equiv s_2$. The function is allowed to iterate over s exactly once.

The algorithm is correct if it has the following behavior:

(I) StreamsEquivalent returns **true** if and only if s is bounded and s_1, s_2 are equivalent.

(II) StreamsEquivalent returns **false** if and only if either s is finitely distinguishable or it is bounded and the streams s_1, s_2 are distinguishable. Additionally, if s is finitely distinguishable, it returns **false** after processing $s[n]$ for the first position n such that $s[:n]$ is finitely distinguishable.

Algorithm DiffStream achieves the required behavior in the following way. Intuitively, it tries to construct a matching to demonstrate equivalence of s_1 and s_2. In doing so, as part of its state it maintains two logically ordered sets u_1 and u_2, both initially empty. Their role is to keep track of the unmatched elements from s_1 and s_2, respectively. In addition to u_1 and u_2, which constitute the

physical state, the algorithm maintains the so-called "ghost state," written in gray in Algorithm DiffStream. The ghost state need not exist in any real implementation of the algorithm; its sole purpose is to aid in proving correctness. As part of the ghost state, the algorithm maintains two additional logically ordered sets p_1 and p_2, whose role is to keep track of the successfully matched prefixes of s_1 and s_2. In the ghost state, the algorithm also explicitly keeps track of the matching $f : p_1 \to p_2$.

When processing a new element x from s_i (lines 5–14 in Algorithm DiffStream), there are three distinguished cases:

(1) The element x is minimal in u_i and there is a corresponding unmatched minimal element $y \in u_j$ such that $x \equiv y$ (line 7). In this case we remove y from u_j. In the ghost state, we add x to p_i and y to p_j, and we extend the matching f to map x to y or y to x, depending on whether $x \in s_1$ or $y \in s_1$ (lines 9–10).
(2) The element x depends on some unmatched element $y \in u_j$ (line 11). If this is the case, then we have detected finite distinguishability and the function returns **false** (line 12).
(3) If neither of the previous cases holds (line 13), then for every $y \in u_j$, x and y are unequal and independent. We add x to u_i as an unmatched element (line 14).

If the whole connected stream has been processed and the function StreamsEquivalent did not return **false** in line 12, it returns **true** in line 15 if and only if both sets of unmatched elements are empty.

Example 4.4. Let us demonstrate the execution of Algorithm DiffStream on streams s_1 and s_2 from Example 4.2. Suppose the connected stream given as input is

$$s = \langle (a, 1), (c, 2), (c, 1), (b, 1), (a, 2), (b, 2) \rangle .$$

At the time of processing the element $s[3] = (c, 1)$, the first two elements have already been processed, and both of them are unmatched: $u_1 = \{a\}$ and $u_2 = \{c\}$. The algorithm detects that the new element c is minimal in u_1 and it can be matched with the element $c \in u_2$, so it updates the matching f with $f(s_1[2]) = s_2[1]$ and removes c from u_2. Next, it processes $s[4] = (b, 1)$: the element b is not minimal in u_1 as it depends on $a \in u_1$. It is also not dependent on any element in u_2, as u_2 is empty. Therefore, it is added to u_1, which now contains the ordering $a < b$. Finally, the two elements $s[5] = (a, 2)$ and $s[6] = (b, 2)$ arrive precisely in the right order to be matched with the elements in u_1, and the algorithm concludes that the streams are equivalent.

If the dependence relation contained the additional dependence $a \mathrel{D} c$, the processing would have stopped at the element $s[2] = (c, 2)$, since the element c from s_2 would have been dependent on an unmatched element $a \in u_1$. And indeed, the connected stream $s[:2] = \langle (a, 1), (c, 2) \rangle$ would have been finitely distinguishable.

4.3 Correctness

In order to show that the algorithm is correct, we will show that the loop in StreamsEquivalent (lines 5–14) maintains the following invariants.

(I0) For every $x \in u_1$ and $y \in u_2$, x and y are unequal and independent: $\forall x \in u_1, \forall y \in u_2 : x \not\equiv y \wedge x \not\mathrel{D} y$.
(I1) p_1 is a prefix of s_1: $\forall x \in p_1, \forall y \in s_1 : y < x \Rightarrow y \in p_1$.
(I2) p_2 is a prefix of s_2: $\forall x \in p_2, \forall y \in s_2 : y < x \Rightarrow y \in p_2$.
(I3) $f : p_1 \to p_2$ is a maximal matching, that is, it is a matching and no extension $f' : p_1' \to p_2'$ to proper supersets $p_1' \supset p_1$ and $p_2' \supset p_2$ is a matching. We also say that p_1 and p_2 are maximally matched prefixes.

Lemma 4.5. *The loop in StreamsEquivalent (lines 5–14) maintains invariants (I0)–(I3).*

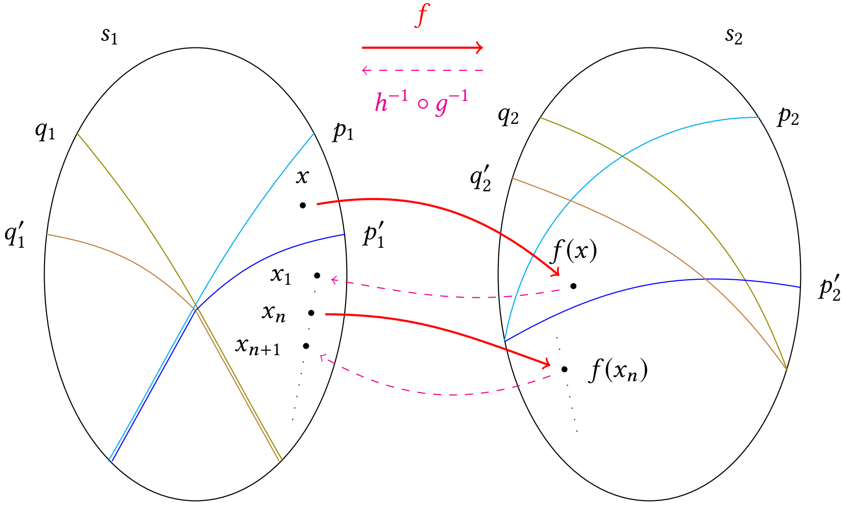


Fig. 9. Illustration of the argument showing that $p'_1 \neq p_1$ is impossible in the proof of Lemma 4.6.

Proof. Of the four invariants, the one that is least straightforward is (I3), so let us show that it holds. In particular, let us show that after the update in lines 9–10 of Algorithm DiffStream, the function f remains a matching between p_1 and p_2. Clearly it is still a bijection and it maps elements in p_1 to equal elements in p_2. In order to show that it still preserves the logical order, we only need to show that for every $x' \in p_1$, $x' < x$ if and only if $f(x') < f(x) = y$. Let us show this claim in one direction (the other one is analogous). Assume $x' < x$. By definition, there exists $n \geq 1$ and elements $x_0, \ldots, x_n \in p_1$ such that $x' = x_0 \mathrel{D} x_1 \mathrel{D} \ldots \mathrel{D} x_n = x$. By the compatibility of $\equiv$ and D, we also have $f(x') = f(x_0) \mathrel{D} f(x_1) \mathrel{D} \ldots \mathrel{D} f(x_n) = y$. By the invariant (I2), y does not logically precede $f(x_{n-1})$, so it must be $f(x_{n-1}) < y$, and finally by transitivity $f(x') < y$. As for the maximality of f, since (I3) holds at the start of the loop, any extension to f must involve the element x that is being processed. Thus, if f can be extended, it is extended by the ghost statements in lines 9–10. □

Lemma 4.6. *In a bounded connected stream s, all maximally matched prefixes of $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$ are equivalent.*

Proof. Let $f : p_1 \longrightarrow p_2$ and $g : q_1 \longrightarrow q_2$ be two maximal matchings, where p_1 and q_1 are prefixes of s_1, and p_2 and q_2 are prefixes of s_2. Let $u_1 = s_1 \setminus p_1, u_2 = s_2 \setminus p_2$, and $v_1 = s_1 \setminus q_1, v_2 = s_2 \setminus q_2$ be the corresponding unmatched elements. To show the claim, it suffices to show that $p_1 \equiv q_1$.

