Stream Processing with Dependency-guided Synchronization

Konstantinos Kallas*
University of Pennsylvania
Philadelphia, PA, USA
kallas@seas.upenn.edu

Filip Niksic*
University of Pennsylvania
Philadelphia, PA, USA
fniksic@seas.upenn.edu

Caleb Stanford*
University of Pennsylvania
Philadelphia, PA, USA
castan@cis.upenn.edu

Rajeev Alur
University of Pennsylvania
Philadelphia, PA, USA
alur@cis.upenn.edu

ABSTRACT

Real-time data processing applications with low latency requirements have led to the increasing popularity of data stream processing systems. While such systems offer convenient APIs that can be used to achieve data parallelism automatically, they offer limited support for computations which require synchronization between parallel nodes. In this paper we propose *dependency-guided synchronization (DGS)*, an alternative programming model and stream processing API for stateful streaming computations with complex synchronization requirements. In a nutshell, using our API the input is viewed as partially ordered, and the program consists of a set of parallelization constructs which are applied to decompose the partial order and process events independently. Our API maps to an execution model called *synchronization plans* which supports synchronization between parallel nodes. Our evaluation shows that for a range of applications requiring synchronization, parallelism cannot be achieved automatically in existing systems. In contrast, DGS enables implementations which scale automatically, the resulting synchronization plans offer throughput improvements over what can be achieved in existing systems, and the programming overhead is small compared to writing sequential code.

PVLDB Reference Format:
Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. Stream Processing with Dependency-guided Synchronization. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

A wide range of applications in domains such as healthcare, transportation, and smart homes are increasingly relying on real-time data analytics with low latency and high throughput requirements. This has led to a significant amount of research on *stream processing*, spanning different layers of abstraction in the software stack. At the lowest level, ***stream processing systems*** (e.g. Flink [9], Samza [41], Storm [18], Spark Streaming [49], Trill [11], and Heron [31]) handle scheduling, optimizations and operational concerns; at the intermediate level, ***stream processing APIs*** (e.g. MapReduce online [13], SPADE [20], SP Calculus [46], Timely Dataflow [38, 39], StreamIt [47], Flink's Datastream API [9]), usually based on a form of dataflow [21, 33], abstract the computation in a way that hides implementation details, while exposing parallelization information to the underlying system; and at the top level, ***high-level query languages*** (e.g. Streaming SQL [8, 27], SamzaSQL [43], Structured Streaming [6], StreamQRE [36], CQL [5], AFAs [12]) provide convenient abstractions that are built on top of the core streaming APIs. Of these layers, the stream processing API plays a central role in the successful scaling of applications since its expressiveness restricts the form of available parallelism. In this paper we focus on rethinking the dataflow model at this intermediate layer to enable parallel implementations for a broader range of programs.

The success of stream processing APIs based on the dataflow model can be attributed to their ability to simplify the task of parallel programming. They typically do this by exposing a simple but effective mechanism for data-parallelism called *sharding*: nodes in the dataflow graph are automatically replicated into many parallel instances, each of which processes a different partition of the input events. However, while it is intuitive for programmers, sharding also implicitly limits the scope of parallel patterns that can be expressed. Specifically, it prevents arbitrary *synchronization of state across parallel instances* since it disallows communication between them. This is a severe restriction in a range of modern applications (from video processing [26] to distributed machine learning [42]), which require some synchronization between nodes, but also require high throughput and could therefore benefit from parallelization. This limitation of sharding has been recognized by developers; evidence for the need to support synchronization requirements can be found in numerous feature requests [1–3] in state-of-the-art stream processing systems. Some of these requests have been addressed by developing custom extensions for the APIs (e.g. broadcasting events), enabling parallelization for specific use cases. Unfortunately these extensions do not generalize, as we show in Section 2 using a machine learning fraud detection application. In the general case, users are left with two unsatisfying solutions: either ignore any parallelization potential, implementing their application sequentially; or circumvent the stream processing APIs

*The three marked authors contributed equally to the paper.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

using low level external mechanisms to achieve synchronization between parallel instances.