Clearly the identity function $\mathrm{id} : p_1 \cap q_1 \longrightarrow p_1 \cap q_1$ is a matching. Let $h : p'_1 \longrightarrow q'_1$ be a maximal matching between p_1 and q_1 that extends id; thus, p'_1 and q'_1 are prefixes such that $p_1 \cap q_1 \subseteq p'_1 \subseteq p_1$ and $p_1 \cap q_1 \subseteq q'_1 \subseteq q_1$. To show $p_1 \equiv q_1$, it suffices to show that $p'_1 = p_1$ and $q'_1 = q_1$.

First we note that for the matchings f, g, h, an analog of the invariant (I0) holds. In particular, the sets $p_1 \setminus p'_1$ and $q_1 \setminus q'_1$, as well as v_1 and v_2 are pairwise independent and unequal. Moreover, let us set $p'_2 = f(p'_1)$ and $q'_2 = g(q'_1)$. The sets $p_2 \setminus p'_2$ and $q_2 \setminus q'_2$ are also pairwise independent and unequal. Let us show the claim for $p_1 \setminus p'_1$ and $q_1 \setminus q'_1$. Suppose first that $x \in p_1 \setminus p'_1$ and $y \in q_1 \setminus q'_1$ are elements such that $x \mathrel{D} y$. Since x and y are both in s_1, we either have $x < y$ or $y < x$. If $x < y$, then we have $x \in q_1$ since q_1 is a prefix, and consequently $x \in p_1 \cap q_1 \subseteq p'_1$, which is a contradiction. Likewise, $y < x$ leads to $y \in q'_1$, which is also a contradiction. Suppose now that $x \in p_1 \setminus p'_1$ and

$y \in q_1 \setminus q'_1$ are elements such that $x \equiv y$. They cannot both be minimal, for we would be able to extend h with $h(x) = y$. Thus, one of x and y has a logical predecessor; say $x' \in p_1 \setminus p'_1$ is such that $x' < x$. Without loss of generality, $x' \mathrel{\mathrm{D}} x$. Since $\equiv$ and D are compatible, this leads to $x' \mathrel{\mathrm{D}} y$, which we have just shown to be impossible.

We are now ready to prove $p'_1 = p_1$ ($q'_1 = q_1$ is analogous). An illustration of this proof is shown in Fig. 9. For the sake of contradiction, suppose that $p'_1 \neq p_1$, that is, there exists an element $x \in p_1 \setminus p'_1$. Note that $x \notin q_1$, that is, $x \in v_1$. We send x to s_2 via f; we have $f(x) \in p_2 \setminus p'_2$. The element $f(x)$ cannot be in v_2 since $x \in v_1$ and $x \equiv f(x)$. The element $f(x)$ also cannot be in $q_2 \setminus q'_2$, since it is in $p_2 \setminus p'_2$ and $f(x) \equiv f(x)$. Hence, $f(x) \in q'_2$. Now we pull $f(x)$ back to s_1 via g and h. Set $x_1 = h^{-1}(g^{-1}(f(x)))$; we have $x_1 \in p'_1$. Since $x \notin p'_1$, x_1 and x are distinct elements such that $x_1 \equiv x$. We can continue iterating the described process. Suppose we have defined distinct elements $x_1, \ldots, x_n \in p'_1$ for some $n \geq 1$ such that $x \equiv x_1 \equiv \ldots \equiv x_n$. We send x_n to s_2 via f and note that neither $f(x_n) \notin v_2$ (otherwise we would have $x \in v_1$, $f(x_n) \in v_2$, and $x \equiv f(x_n)$) nor $f(x_n) \in q_2 \setminus q'_2$ (otherwise we would have $f(x) \in p_2 \setminus p'_2$, $f(x_n) \in q_2 \setminus q'_2$, and $f(x) \equiv f(x_n)$). Therefore, $f(x_n) \in q'_2$ and we can bring it back to s_1 via g and h to get a well-defined element $x_{n+1} = h^{-1}(g^{-1}(f(x_n))) \in p'_1$. Suppose $x_{n+1} = x_k$ for some k with $1 \leq k \leq n$. By removing k layers of application of $h^{-1} \circ g^{-1} \circ f$, we conclude that $x_{n+1-k} = x$, which cannot be since $x_{n+1-k} \in p'_1$ and $x \notin p'_1$. Therefore, x_{n+1} is distinct from all previously defined elements.

By defining the described process, we have shown that there are infinitely many distinct elements in p'_1, which cannot be since p'_1 is a finite set. Hence, $x \in p_1 \setminus p'_1$ cannot exist in the first place, and we have established that $p_1 = p'_1$. Analogously, we have $q_1 = q'_1$, and since $p'_1 \equiv q'_1$, we also have $p_1 \equiv q_1$. Finally, using $p_1 \equiv q_1$ as a link, we establish that $p_2 \equiv p_1 \equiv q_1 \equiv q_2$, that is, all the maximally matched prefixes in s are equivalent. ◻

Lemma 4.7. *If* StreamsEquivalent *returns* ***false*** *in line 12 while processing* $s[n]$ *for some* $n \geq 1$*, then the connected stream* $s[:n]$ *is finitely distinguishable.*

Proof. Let $s[n] = (x, 1)$ and assume that ProcessElement returns **false** in line 12 when processing $s[n]$. Since the condition in line 11 was satisfied, there exists $y \in u_2$ such that $x \mathrel{\mathrm{D}} y$. Without loss of generality, let y be a minimal such element in u_2.

The challenge here is that even though f can never be extended to match either x or y, it is plausible that $s[:n]$ can nevertheless be extended to s' with a completely different matching that somehow accommodates both elements. To show that such a scenario is impossible, suppose s' is an extension of $s[:n]$ such that $\pi_1(s') = s'_1 \supseteq s_1$, $\pi_2(s') = s'_2 \supseteq s_2$, and suppose $g\colon s'_2 \to s'_1$ is a matching (it helps to view g in the direction opposite of f). Since $x \mathrel{\mathrm{D}} y$ and $g(y) \equiv y$, we have $x \mathrel{\mathrm{D}} g(y)$, so x and $g(y)$ are logically ordered in s'_1. There are three possibilities: $g(y) = x$ (the elements are identical), $g(y) < x$, or $x < g(y)$.

The first possibility can easily be discarded. From $g(y) = x$ it follows that $x \equiv y$. The elements x and y cannot both be minimal elements unmatched by f, otherwise x would have been matched in lines 7–10. Therefore, either there is $x' \in u_1$ such that $x' < x$ and $x' \mathrel{\mathrm{D}} x$, or there is $y' \in u_2$ such that $y' < y$ and $y' \mathrel{\mathrm{D}} y$. In the former case we have $x' \mathrel{\mathrm{D}} y$, contradicting the invariant (I0), and in the latter case we have $x \mathrel{\mathrm{D}} y'$ and $y' < y$, contradicting the choice of y as a minimal unmatched element such that $x \mathrel{\mathrm{D}} y$.

The second possibility, that $g(y) < x$, can be discarded as follows. Set $y_0 = y$. The element $g(y_0)$ cannot be in u_1 as that would break the invariant (I0); therefore it has to be in p_1. We can send it to p_2 via f; let $y_1 = f(g(y_0))$. Since $y_0 \notin p_2$, y_0 and y_1 are distinct elements such that $y_0 \equiv y_1$. We iterate the process: suppose we have defined the elements $y_1, \ldots, y_n \in p_2$ for some $n \geq 1$ such that all of them are equal to y_0, and all of them together with y_0 are distinct. Since $g(y_n) \mathrel{\mathrm{D}} x$, $g(y_n)$ and x are logically ordered. It cannot be $g(y_n) \geq x$, as that would imply $g(y_n) > g(y_0)$ and consequently

$y_n > y_0$. But then, since $y_n \in p_2$ and p_2 is a prefix, we would have $y_0 \in p_2$, which is a contradiction. Thus, $g(y_n) < x$ and consequently $g(y_n) \in p_1$ due to the invariant (I0). Hence, $y_{n+1} = f(g(y_n))$ is a well-defined element such that $y_{n+1} \in p_2$. Clearly y_{n+1} is distinct from y_0, and $y_{n+1} \equiv y_0$. Suppose $y_{n+1} = y_k$ for some k with $1 \leq k \leq n$. By "pealing off" k layers of applications of f and g, we would get $y_{n+1-k} = y_0$, which is a contradiction. Therefore, the new element is distinct from every previously defined element. We have thus defined infinitely many distinct elements in the finite set p_2, which is a contradiction. Hence, $g(y) \nless x$.

The third possibility, that $x < g(y)$, is discarded by a similar argument. The idea is again to start from $x_0 = x$ and define an infinite sequence of distinct elements $x_1, x_2, \ldots$ in p_1, all of which are equal to and distinct from x_0. However, the argument that shows the sequence is well-defined is slightly different. Similarly as before, given x_n for $n \geq 1$ we start by arguing that $g^{-1}(x_n) \in p_2$. We first establish that $g^{-1}(x_n) < y$, as otherwise $g^{-1}(x_n) \geq y > g^{-1}(x_0)$ would imply $x_n > x_0$ and consequently $x_0 \in p_1$. Next, if $g^{-1}(x_n) \in u_2$, then we argue as in the first possibility: either x and $g^{-1}(x_n)$ can be matched, or there is $x' < x$ in u_1 such that $x' \mathrel{D} g^{-1}(x_n)$, contradicting the invariant (I0), or there is $y' < g^{-1}(x_n) < y$ in u_2 such that $x \mathrel{D} y'$, contradicting the minimality of y. Hence, $g^{-1}(x_n) \in p_2$ and $x_{n+1} = f^{-1}(g^{-1}(x_n))$ is well-defined. Showing that it is distinct from previously defined elements is done using the same argument as before. Thus, we again reach a contradiction, and $x \nless g(y)$.

By discarding all three possibilities, we conclude that the extension of $s[:n]$ to a connected stream s' such that $s'_1 \equiv s'_2$ does not exist. Hence, $s[:n]$ is finitely distinguishable. □

Theorem 4.8. *Algorithm DiffStream is correct.*

Proof. We first show that Algorithm DiffStream satisfies the correctness condition (I). If StreamsEquivalent returns **true** in line 15, then the whole connected stream s has been processed. Hence, s is bounded. Moreover, in this case both u_1 and u_2 are empty, implying $p_1 = s_1$ and $p_2 = s_2$. Therefore, $s_1 \equiv s_2$ follows from the invariant (I3). Conversely, if s is bounded and s_1, s_2 are equivalent, by Lemma 4.7, StreamsEquivalent does not return **false** on line 12, and the loop in lines 5–14 finishes. By the invariant (I3) and Lemma 4.6, f must fully match s_1 and s_2. Hence, u_1 and u_2 are empty and StreamsEquivalent returns **true**.

Next, we show the correctness condition (II). If StreamsEquivalent returns **false**, it either returns **false** in line 12 and s is finitely distinguishable by Lemma 4.7, or it returns **false** in line 15. In the latter case, s is bounded and one of u_1 and u_2 is not empty. Hence, s_1 and s_2 are distinguishable by the invariant (I3) and Lemma 4.6. Conversely, if s is bounded and s_1, s_2 are distinguishable, by (I) the function StreamsEquivalent does not return **true**, so it returns **false**.

It remains to show that if s is finitely distinguishable, then StreamsEquivalent returns **false** in line 12 for the first position n such that $s[:n]$ is finitely distinguishable. Clearly StreamsEquivalent does not return **false** on line 12 when processing $s[k]$ for $k < n$, since in that case by Lemma 4.7 already $s[:k]$ would be finitely distinguishable. Let $s[n] = (x, 1)$ and let u_1 and u_2 be the logically ordered sets of unmatched elements at the start of the loop when processing $s[n]$. If x can be matched with an element $y \in \min u_2$, then we extend $s[:n]$ with $\iota_1(u_2 \setminus \{y\})$ followed by $\iota_2(u_1)$, with the elements of $u_i, i \in \{1, 2\}$ given in the order in which they appear in $s[:n-1]$. Likewise, if $x \not\mathrel{D} y$ for every $y \in u_2$, then we extend $s[:n]$ with $\iota_1(u_2)$ followed by $\iota_2(u_1 \cup \{x\})$. In either case, from the invariants (I0)–(I3) it follows that Algorithm DiffStream would decide that the two streams in the extension are equivalent, and since the algorithm is correct for bounded equivalent streams, the streams would indeed be equivalent. Therefore, in both cases the connected stream $s[:n]$ would not be finitely distinguishable. Since $s[:n]$ is finitely distinguishable, the only remaining option is that there exists $y \in u_2$ such that $x \mathrel{D} y$, and hence StreamsEquivalent returns **false** in line 12 when processing $s[n]$. □

4.4 Optimality

When it comes to stateful stream processing programs, space usage is an important topic. If a stateful stream processing program inadvertently stores too much of the stream's history, since the stream is potentially unbounded, the program's space usage may grow unboundedly as well. In case of Algorithm DiffStream, its space usage may indeed grow unboundedly. Since Algorithm DiffStream stores sets of unmatched elements, it can be forced to keep the complete history of the connected stream it takes as input. For example, this would happen on a connected stream s such that $\pi_1(s) = \langle a, a, \ldots \rangle$, $\pi_2(s) = \langle b, b, \ldots \rangle$, with $a \not\equiv b$ and $a \not{D}\, b$. However, it turns out that for a correct equivalence-matching algorithm there is no way around this: a correct equivalence-checking algorithm must store a certain amount of unmatched elements in one way or another. In each step, Algorithm DiffStream stores minimal sets of unmatched elements (the complements of maximally matched prefixes), and as we show in this subsection, in this sense it is optimal.

Recall that by Lemma 4.6, in a bounded connected stream s with $s_1 = \pi_1(s)$ and $s_2 = \pi_2(s)$, all maximally matched prefixes of s_1 and s_2 are equivalent. It follows that their complements—minimal sets of unmatched elements—are equivalent as well. More precisely, let (p_1, p_2) and (q_1, q_2) be two pairs of maximally matched prefixes such that $p_i, q_i \subseteq s_i$ for $i \in \{1, 2\}$, and let $u_i = s_i \setminus p_i$ and $v_i = s_i \setminus q_i$ for $i \in \{1, 2\}$. Then $u_1 \equiv v_1$ and $u_2 \equiv v_2$. This allows us to define up to equivalence a function $u(s) = (u_1, u_2)$, where u_1 and u_2 are any minimal sets of unmatched elements in s_1 and s_2. We write $(u_1, u_2) \equiv (v_1, v_2)$ to mean $u_1 \equiv v_1$ and $u_2 \equiv v_2$.

If a bounded connected stream s with $u(s) = (u_1, u_2)$ is not finitely distinguishable, then an analog of the invariant (I0) holds for u_1 and u_2: for every $x \in u_1$ and $y \in u_2$, $x \not\equiv y$ and $x \not{D}\, y$.

Theorem 4.9. *Algorithm DiffStream is optimal. More precisely, let A be any other* correct *algorithm for the equivalence-checking problem. If s and s' are two bounded connected streams that are not finitely distinguishable, and if Algorithm DiffStream reaches a different state after processing s and s', then A reaches a different state after processing s and s'.*

Proof. The state stored by Algorithm DiffStream on processing a string s is $u(s)$. Suppose a correct equivalence-checking algorithm reaches the same state after processing s and s'. Let $u(s) = (u_1, u_2)$ and $u(s') = (u_1', u_2')$. We extend both s and s' with $\iota_1(u_2)$ followed by $\iota_2(u_1)$. It is not difficult to see that the two connected streams in the extension of s are equivalent, and the streams in the extension of s' are not equivalent. However, since the algorithm is deterministic, it would come to the same decision for both extensions, which contradicts its correctness. □

4.5 Practical Bounds on Space Usage

While the space used by Algorithm DiffStream can grow with the input stream in the worst case, Theorem 4.9 shows that it is impossible to write an algorithm which uses a smaller amount of space. In addition to this result, it is possible to give concrete bounds on the space usage in certain cases. Here we discuss some patterns that we have encountered, including the examples we implemented in Section 5, and how bounds on the space usage can be derived for these patterns.