To address the need for parallelism in the context of synchronization we make two contributions. First, we propose *synchronization plans*, a tree-based execution model which is a restricted form of communicating sequential processes [25]. Synchronization plans are hierarchical structures that represent parallel implementations of streaming computations and capture two important aspects of synchronization: which events need to be processed in order and therefore require synchronization—*when* to synchronize; and how to combine and disaggregate states of independent parallel instances—*what* to communicate. Second, on top of that execution model we propose DGS, a stream processing API that allows the user to provide a sequential specification of the computation and abstractly describe *when* sychronization is required and *what* to communicate if so.

When synchronization is required: To describe when synchronization should happen, our stream processing API allows the user to specify a *dependence relation*[37] on input events. The dependence relation describes which events of the input data stream can be processed independently and out-of-order, allowing the input stream to be viewed as a *partially ordered set* of events.

What to communicate: In order to process an input event that requires synchronization, independent workers have to communicate and aggregate their local states. Our stream processing API allows the user to specify *fork-join* pairs that abstractly describe how to fork the state of a worker into two new workers and how to join back two states into one when synchronizing.

Given this specification, the main technical challenge is to generate a synchronization plan, which corresponds to a concrete implementation, that is both *correct* and *efficient*. More precisely, the challenge lies in ensuring that a derived implementation is correct in the presence of complex synchronization requirements. To address this, we (i) formalize a set of conditions that ensure that a specification is consistent, and (ii) we define a notion of *S-valid* synchronization plans, i.e. plans that are correct with respect to a given specification S. We then provide an end-to-end proof that any concrete implementation that corresponds to an S-valid synchronization plan is *correct* with respect to a consistent specification S. A pluggable optimizing component can then be used to search for an efficient synchronization plan out of the set of S-valid synchronization plans without jeopardizing correctness.

In order to evaluate DGS, we perform a set of experiments to investigate the data parallelism limitations of Flink [9], a representative high-performance stream processing system. We show that these limits can be surpassed by manually implementing synchronization plans in these systems. This however comes at a cost, since the user has to sacrifice the high-level principle of *parallelism-independence* which makes streaming APIs desirable, in particular, the code is not written in a way that is oblivious to the number of parallel nodes. We then develop a prototype that implements DGS in Erlang [7] and show that it can automatically produce scalable implementations (through generating synchronization plans from the specification) independent of parallelism. Finally, we show that our prototype can be used to program realistic applications in two case studies: it compares similarly to handcrafted reference implementations, and the effort required (measured by lines of code) to achieve parallelism is minimal compared to a sequential implementation.

Contributions:

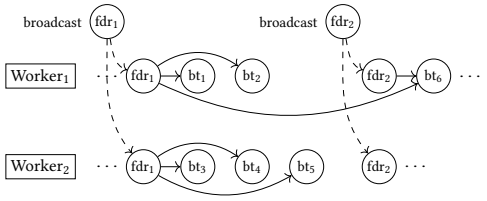
- *DGS*, a novel stream processing API that is designed for specifying streaming computations that require synchronization, and allows viewing input streams as partially ordered sets of events. (Section 3)
- *Synchronization plans*, a tree-based execution model that is designed for parallel streaming computations that require synchronization, and a framework for generating a correct synchronization plan given a specification. We provide an end-to-end proof that given any consistent specification, the framework produces a correct plan. (Section 4)
- A prototype implementation in Erlang, and an evaluation that demonstrates the throughput and scalability benefits achieved by synchronization plans and DGS for synchronization-centered applications compared to Flink, as well as its programming feasibility in real-world case studies. (Sections 5 and 6).

2 OVERVIEW

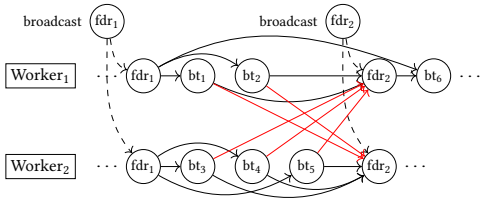
In this section we argue that computations requiring synchronization both arise in practice and cannot be captured by existing stream processing APIs. We focus on a fraud detection streaming application with high throughput requirements that is inspired by a state-of-the-art statistical outlier detection algorithm [42]. We show why the standard way of achieving data parallelism in the dataflow model, sharding, cannot achieve a parallel implementation for the fraud detection application without the use of low level external mechanisms. Indeed, this limitation has been recognized, yet we show that existing techniques for extending or replacing the dataflow model do not solve the problem in general. Finally, we describe the architecture of our solution and how it can be used to achieve a parallel implementation for such an application.

2.1 Background

Data parallelism in systems that are based on the dataflow model is usually achieved using sharding, i.e. replicating parts of the dataflow graph in different parallel instances and then partitioning the inputs and sending each partition to a different shard. For sharding to be applicable, it is necessary that the input events can be partitioned in such a way that there are no dependencies across different partitions. Sharding enables data parallelism in many computations: e.g. stateless ones where all events are independent (e.g. a map that multiplies each input by a constant number), and key-based computations where dependencies only occur between events with the same key (e.g. a streaming join where events are joined when they have the same value in one of their fields). These computations are very common in applications in practice, and therefore sharding is a convenient way for developers to achieve data parallelism without having to think about how computations are parallelized.



(a) Fraud detection.



(b) ML extension of fraud detection.

Figure 1: Illustration of an execution of two fraud detection examples and the event dependencies. Different rows represent different parallel workers, circles represent processing of events, dashed edges represent broadcasting an event, black edges represent dependencies, and red edges represent dependencies across parallel instances.

2.2 Motivating Application

Despite the success and simplicity of sharding, it is not sufficient to achieve data parallelism in general. In particular, we now consider some computations where the input events cannot be partitioned in a way that avoids cross-partition dependencies: i.e. the processing of at least some events in one partition depends on the processing of some of the events in a different partition.

Fraud Detection Application Suppose that we have two types of input events: bank transactions and fraud detection rules that are used to flag some transactions as fraudulent. Bank transactions are continuously input in the system with high rate while fraud detection rules are infrequent. An example of a new rule would be the addition of a specific account to a database of suspicious accounts. In this computation there is no way to partition inputs while avoiding cross-partition dependencies since all events (both transactions and rules) depend on previous rule events. Still, if the rate of bank transaction events becomes a bottleneck (and fraud detection rule events happen rarely) it would be beneficial to parallelize the processing of bank transaction events.

One of the solutions that has been used to achieve data parallelism in such computations involves extending the dataflow model with a broadcast pattern that allows sending some input events to *all* parallel shards (and partitioning the rest of the events as usual). An example execution of the above computation that utilized the broadcast pattern is shown in Figure 1a. By broadcasting the fraud detection rule events a parallel implementation is achieved without requiring cross-instance communication between the parallel instances since dependencies are contained in a single shard. However, as we see below, this solution does not generalize to more complex dependencies.

Fraud Detection with ML extension Consider a simple extension of the above application that is inspired by today's ML workflows. In addition to user-input fraud detection rules, the extended application also trains a model in an unsupervised way using previously seen transactions. Therefore the application now also keeps a sketch of previously seen bank transaction events and when a new fraud detection rule arrives it aggregates the sketches and uses them to retrain the fraud detection model. The dependencies of input events are now more complex, since the processing of fraud detection rule events now depends on all the previous bank transactions. As shown in the example execution in Figure 1b, there are now dependencies across shards (even after broadcasting the rules), so there is need for communication between them.

Unfortunately, standard ways of achieving data parallelism do not support computations with such dependencies, leaving the user with two unsatisfying solutions. They can either accept this shortcoming and execute their computations sequentially—introducing a throughput bottleneck in their application, or they can manually implement the required synchronization using low level external synchronization mechanisms (e.g. a key-value store with strong consistency guarantees). This is error prone and, more importantly, violates the assumptions that systems make on the usage of their stream processing APIs possibly leading to bugs and erroneous behaviors.