The simplest pattern is differential testing of sequential outputs: both streams are completely ordered, i.e., any two events are dependent. In this case, any differential testing algorithm must at least keep track of the difference of the two streams seen so far (assuming their prefixes are equal). For example, suppose both streams are sequences of integers, and one stream has seen m integers and the other has seen n integers, where $m < n$. Then if the first m integers are equal, any matching algorithm must keep track of the remaining $(n - m)$ integers. Thus, the space usage of the algorithm is bounded by the maximum *drift* between the two streams, defined as the difference in the number of events produced. In practice, such drift is typically bounded since inputs arrive at the same rate for both the implementations, and most systems try to ensure that no operator

in the stream processing dataflow graph lags behind the others by accumulating a large queue of unprocessed inputs. This dependence relation pattern (and the resulting bound) occurs in the case studies of Sections 5.2 and 5.3.

A second common pattern is *key-based parallelism*, where events with the same key are dependent, but events with different keys are independent. In such a case, considering the drift between the two streams is not enough. For example, suppose there are only two keys, a and b, and one stream produces n as, but the other stream produces n bs. Then although the two streams are producing the same number of items, because stream 1 never produces an a and stream 2 never produces a b, any algorithm for correct matching must keep at least the as and the bs so far until they are matched on the other stream. To address this, we can obtain a bound on the space by considering the drift *per key*. In general, "keys" can be generalized as dependent subsets of events, and a bound can be obtained by taking the maximum drift on any dependent subset, together with the number of independent keys in the input. This dependence relation pattern (and the resulting bound) occur in the case study of Section 5.1. Related to key-based parallelism, *fully independent parallelism* (where all input events are independent) occurs in the case studies of Sections 5.3 and 5.4.

Finally, we have encountered cases where the input includes special synchronization events, such as punctuation marks and end-of-day markers, as described in Section 3. These events are dependent on all other events. If both input programs to the differential testing algorithm produce these events at regular intervals, then the space usage becomes bounded. In particular, suppose that the *frequency* of such events is at least one in every k events, and the *drift* restricted only to such events is d. Then the space usage of our algorithm is at most $k \times d$.

To ensure that bounded drift holds between the two streams in practice, one approach would be to leverage *back-pressure* of the underlying system [Chen et al. 2017; Collins and Carloni 2009; Kulkarni et al. 2015]. In particular, back-pressure may prevent drift from growing in cases where one implementation is significantly faster than the other, since the system would slow down the fast implementation to prevent unbounded buffering.

5 IMPLEMENTATION AND CASE STUDIES

We implemented the matcher algorithm in DiffStream, a differential testing library written in Java. The matcher can be used to test programs in any distributed stream processing system given an output interface. For our case studies we chose Flink as the target platform because it is one of the most widely used distributed stream processing frameworks [Stack Overflow 2020]. We integrated DiffStream with JUnit-QuickCheck [Holser 2013] to support generation of streams of random input values.

Our first case study (Section 5.1) is used to qualitatively measure the developer effort needed to test an application with non-trivial ordering dependency in its output. In the second case study (Section 5.2) we demonstrate that getting performance benefits from parallelization in Flink might require an elaborate implementation and we illustrate how our tool can be used to streamline that process. In the third case study (Section 5.3), we show that our framework is successful in finding real bugs while largely avoiding false positives by adapting a set of non-deterministic MapReduce programs from the literature [Xiao et al. 2014]. The final case study (Section 5.4) investigates the performance overhead when using DiffStream for online monitoring of long-running applications.

5.1 Taxi Distance

This case study illustrates the process that one has to follow in order to test their implementation using our tool. Recall the taxi distance example in Section 2.1. This example shows two seemingly equivalent implementations of the same query that produce different results in the presence of

```
public void testKeyBy() throws Exception {
    StreamExecutionEnvironment env = ...;

    DataStream input = generateInput(env);

    StreamEquivalenceMatcher matcher =
        StreamEquivalenceMatcher.createMatcher(
            sequentialImpl(input), parallelImpl(input),
            (ev1, ev2) -> ev1.taxiID == ev2.taxiID);

    env.execute();
    matcher.assertStreamsAreEquivalent();
}
```

Fig. 10. An example test in DiffStream.

parallelism. Here is an instantiation of that example in Flink; the first implementation preserves the order of events for each key, while the second one does not:

```
inStream.keyBy("taxiID").project("taxiID", "position");

inStream.project("taxiID", "position").keyBy("taxiID");
```

Note that both implementations preserve the order when executed sequentially. Since such subtle differences are difficult to spot manually, we would like to be able to test a parallel implementation against a sequential one, before deploying it. A slightly simplified example of a test that can exercise this bug using our framework is shown in Fig. 10. First, the Flink execution environment is initialized and the dataflow graph is setup. Then, a random input stream is generated and fed to both implementations, that are finally compared using the matcher for output equivalence.

Notice that the final argument of the matcher is a lambda expression representing the dependence relation that is expected from the consumer of the output. This specific instantiation represents the dependence that was shown in Fig. 4a—i.e., that two items are dependent (and thus must be ordered) if they have the same `taxiID`. If the user did not want to test differences in the ordering of the output, they can use `(ev1, ev2) -> false` as the dependence relation.

In order to compare the effort required to write a test with and without using our framework, we manually implemented a test that exercises this bug. The manually implemented matcher spans two Java classes, totaling around 100 LoC (in contrast to the 13 LoC of the test in our framework shown in Fig. 10). This does not include input generation, for which we used JUnit QuickCheck. The manually implemented matcher keeps two hashmaps—one for each implementation—that map keys to lists, in order to encode the dependence of events of the same key. It appends each output item to the list associated with its key. After the two implementations stop executing, it checks that the two hashmaps represent equivalent output. Note that this manual matcher is not online, in the sense that the two implementations have to stop producing outputs for it to make a decision. We also implemented an online version of it by extending it with 30 more lines of code.

The important point is that the dependence relation abstraction enables the design of a reusable testing framework that can be used for testing applications with different ordering requirements on their outputs. In contrast, the main drawback of the manual matcher is that it is tied to a specific ordering requirement on the output; whenever a user wants to write a test that requires a different output dependence relation, they would have to implement a new matcher that maintains the

output in a data structure suitable for the specific dependence. This can quickly become an overhead if the user wants to write tens or hundreds of tests for different parts of their application.

In summary, we have shown that using our tool to write a test for a stream processing application is significantly easier than writing custom tests (~10 LoC vs. ~100 LoC). In addition, our tool offers additional flexibility, as it can be used to test any two implementations just by changing the dependence relation given to it. This flexibility reduces the effort needed to implement tests for an application. It also exposes ordering requirements, forcing the developer to think about them explicitly.

5.2 Topic Count

The main goal of our second case study is to show that achieving parallelism in distributed stream processing programs can be very difficult and require a drastically more complicated solution than the sequential code. In particular, we consider the example introduced in Section 2.1, that involves counting topics associated with words in a long document and outputting the most frequent topic as the overall topic of the document. The documents are streamed word by word, with end-of-file markers delineating words in different documents. In the sequential solution (Fig. 2, top), we process each word by querying the topic for that word, then updating the total count for that topic; when we get end-of-file, we emit the counts. We feed this output to a second operator *count max*, which finds the maximum over all topics of the count.