2.3 Solution Architecture

Our solution can achieve data parallelism for the extended fraud detection application by utilizing synchronization plans, hierarchical structures of communicating sequential processes that can express parallel implementations with complex dependency patterns. We propose DGS, a stream processing API that can be used on top of synchronization plans. The specification for a program in DGS is split in three parts. First, the user needs to provide a sequential implementation of the program, where the input is assumed to arrive in order and one event at a time. The sequential implementation consists of a stateful update function that can output events and update its state every time an input event is processed. For the fraud detection example, the update function would process bank transactions by checking if they are fraudulent and by constructing a sketch of the previously seen transactions, and fraud detection rules by using the sketch of previously seen transactions and the new rule to update the statistical fraud model. Second, the user provides a dependence relation that indicates the input events for which the processing order must be preserved, inducing a partial order on input events. For the example above, the user would simply indicate that fraud detection rule events depend on all other events. The final part of a specification consists of primitives that describe how to *fork* the state to two independent copies to allow for parallel processing and how to *join* two states when synchronization is required. These primitives abstractly represent splitting the computation into independent computations and merging the results, and are not tied to a specific implementation.

The specifications in DGS are portable, in the sense that they could be used to generate several different concrete implementations. A specification in DGS induces a set of synchronization plans, and then a pluggable optimizing component picks one of them

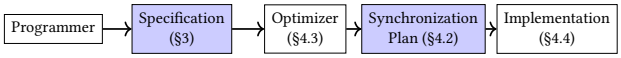


Figure 2: Framework overview. Filled boxes denote the key internal representations, which highlight the separation between specification and implementation.

based on information about the target execution environment, e.g. the number of processing workers and the location of the input streams. As a starting point, we have developed a simple optimizer that tries to minimize the number of messages exchanged between different workers using information about the execution environment and the input streams. We provide an end-to-end proof that all synchronization plans that are generated by our framework are correct with respect to the sequential specification, so the optimizer is free to pick any of them without endangering correctness. An overview of the framework is shown in Figure 2.

3 DEPENDENCY-GUIDED SYNCHRONIZATION

Specifications in our stream processing API contain three components: a *sequential specification*, a *dependence relation* on input events to enforce synchronization, and *fork* and *join* parallelization primitives. Users can provide such a specification by directly writing Erlang code for each of the components. For the additional guarantee that the parallelized implementation will be equivalent to the sequential one (see section 4.5), the user-written components should also be *consistent*, a notion defined in section 3.2.

3.1 Specifications

For a simple but illustrative example, assume that we want to implement a stream processing application that simulates a map from keys to counters with two types of input events: *increment* events, denoted $\mathtt{i}(k)$, and *read-reset* events, denoted $\mathtt{r}(k)$, where k is the associated key with each event. On each increment event, the counter associated with that key should be increased by one, and on each read-reset event, the current value of the counter should be produced as output, and then should be reset to zero. Note that in our stream processing API events have two components: a tag indicating their type and a payload containing their body. In this example both types of events have an empty payload.

Sequential specification. In our stream processing API a user first provides a sequential specification of the computation by describing how to update the state on each input event. A pseudocode version of the specification for the example above is shown in Figure 3. It consists of (i) the state type `State`, i.e. the map from keys to counters, (ii) the initial value of the state `init`, i.e. 0 for all keys, and (iii) a function `update`, which contains the logic for processing input events. Conceptually, the sequential specification describes how to process the data assuming it was all combined into a single stream (e.g., sorted by system timestamp). For example, if the input stream consists of the events $\mathtt{i}(1)$, $\mathtt{i}(2)$, $\mathtt{r}(1)$, $\mathtt{i}(2)$, $\mathtt{r}(1)$, then the output would be 1 followed by 0, produced by the two $\mathtt{r}(1)$ (read-reset) events.

```
// Types
Key = Integer
Tag = i(Key) | r(Key)
Payload = ()
Event = (Tag, Payload)
State = Map(Key, Integer)

// Sequential Implementation
keys = Set(1, ..., n)
init: State
init = Map((k, 0) for k in keys)
update: (State, Event) -> State
update(s, (i(k), ())) =
    s[k] = s[k] + 1; return s
update(s, (r(k), ())) =
    output(s[k]); s[k] = 0; return s

// Dependence Relation
dependencies: Map(Tag, Set(Tag))
dependencies = List2Map(
    [(r(k), Set(r(k), i(k))) for k in keys] +
    [(i(k), Set(r(k))) for k in keys]
)

// Fork and Join
fork: (State, Set(Tag), Set(Tag)) -> (State, State)
fork(s, tags1, tags2) =
    s1 = init(); s2 = init() // two forked states
    for k in keys:
        if r(k) in tags1:
            s1[k] = s[k]
        else: // r(k) in tags2 OR r(k) in neither
            s2[k] = s[k]
    return (s1, s2)
join: (State, State) -> State
join(s1, s2) =
    Map((k, s1[k] + s2[k]) for k in keys)
```

Figure 3: Specification of a map from keys to counters, which can process increment operations $\mathtt{i}(k)$ and read-reset operations $\mathtt{r}(k)$ for each key k. Erlang syntax has been edited for readability.

Dependence relation. To parallelize a sequential computation, the user needs to provide a dependence relation which encodes which events are independent, and thus can be processed in parallel, and which events are dependent, and therefore require synchronization. The dependence relation abstractly captures all the dependency patterns that appear in an application, inducing a partial order on input events. In this example, there are two forms of independence we want to expose. To begin with, *parallelization by key* is possible: the counter map could be partitioned so that different sets of keys are processed independently. Moreover, *parallelizing increments* on the counter of the same key is also possible. In particular, different sets of increments for the same key can be processed independently; we only need to aggregate the independent counts when a read-reset operation arrives. On the other hand, read-reset events depend on all other events: their output is affected by the processing of increments as well as other read-reset events.

We capture this combination of parallelization and synchronization requirements by defining the dependence relation `dependencies` in Figure 3, which is also visualized in Figure 4. For simplicity, here

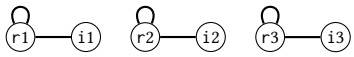


Figure 4: The `dependencies` relation from Figure 3 with three keys, visualized as an undirected graph. Edges enforce synchronization requirements, while non-edges expose parallelism.

the set of keys `keys` is known up front, and used in the dependence relation. For a larger set of keys, we would project the keys down to a fixed number of hash values. The dependence relation then relates hash values instead of keys. The dependence relation should also be *symmetric*, i.e. `e1` is in `dependencies(e2)` if and only if `e2` is in `dependencies(e1)`.

Parallelization primitives: *fork* and *join*. While the dependence relation indicates the possibility of parallelization, it does not provide a mechanism for parallelizing state. Our basic abstraction here is a pair of functions for `fork`ing one state into two, and for `join`ing two states into one, where the fork function additionally takes as input two (possibly non-disjoint) sets of tags where every tag from one set is independent of all tags in the other set. The contract is that after the state is forked into two independent states, each state will then *only be updated using events from the given set of tags*. To see why this is necessary, consider a state which has accumulated a counter v associated with key k, and suppose we are asked to fork it. Then we don't know whether to propogate the pair (k, v) to the first forked state, or the second. This depends on which of the two processes, if any, is responsible for producing output related to key k, i.e. the one that processes events $\mathtt{r}(k)$. A fork-join specification along these lines is shown in fig. 3.

For our specific example, the guarantee is that only one of two forked states will be assigned to process read-reset events for a key. If the input $\mathtt{r}(k)$ is in `tags1`, then we know that updates of key k will all be processed by the first state, and so we copy the value for this key to the first state. Conversely, if the input $\mathtt{r}(k)$ is in `tags2`, we copy the value for this key to the second state. The third case is that the input $\mathtt{r}(k)$ is in neither `tags1` nor `tags1`; then we can copy the value to *either* the first or the second state. In this case, both states may receive increment events. This is acceptable because before any read-reset operation can be processed, the `join` function will be called, which adds the two totals back together again.

Multiple state types. A specification can be more general than we have discussed so far, because we allow for multiple *state types*, instead of just one. The initial state must be of a certain type, but forks and joins can convert from one state type to another: for example, forking a pair into its two elements. The complete stream processing API is summarized in the following definition.

Definition 3.1. Formally, a specification consists of:

(1) A type of input events `Event` and a finite set of tags `Tag`, where the first component of each input event is the associated tag.
(2) A type of output events `Out`.
(3) Finitely many state types `State_0`, `State_1`, etc.
(4) For each state type `State_i`, a set of allowed tags `Tag_i` — this specifies which input values this type of state can process.