At first, one may think that going from a sequential to a parallel program is simply a matter of setting the Flink's parallelism parameter to more than 1. Unfortunately, this is not the case. Consider the first operator in the sequential dataflow shown in Fig. 2 (top). The problem with parallelizing this operator is that an end-of-file marker for a particular document would only be processed by a single sub-operator; thus the other sub-operators would not be able to properly delineate words from this document and the next one. Another way of stating the problem is to say that even though the words themselves are independent, they are dependent on the end-of-file markers, and thus the obvious parallelization is not possible. A differential test that compares the sequential dataflow to the same dataflow with parallelism set to more than 1 quickly discovers that the two versions are indeed not equivalent.

Instead, a correct parallel solution (Fig. 2, bottom) works as follows. We first attach logical timestamps to each word in the input, corresponding to the number of the document that we are currently processing. We then replace end-of-file markers, which act as explicit punctuation in the stream, with punctuated watermarks—a mechanism in Flink that informs the dataflow operators about the passage of logical time. Unlike the explicit end-of-file markers that cannot be shared by multiple sub-operators, the watermarks are seamlessly propagated by the system. In effect, they allow us to break the explicit dependence between words and end-of-file markers in the input stream, allowing the later stages of the dataflow to be parallelized. We parallelize with *key-by* (keys can be assigned arbitrarily to words), and query the database for each word. The next operator is a tumbling window which uses the logical timestamps from earlier to form the window of events *for a single document*, still parallelized. Finally, we sum up the values in each window by topic, and in the last stage *count max* we find the maximum over all topics of the count.

In our search for a correct parallel solution, we consulted with Flink users on the Flink mailing list. Several iterations of feedback were needed to find a correct parallel implementation, and this shows that it is not obvious. Having the differential testing framework helped guide the search by quickly dismissing wrong implementations. The final solution is the dataflow shown in Fig. 2 (bottom), consisting of 6 dataflow operators and twice as many lines of code as the sequential solution.

Solution	Lines of Code	Speedup due to parallelism (par.)			
		par. 2	par. 4	par. 6	par. 8
Sequential	68	–	–	–	–
Correct Parallel	133	2.01	4.33	5.0	4.99

Fig. 11. Results of the second case study on difficulty of writing parallel code.

It is not necessarily true that a solution that seems parallel achieves a speed-up in practice. We therefore finally need to measure the performance of the parallel solution to show that it indeed takes advantage of parallelism and scales performance with the level of parallelism. We evaluated our parallel solution on an input stream of 5 documents, each consisting of 500,000 words randomly selected from a list of 10,000 most common English words. Each word had previously been randomly assigned one of 20 topics, and the association had been stored in a standalone Redis key-value store. The purpose of having the Flink program query the Redis store was to simulate a series of non-trivial operations that would benefit by being parallelized. We executed this experiment on a server with an Intel Xeon Gold 6154 processor and 384 GB of memory. The results of the evaluation are shown in Fig. 11. By increasing parallelism from 1 to 6, the execution time decreases from 80 s to 16 s (on our setup, the benefits taper off after 6).

In summary, although there is a clear performance benefit in having a parallel solution (Fig. 11), the correct solution is difficult to find, as evidenced qualitatively by our search and discussions on the Flink mailing list, and quantitatively because it has about twice as many lines of code as the sequential solution. Applying differential testing to the parallel solution ensures that bugs were not introduced in the process.

5.3 Real-World MapReduce Programs

To determine whether our testing framework can successfully find bugs in real-world programs, we surveyed the literature for empirical studies which have collected and categorized bugs in stream- and batch-processing programs. We excluded works that focus on job failures and performance issues [Kavulya et al. 2010; Li et al. 2013; Schroeder and Gibson 2009; Zhou et al. 2015], as they do not provide examples of semantic bugs where the output might be incorrect. In contrast, Xiao et al. [2014] study nondeterminism due to parallelism in MapReduce jobs in production workflows, identifying both real bugs as well as several nondeterministic code patterns which are bug-free. This empirical study provides a good starting point to evaluate whether our framework can successfully identify the bugs, while not falsely flagging the bug-free examples.

By examining 507 custom (i.e., user-written) reduce functions, the study identifies 5 common reducer patterns (spanning 258 of the custom reducers) which are *non-commutative* on general input data, meaning that there is potential for nondeterminism in the output due to parallelism. An example reducer pattern reported by the study is shown in Fig. 13 (left). While the original custom reducers are not publicly available, the 5 code patterns are nevertheless minimal test cases which exhibit the same behavior, and a large majority of non-commutative reducers (88%) were found to fall into these categories. However, as the study notes, there are two reasons why code written using these patterns might not be erroneous. First, with certain input assumptions, the nondeterminism may disappear (this is possible in 4 out of 5 patterns). Second, nondeterminism may simply be acceptable for the particular application (this is most conceivable in 3 out of 5 patterns). We therefore evaluate our testing framework for each of the five patterns, considering three possibilities for the application-specific requirements: determinism (nondeterminism is not

Code pattern	Determinism	Application-Specific Requirements	
		Determinism under input assumptions	None (nondeterminism acceptable)
`SingleItem`	✓	✓	n/a
`IndexValuePair`	✓	✓	n/a
`MaxRow`	✓	✓	✗
`FirstN`	✓	✓	✗
`StrConcat`	✓	n/a	✓

Fig. 12. Results of the MapReduce case study. A ✓ indicates successfully identifying the bug in the first column, and successfully avoiding a false positive in the second and third columns, for each of the 5 reducers implemented.

acceptable), determinism under certain input assumptions, and no determinism required. We want to answer the following questions, corresponding to these three possibilities:

Q1. If determinism is required on all inputs, can we write a test which successfully detects the nondeterminism and reports a bug?
Q2. If determinism is required but only under certain input assumptions, can we write a test using those input assumptions to avoid a false positive?
Q3. If determinism is not required at all, can we write a test which avoids a false positive?

For each question, we implement the reducer adapted to the streaming setting, and run DiffStream to compare a sequential and parallel version. The results are summarized in Fig. 12, where columns 1, 2, and 3 correspond to the above three questions, respectively.

For example, in the `IndexValuePair` pattern in Fig. 13 (left), the user wrote a reducer to aggregate input items with two input fields, `x` and `y`, by accumulating them in a map where the value of field `x` is set to the value of field `y`. The reducer is nondeterministic in general because there may be multiple updates to the same field: multiple items with the same `x` and a different `y` (which may be out-of-order due to parallelism in the map stage of MapReduce). However, if the input satisfies a *functional dependency* where `y` is a function of `x`, the pattern becomes deterministic. Without knowing the user's intention, it may be that determinism is required, and the functional dependency is not satisfied, in which case this code is a bug (Fig. 12, first column); or it may be that determinism is required, and the functional dependency is satisfied, in which case this code is not a bug (Fig. 12, second column). Though unlikely for this particular pattern, the study authors also noted for some patterns that nondeterminism in the output may be acceptable, despite the functional dependency not being satisfied (Fig. 12, third column).

We implemented each of the 5 reducer patterns in Flink. We start by translating each reducer directly to an aggregator (`AggregateFunction` in Flink); for the `IndexValuePair` reducer, this yields the code in Fig. 13 (right). To adapt the reducer to the streaming setting, a tumbling window is applied to the input stream, and the reducer is applied to get a result for each window. We used the same input stream item type (`Item`) for all examples, which contains fields `x` and `y`. Before constructing the tumbling window, we used an identity map operator to shuffle the data for each key, so that the order is nondeterministic (thus potentially exposing bugs due to parallelism). We then compared the parallel version of this pipeline with the sequential one using our matcher to look for a bug (difference in the outputs).