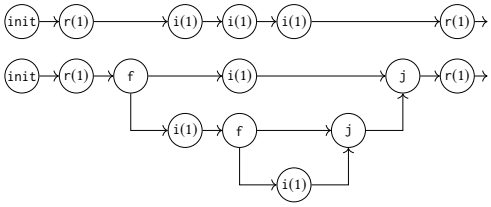


Figure 5: Example of a sequential and parallel execution of the program in fig. 3 on the input stream $\mathtt{r}(1), \mathtt{i}(1), \mathtt{i}(1), \mathtt{i}(1), \mathtt{r}(1)$. There is only one key $k = 1$; `f` and `j` denote fork and join functions. *Top: Sequential execution: each item of the input stream is processed as an update to a global state. Bottom: One possible parallel execution: in between some updates, the state is forked or joined, and* `i` *events are processed as independent updates to one of the forked states.*

(5) A sequential implementation, consisting of a single initial state `init: State_0` and a separate function `update_i: (State_i, Event) -> (State_i, Out)` for each state type `State_i`.
(6) The dependence relation `dependencies: Tag -> Set(Tag)`.
(7) Any number of parallelization primitives, where each is a *fork* or a *join*. A fork function has type `(State_i, Set(Tag_i), Set(Tag_i)) -> (State_j, State_k)`, and a join function has type `(State_j, State_k) -> State_i`, for some i, j, k.

3.2 Consistency of Specifications

The semantics of our stream processing API can be visualized using wire diagrams, as in Figure 5. Computation proceeds from left to right and each wire is associated with a state (of type `State`) and a set of tags corresponding to the events that it can process. Input events are processed as updates to the state, which means they take one input wire and produce one output wire, while forks take one input wire and produce two, and joins take two input wires and produce one. Notice that the same updates are present in both sequential and parallel executions. It is guaranteed in the parallel execution that *fork and join come in pairs*, like matched parentheses. Each set of tags that is given as input to the `fork` function indicates the set of input events that can be processed along the corresponding wire. An execution invariant is that updates, i.e. the sets of tags, on parallel wires must be independent. In the example, the wire is forked into two parts and then forked again, and all three resulting wires process $\mathtt{i}(1)$ events. Note that $\mathtt{r}(1)$ events cannot be processed at that time because they are dependent on $\mathtt{i}(1)$ events.

Any parallel execution is guaranteed to preserve the sequential semantics, i.e. processing all input events in order using the `update` function, as long as the following *consistency conditions* are satisfied. More precisely, these conditions ensure that all possible logical parallel executions of the program are equivalent.

```
join(update(s1, e), s2) = update(join(s1, s2), e)          (1)
join(fork(s, tags1, tags2)) = s                            (2)
update(update(s, e1), e2)) = update(update(s, e2), e1))
                                  for independent e1, e2   (3)
```

These conditions should hold for *all* of the potentially many occurences in a specification of `update`, `join`, `fork`, events `e1` and `e2`, and states `s1` and `s2`, such that the expressions are well-typed. For example, `update(s, e1)` requires that the tag of `e1` is in the set of allowed tags `Tags_i` for the state type `State_i` of `s`. Additionally, we require that the set of outputs produced by the left expression and the right expression is the same (though not necessarily produced in the same order). Consistency can be thought of as analogous to the commutativity and associativity requirements for a MapReduce program to have deterministic output [16]: just as with MapReduce programs, the implementation does not assume the conditions are satisfied, but if not the semantics will be dependent on how the computation is parallelized.

4 SYNCHRONIZATION PLANS

In this section we describe *synchronization plans*, i.e. representations of streaming program implementations, and our framework for generating them given a specification written in our API (see section 3). Generation of an implementation can be conceptually split in two parts, the first affecting correctness and the second performance. First a specification S induces a set of *S-valid*, i.e. correct with respect to it, synchronization plans. Choosing one of those plans is then an independent optimization problem that does not affect correctness and can be delegated to a separate optimization component (section 4.3). Finally, the workers in synchronization plans need to process some incoming events in order while some can be processed out of order (depending on the dependence relation). We propose an efficient selective reordering technique (section 4.4) that can be used in tandem with heartbeats to address this ordering issue. We tie everything together by providing an end-to-end proof that the implementation is correct with respect to a consistent specification S (and importantly, independent of the synchronization plan chosen as long as it is S-valid) in section 4.5. Before describing the separate framework components, we first articulate the necessary underlying assumptions about input streams in section 4.1.