To evaluate Q1, we generated arbitrary input data and fed it to the sequential and parallel versions. With enough input data (3000 input data items is sufficient), using a small number of keys and possible input data items, our tester consistently detects the incorrectly parallelized program for all 5 patterns. Concretely, of the 5 confirmed bugs found in production code by the previous

```
Dictionary<int, int> dict
    = new ...;
foreach (Row row in input) {
    int x = row["x"].Integer;
    int y = row["y"].Integer;
    dict[x] = y;
    // ...
}
```

```
public class IndexValuePairReducer ... {
    public Map<Integer, Integer>
            createAccumulator() {
        return new HashMap<>();
    }
    ...
    public Map<Integer, Integer> add(
        Item in, Map<Integer, Integer> outmap
    ) {
        outmap.put(in.x, in.y);
        return outmap;
    }
    ...
}
```

Fig. 13. Example MapReduce code: the `IndexValuePair` reducer pattern reported by the study from production MapReduce jobs [Xiao et al. 2014] (left), and our implementation of the pattern in Flink (right).

study (these do not correspond to the 5 patterns), 4 of the 5 are of this nature, so our test cases likely would have identified these 4 bugs.

To evaluate Q2, we use the same setup but with a custom generator for the input data items. For all patterns except one (called `StrConcat`), the output is deterministic if certain assumptions are made on the input; the custom generator enforces these assumptions. For example, for the fields `x` and `y` which are used in the `IndexValuePair` example of Fig. 13, we enforce the requirement that `y` is a function of `x`. We show that in 4 out of 4 patterns, the output successfully passes our tester, i.e. we avoid a false positive in these 4 scenarios.

Finally, to evaluate Q3, we look at three patterns where it is conceivable that nondeterminism in the output is acceptable. For the first two of these, we are unable to write a test which avoids a false positive. The `MaxRow` pattern involves finding the value of one field such that another field is maximized; it is nondeterministic because there may be multiple values which achieve the maximum. In such a case, differential testing results in a false positive because the two programs may return different values, even though they are both correct. Similarly, the `FirstN` pattern is a reducer which discards all but the first N elements that are seen; on inputs with more than N elements, differential testing results in a false positive for a bug. However, we can avoid the false positive in the last `StrConcat` pattern. This reducer consumes a sequence of input items and concatenates them into a single string separated by a special character (say, `@`). It is nondeterministic because the concatenation is non-commutative, but it is likely that the application requirements consider this acceptable. For this pattern, we implement a custom-defined equality on output data items to define when strings separated by `@` are equal; using this, the pattern is able to pass our tester. Additionally, we implement a second version of `StrConcat` which is more suited to the streaming setting: instead of collecting all items in a single `@`-separated string, we output the items as a data stream. In this case, our tester successfully reports a bug when nondeterminism is undesirable; but when the dependence relation is used to indicate that nondeterminism is acceptable, the test passes.

Summary. When nondeterminism is undesirable, we have written tests to successfully identify it (if it exists) for 5 out of 5 reducer patterns. Of the reducer patterns where nondeterminism might not be present due to input data assumptions, we show how an input data generator can be used to cause the programs to pass our tester for 4 out of 4 patterns. Finally, in the reducer patterns where it is conceivable that nondeterminism in the output might be acceptable, we show how using the dependence relation *or* custom equality with our tester can successfully make the test pass for 1 of

the 3 patterns (`StrConcat`). In total, out of the scenarios where the reducer might conceivably not be buggy (the 4 and 3 patterns just mentioned, respectively), we avoid a false positive in 5 out of 7.

5.4 Performance for Online Monitoring

In Sections 4.4 and 4.5 we discussed theoretical lower and upper bounds on DiffStream's space usage. We showed that DiffStream is optimal, but also that depending on the application and dependence relation, space usage in the worst case could grow unboundedly. In this subsection we evaluate the performance of the matcher in practice for monitoring a realistic streaming application. We aim to demonstrate DiffStream's applicability in online monitoring scenarios, e.g. when testing an application under production load for bugs, or for multi-version execution—a method commonly used to safely update production software [Hosek and Cadar 2013; Maurer and Brumley 2012; Tucek et al. 2009].

We broadly aim to answer the question: what is the overhead of the matcher? Relevant metrics are both its memory usage (reflecting unmatched items) and its impact on the application performance (throughput and latency). We evaluate the following three questions:

Q1. What is the effect of the equivalence matcher on the maximal throughput of the application?
Q2. What is the memory footprint of the matcher? How does the memory usage vary over time?
Q3. What is the latency of the matcher, i.e. how much time does it take for the matcher to process a single event?

For the application, we chose the Yahoo Streaming Benchmark [Chintapalli et al. 2016], a standard performance benchmark for stream processing systems. The Yahoo Streaming Benchmark implements a simple advertisement processing application that receives a stream of advertisement events in the JSON format. The events are parsed, filtered for the ad view events, joined with the ad campaign data from an external database, aggregated over time windows, and stored into the database. The application integrates a Kafka queue, a Flink program, and a Redis database, which makes it representative of streaming applications which interact with external services. In our evaluation, we run a sequential and a parallel version of the advertisement program, with the parallelism parameter set to 2. The matcher compares the streams of the two programs after the join with the ad campaign data, but before the aggregation. The aggregation simply counts the number of ad view events per campaign id and per time window, so it does not depend on the order of events. Thus, we use an empty dependence relation (all events are independent).

To answer Q1, we modified the ad event producer to steadily increase the input rate over time. We then executed two experiments: one with the matcher and one without it. (We simulated the absence of the matcher with a dummy matcher that ignores every event.) Both experiments started with an input rate of 40,000 events/s and the acceleration of 10 events/s^2 and ran for 2,500 seconds, allowing the input rate to increase to 65,000 events/s. The expected ideal throughput of the matcher is 2/3 of the input rate due to (i) the duplication for the two versions of the program, and (ii) the filtering step, which filters approximately 2/3 of the events and leaves 1/3 that finally reach the matcher. The measured throughput is shown in Fig. 14a. Initially, in both experiments the measured throughput matches the ideal throughput. After 1,737 seconds, the experiment with the matcher reaches the throughput of 38,072 events/s, after which it rapidly drops and starts fluctuating. In the experiment without the matcher, the throughput continues to rise with the input rate until it reaches 40,082 events/s after 2,023 seconds, after which it also starts fluctuating. Thus, using the matcher results in approximately 5% decrease in the maximal throughput.

To answer Q2 and Q3, we implemented a rudimentary memory profiler that outputs the total heap memory used by the Java Virtual Machine running the Flink program in 1-second intervals. In addition, we measured the latencies of the matcher by measuring the time it takes to process each

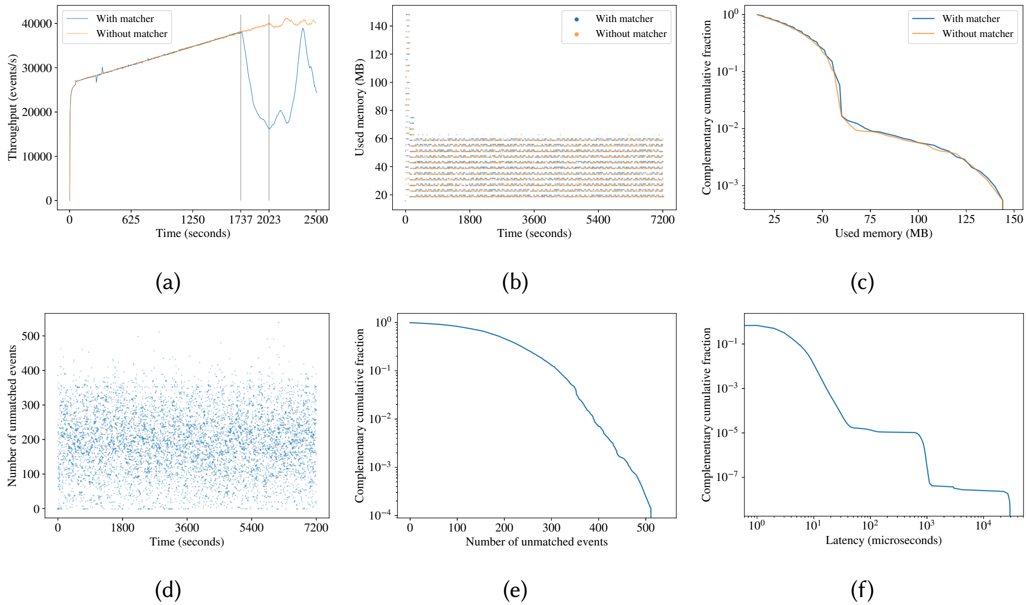


Fig. 14. Results of the fourth case study: performance measurements of monitoring an application with DiffStream on the Yahoo streaming benchmark over a span of 2 hours, compared to the same application without the DiffStream matcher.

event; we stored the latencies in a file during execution. We executed two experiments, with and without the matcher, with a constant input rate of 45,000 events/s and the duration of 7,200 seconds (2 hours). The input rate of 45,000 events/s translated into the throughput of 30,000 events/s, which was stable during the execution.

Figure 14b shows a scatter plot of the memory samples taken during execution. In particular, the total memory used by the program stays bounded throughout the duration of the experiment. Figure 14c visualizes the same information in a different way: for an amount of used memory x on the x-axis, the corresponding value on the y-axis is the fraction of memory samples that exceed x. The two figures together show that there is virtually no difference in the memory usage in the two experiments. We get a more fine-grained view if we look at the number of unmatched events that the matcher stores during execution. Figure 14d shows a scatter plot of samples of unmatched events taken in 1-second intervals during execution. In particular, in almost all samples, the number of unmatched events is below 500. This is even more clearly illustrated in Fig. 14e, which shows the fraction of samples exceeding the given number of unmatched events. Thus, to answer Q2, the memory footprint of the matcher is bounded and negligible relative to the memory used otherwise and the throughput of 30,000 events/s.

Finally, to answer Q3, Fig. 14f shows the fraction of recorded latencies that exceed a given number of microseconds. The mean latency is 2 microseconds. While the maximal latency is 30 milliseconds, this is rare: 98.77% of latencies are at most 10 microseconds, and $>$ 99.99% of latencies are at most 100 microseconds.

Summary. The results show that the overhead of running DiffStream in practice is low. First, the matcher results in a modest 5% reduction in maximal throughput. Second, the memory usage is stable over time and reflects the theoretical optimality for applications where drift between the two

streams is bounded: the number of unmatched elements is $< 1.67\%$ compared to the number events that it processes. Finally, the latency of the matcher is at most 100 microseconds in $> 99.99\%$ of cases, which competitively meets streaming performance standards.

6 RELATED WORK

We survey three broad categories of related work. First, we describe the state of the art on establishing correctness for data-parallel programs: we include both batch processing frameworks like MapReduce, and stream processing frameworks like Flink. Second, we survey the use of partially ordered traces in modeling concurrency, and the associated problems that have been considered such as checking for linearizability, serializability, and data races. Finally, we briefly mention the general testing methodologies our work fits under.

6.1 Correctness for Data-Parallel Programs

Testing. Many previous works focus on batch processing programs written in the MapReduce [Dean and Ghemawat 2008] framework [Chen et al. 2016; Csallner et al. 2011; Marynowski et al. 2012; Xu et al. 2013a] (see also the recent survey by Morán et al. [2019]). Going beyond batch processing, Xu et al. [2013b] study testing semantic properties of operators in general dataflow or stream-processing programs. One problem with many of these works [Chen et al. 2016; Csallner et al. 2011; Xu et al. 2013a,b] is that real-world MapReduce programs (and, by extension, aggregators in stream processing programs) can be non-commutative: the empirical study at Microsoft [Xiao et al. 2014] reports that about 58% of 507 user-written reduce jobs are non-commutative, and that most of these are most likely not buggy. The previous work on testing would erroneously flag such programs as containing bugs due to nondeterminism, which would generate a large number of false positives. We adopt a black-box differential testing approach with the goal of avoiding this problem. Concretely, we have shown in Section 5.3 how to avoid a false positive for most cases where the application requirements imply that the nondeterminism is acceptable.

Static Verification. In addition to testing—a dynamic method of checking correctness—there has also been research on the static verification of data-parallel programs. Recent work focuses on the verification of parallel aggregators that are used in MapReduce programs; either by enabling automated verification and synthesis of *partial aggregators* given an aggregation function [Liu et al. 2014], or by parallelizing user defined aggregators using symbolic execution [Raychev et al. 2015]. Both of these works help developers by statically providing guarantees about the correctness of parallel aggregator functions. DiffStream complements these approaches by checking the correctness of general stateful streaming programs, which are not always decomposable into aggregators, and whose parallel and sequential implementations might have significant structural differences (like the Topic Count case study in Section 5.2), implying that the parallel implementation cannot be simply derived from the sequential implementation. Finally, another difference is that DiffStream (and dynamic approaches in general) can be used on programs that interact with external services (e.g. the Redis database in Section 5.2) without having to model them—as is the case with static approaches.

Language Design. In contrast to the dynamic approach of testing and static verification techniques, there are language-based restrictions to achieve correct (i.e., semantics-preserving) parallelization in stream processing programs. The language StreamIt [Thies et al. 2002] leverages Synchronous Dataflow [Lee and Messerschmitt 1987] to achieve correct parallelization; however, this requires a restriction on dataflow graphs where all operators must have a static *selectivity* (number of output items produced per input item), so it is not appropriate for general stream processing where operators often lack static selectivity. For general stream processing, Schneider et al. [2013] and

Mamouras et al. [2019] have proposed and implemented different approaches to ensure correct parallelization: the first is based on categorizing operators for properties such as statefulness and selectivity, while the second is based on a type discipline where streams are annotated with types. The idea of using dependence relations to specify partially-ordered output streams is originally proposed by Mamouras et al. [2019], and here we apply that idea to testing and online monitoring.

Other Approaches. Complementary to directly establishing the correctness of user-written programs, there is work on indirectly facilitating correctness through visualization, through debugging, and finally by ensuring correctness at the system level. Visualization includes generating example inputs for dataflow programs showcasing typical semantic behavior [Olston et al. 2009]. Debugging includes, e.g., setting up breakpoints, stepping through computations, and determining crash culprits [Gulzar et al. 2016; Olston and Reed 2011]. Towards testing functional correctness of a stream processing system implementation, a framework has been proposed for Microsoft StreamInsight [Raizman et al. 2010].

6.2 Partially Ordered Traces in Concurrency Theory

Mazurkiewicz Traces. We build on foundational work in concurrency theory dating back to Mazurkiewicz [1986], where partially ordered sets of events are called *traces*. Mazurkiewicz traces have been studied from the viewpoint of algebra, combinatorics, formal languages and automata, and logic [Diekert and Rozenberg 1995]. In practical applications to verification and testing of concurrent systems, they appear in relation to *partial order reduction* [Godefroid 1996; Peled 1994], a technique for pruning the search space of possible execution sequences. The idea of a dependence relation to specify output ordering originally comes from Mazurkiewicz traces; however, the core algorithmic problem in our work corresponds to checking *equivalence* of two Mazurkiewicz traces, and to our knowledge this particular testing problem has not been studied in any of the mentioned contexts. An additional difference is that in the theory of Mazurkiewicz traces, one usually assumes a finite, symmetric, and reflexive dependence relation. In contrast, we only require it to be symmetric. This is in order to support user-provided dependence relations over a possibly infinite data domain, which is necessary to model common patterns in the streaming setting: one example is the key-based dependence relation (where the number of keys may be unbounded and different keys are independent). Patterns such as this one cannot be captured by a finite alphabet, and this limits the direct application of classical work on concurrency theory over a finite dependence relation, e.g. Diekert and Rozenberg [1995].

Checking Properties of Traces. Much classical research has focused on deciding properties of traces such as serializability, linearizability, sequential consistency, and data race detection. Broadly speaking, these properties are search problems: the algorithm monitors an execution of events, and it must decide if there exists some possible equivalent execution that witnesses the desired property. For example, race detection involves deciding, given a sequence of events, if there is a valid reordering of the events, subject to the constraints imposed by synchronization events, in which two specific events (representing a potential race condition) get reordered. The search aspect means that race detection is NP-hard [Netzer and Miller 1990, 1992]. Similarly, checking sequential consistency of a given trace is NP-complete [Gibbons and Korach 1992], as is checking linearizability in general [Gibbons and Korach 1997]. As a result, practical tools for testing correctness of traces (e.g., [Burckhardt et al. 2010; Lowe 2017; Park et al. 2011; Savage et al. 1997; Sen 2008; Wing and Gong 1993]) must explore the trade-off between soundness, completeness, and tractability. In contrast, the problem we consider of checking two traces for equivalence up to re-ordering is in PTIME (for the offline variant), and admits an optimal online monitoring algorithm.