4.1 Input Event Ordering

In our model the input is partitioned in some number of input streams that could be distributed, i.e. produced at different locations. We assume that the implementation has access to *some* ordering relation O on pairs of input events (also denoted $<_O$), and the order of events is increasing along each input stream. This is necessary for cases where the *user-written* specification requires that events arriving in different streams are dependent, since it allows the implementation to have progress and process these dependent events in order. Concretely, in our setting O is implemented using *event timestamps*. Note that these timestamps do not need to correspond to real time, if this is not required by the application. In cases where real-time timestamps are required, this can be achieved with well-synchronized clocks, as has been done in other systems, e.g. Google Spanner [15].

Each event in a stream is a triple $\langle \sigma, ts, v \rangle$, containing an *implementation tag* σ, a timestamp ts, and its payload v. Implementation tags are the same as the tags defined in section 3.1, only augmented with an identifier of the input stream. This ensures that each implementation tag is unique to a particular input stream, while a tag can appear in multiple streams (in all of the streams with a corresponding implementation tag).

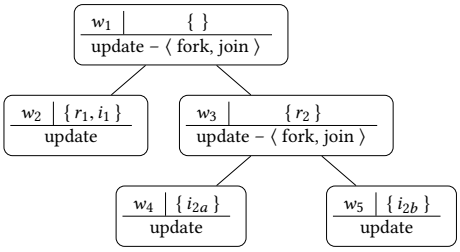


Figure 6: Example synchronization plan that can be derived from the specification in Figure 3 for two keys $k = 2$ and five input streams $r_1, i_1, r_2, i_{2a}, i_{2b}$. Each node contains the set of implementation tags that it is responsible for, a physical node, an update function, and a pair of fork-join functions if it has children.

4.2 Synchronization Plans

We propose *synchronization plans* as a representation of the implementations that can be derived from a specification in our API. Synchronization plans are binary tree structures that encode (i) parallelism, each node of the tree represents a sequential thread of computation that processes input events, and (ii) synchronization, parents have to synchronize with their children to process an event. Synchronization plans are inspired by prior work on concurrency models including fork-join concurrency [19, 32] and CSP [25]. An example synchronization plan for the specification in Figure 3 is shown in Figure 6. Each node has an id w_i, contains the set of implementation tags that it is responsible for, a state type (which is omitted here since there is only one state type `State`), and a triple of update, fork, join functions. Note that a node is responsible to process events from its set of implementation tags, but can potentially handle all the implementation tags of its children. The leaves of the tree can process events independently without blocking, while parent nodes can only process an input event if their children states are joined. Nodes without a descendant-ancestor relationship do not directly communicate, but instead indirectly learn about each other when their common ancestor joins and forks back the state.

A synchronization plan is *S-valid* if it represents an implementation that is correct with respect to the sequential specification S. S-validity is ensured by the end-to-end system if the specification is consistent (see Section 4.5). Formally, a valid plan has to satisfy the following syntactic properties, which are checked statically by our framework: (1) The state type of each node should be consistent with its update-fork-join triple and its implementation tags. The update must be defined on the node state type, and the fork-join pair should be defined for the state types of the node and its children. Both of these properties are satisfied in Figure 6, since there is only one state type that handles all tags. (2) Each pair of nodes that do not have an ancestor-descendant relation, should handle pairwise independent and disjoint implementation tag sets. This property represents the main idea behind our execution model; independent

events can be processed by different workers without any communication. Intuitively, in the example in Figure 6, by assigning the responsibility for handling tag r_2 to node w_3, its children can independently process tags i_{2a}, i_{2b} that are dependent on r_2.