6.3 Testing Methodologies

Runtime Verification. Our work contributes to the large body of work on runtime verification [Havelund and Roşu 2004; Leucker and Schallhart 2009], a lightweight verification paradigm which aims to identify bugs in the output of a program as it is executed, using minimal computational resources. Most work in runtime verification focuses on detecting violations of a property written in a logical specification language (e.g., the temporal logic LTL and its extensions), whereas we consider differential testing of a program against a reference implementation, and we model program execution traces as partially rather than totally ordered.

Differential Testing. Differential testing [Groce et al. 2007; McKeeman 1998] is a well-established, lightweight, and scalable way to detect bugs in complex programs (for instance, in C compilers [Yang et al. 2011]), by simply comparing two programs that are supposed to be equivalent. We consider some specific problems that arise in the stream processing domain, specifically, output comparison in the presence of out-of-order data.

7 CONCLUSION

We have presented DiffStream, an algorithm and library for differential testing of distributed stream processing applications. The input to our library, implemented in Flink, consists of the two programs as well as a *dependence relation* which is used to describe which output events must be produced in order, and an optional custom equality which is used to compare output events. Our four case studies demonstrate that (1) our framework can be used to successfully find bugs in MapReduce programs that were previously identified in the literature, while at the same time it avoids false positives in most scenarios; (2) it can be used to ease the development of Flink applications, particularly programs that are difficult to parallelize; and (3) it can run in an online fashion on large input streams with minimal performance overhead.

The core algorithmic problem is testing output streams from two implementations for equivalence, when the output may contain both ordered and unordered events. Formally, we justified that our algorithm is *correct* in an online sense, meaning that it produces a verdict on whether the streams are equivalent as early as possible; and that it is *optimal*, meaning that for any other correct online algorithm, that algorithm must maintain at least as much information as our algorithm does. These theorems are reflected in the overhead in practice (Section 5.4).

The current work focuses on bugs due to parallelism. In future work, we would like to extend our framework to target other classes of bugs, including those due to node or network faults. Additionally, we would like to generalize the definition of correct behavior, which currently assumes that the output should be determined up to allowed reordering and equality of data items. There are cases where this is too strong, for instance when operations are approximate or randomized. Finally, we hope to extend our framework to consider input generation and input equivalence as well as output equivalence, and use this for a better strategy for selecting inputs.

DiffStream is open source and is available on GitHub.

ACKNOWLEDGMENTS

The authors would like to thank all the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by NSF award CCF 1763514.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices* 49, 1 (2014), 373–384.

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 330–340.

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *ACM Sigplan Notices*, Vol. 49. ACM, 271–284.

Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

Saksham Chand, Yanhong A Liu, and Scott D Stoller. 2016. Formal verification of multi-Paxos for distributed consensus. In *International Symposium on Formal Methods*. Springer, 119–136.

Xin Chen, Ymir Vigfusson, Douglas M Blough, Fang Zheng, Kun-Lung Wu, and Liting Hu. 2017. GOVERNOR: Smoother Stream Processing Through Smarter Backpressure. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 145–154.

Yu-Fang Chen, Lei Song, and Zhilin Wu. 2016. The commutativity problem of the MapReduce framework: A transducer-based approach. In *International Conference on Computer Aided Verification*. Springer, 91–111.

Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

Rebecca L Collins and Luca P Carloni. 2009. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*. 205–214.

Christoph Csallner, Leonidas Fegaras, and Chengkai Li. 2011. New ideas track: testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 504–507.

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

Volker Diekert and Grzegorz Rozenberg. 1995. *The Book of Traces*. World Scientific. https://doi.org/10.1142/2563

Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. ACM, 549–552.

Apache Software Foundation. 2019. Apache Storm. http://storm.apache.org/. [Online; accessed March 31, 2019].

Phillip B Gibbons and Ephraim Korach. 1992. The complexity of sequential consistency. In *[1992] Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE, 317–325.

Phillip B Gibbons and Ephraim Korach. 1997. Testing shared memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244.

Patrice Godefroid. 1996. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer-Verlag.

Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 621–631.

Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 784–795.

Klaus Havelund and Grigore Roşu. 2004. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer* 6, 2 (2004).

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 1–17.

Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

Paul Holser. 2013. junit-quickcheck (software). https://github.com/pholser/junit-quickcheck.

Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 612–621.

JUnit. 2019. JUnit testing framework. https://junit.org/junit5/. [Online; accessed October 19, 2019].

Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. 2010. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE

Computer Society, 94–103.

Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303.

Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 963–972.

Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. 2014. Automating distributed partial aggregation. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–12.

Gavin Lowe. 2017. Testing for linearizability. *Concurrency and Computation: Practice and Experience* 29, 4 (2017), e3928.

Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 670–685.

João Eugenio Marynowski, Michel Albonico, Eduardo Cunha de Almeida, and Gerson Sunyé. 2012. Testing MapReduce-based systems. *arXiv preprint arXiv:1209.6580* (2012).

Matthew Maurer and David Brumley. 2012. Tachyon: Tandem Execution for Efficient Live Patch Testing. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 617–630. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/maurer

Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer, 278–324.

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Jesús Morán, Claudio de la Riva, and Javier Tuya. 2019. Testing MapReduce programs: A systematic mapping study. *Journal of Software: Evolution and Process* 31, 3 (2019), e2120.

Robert HB Netzer and Barton P Miller. 1990. *On the complexity of event ordering for shared-memory parallel program executions*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

Robert HB Netzer and Barton P Miller. 1992. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 1 (1992), 74–88.

Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 245–256.

Christopher Olston and Benjamin Reed. 2011. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 1221–1224.

Stack Overflow. 2020. Questions tagged with apache-flink on Stack Overflow. https://stackoverflow.com/questions/tagged/apache-flink. [Online; accessed January 27, 2020].

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 160.

Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. *ACM SIGPLAN Notices* 51, 6 (2016), 614–630.

Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. 2011. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 51.

Doron Peled. 1994. Combining Partial Order Reductions with On-the-fly Model-Checking. In *Computer Aided Verification, Proc. 6th Int. Conference (LNCS 818)*. Springer-Verlag.

Alex Raizman, Asvin Ananthanarayan, Anton Kirilov, Badrish Chandramouli, and Mohamed H Ali. 2010. An extensible test framework for the Microsoft StreamInsight query processor.. In *DBTest*.

Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 153–167.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2013. Safe data parallelism for general streaming. *IEEE transactions on computers* 64, 2 (2013), 504–517.

Bianca Schroeder and Garth Gibson. 2009. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing* 7, 4 (2009), 337–350.

Koushik Sen. 2008. Race directed random testing of concurrent programs. *ACM Sigplan Notices* 43, 6 (2008), 11–21.

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer, 179–196.

Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient Online Validation with Delta Execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/1508244.1508267

Alexandre Vianna, Waldemar Ferreira, and Kiev Gama. 2019. An Exploratory Study of How Specialists Deal with Testing in Data Stream Processing Applications. *arXiv preprint arXiv:1909.11069* (2019).

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN Notices* 50, 6 (2015), 357–368.

Jeannette M. Wing and Chun Gong. 1993. Testing and verifying concurrent objects. *J. Parallel and Distrib. Comput.* 17, 1-2 (1993), 164–182.

Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. 2014. Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 44–53.

Zhihong Xu, Martin Hirzel, and Gregg Rothermel. 2013a. Semantic characterization of MapReduce workloads. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–97.

Zhihong Xu, Martin Hirzel, Gregg Rothermel, and Kun-Lung Wu. 2013b. Testing properties of dataflow program operators. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 103–113.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 423–438.

Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 17–26.