4.3 Optimization Problem

As described in the previous section, a set of S-valid synchronization plans can be derived from a computation specification S. This decouples the optimization problem of finding a well-performing implementation, allowing it to be addressed by an independent optimizer (see fig. 2), which takes as input a description of the available computer nodes and the input streams. This design means that different optimizers could be implemented for different performance metrics (e.g. throughput, latency, network load, energy consumption).

The design space for optimizers is vast and thoroughly exploring it is outside of the scope of this work. For the purposes of evaluation, in this work we have implemented an optimizer based on a simple heuristic: it tries to generate a synchronization plan with a separate worker for each input stream, and then tries to place these workers in a way that minimizes communication between them. This optimizer assumes a network of computer nodes and takes as input estimates of the input rates at each computer node. It searches for an S-valid synchronization plan that maximizes the number of events that are processed by leaves; since leaves can process events independently without blocking. The optimizer first uses a greedy algorithm that generates a graph of implementation tags (where the edges are dependencies) and iteratively removes the implementation tags with the lowest rate until it ends up with at least two disconnected components. To better understand this let's look at the example of fig. 6, assuming that r_2 has the lowest rate, followed by r_1, i_1, i_{2a}, and i_{2b}. Since events of different keys are independent, there are two connected components in the initial graph—one for each key. Therefore the optimizer starts by separating them into two subtrees. It then recurses on each disconnected component, until there is no implementation tag left, ending up with the tree structure shown in Figure 6. Finally, the optimizer exhaustively tries to match this implementation tag tree, with a sequence of forks, in order to produce a valid synchronization plan annotated with state types, updates, forks, and joins.

4.4 Implementation

Each node of the synchronization plan can be separated into two components: an event processing component that is responsible for computation via executing `update`, `fork`, and `join` calls; and a mailbox component that is responsible for enforcing ordering requirements and synchronization.

Event Processing The worker processes execute the `update`, `fork`, and `join` functions associated with the tree node. Whenever a worker is handed a message by its mailbox, it first checks if it has any active children, and if so, it sends them a join request and waits until it receives their responses. After receiving these responses, it executes the join function to combine their states, executes the update function on the received event, and then executes the fork function on the new state, and sends the resulting states to its children. In contrast, a leaf worker just executes the update function on the received event.

Event Reordering The mailbox of each worker ensures that it processes incoming dependent events in the correct order by implementing the following selective reordering procedure. Each mailbox contains an event buffer and a timer for each implementation tag. The buffer holds the events of a specific tag in increasing order of timestamps and the timer indicates the latest timestamp that has been encountered for each tag. When a mailbox receives an event $\langle \sigma, ts, v \rangle$ (or a join request), it follows the procedure described below. It first inserts it in the corresponding buffer and updates the timer for σ to the new timestamp ts. It then initiates a cascading process of releasing events with tags σ' that depend on σ. During that process all dependent tags σ' are added to a dependent tag workset, and the buffer of each tag in the workset is checked for potential events to release. An event $e = \langle \sigma, ts, v \rangle$ can be released to the worker process if two conditions hold. The timers of its dependent tags are higher than the timestamp ts of the event (which means that the mailbox has already seen all dependent events up to ts, making it safe to release e), and the earliest event in each buffer that σ depends on should have a timestamp $ts' > ts$ (so that events are processed in order). Whenever an event with tag σ is released, all its dependent tags are added to the workset and this process recurses until the tag workset is empty.

Heartbeats As discussed in section 4.1, a dependence between two implementation tags σ_1 and σ_2 requires the implementation to process any event $\langle \sigma_1, t_i, v_i \rangle$ after processing all events $\langle \sigma_2, t_j, v_j \rangle$ with $t_j \leq t_i$. However, with the current assumptions on the input streams, a mailbox has to wait until it receives the earliest event $\langle \sigma_2, t_j, v_j \rangle$ with $t_j > t_i$, which could arbitrarily delay event processing. We address this issue with *heartbeat* events, which are system events that represent the absence of events on a stream. These are commonly used in other systems under different names, e.g. heartbeats [29], punctuation [48], watermarks [9], or pulses [44]. Heartbeats are generated periodically by the stream producers, and are interleaved together with standard events of input streams. When a heartbeat event $\langle \sigma, t \rangle$ first enters the system, it is broadcast to all the worker processes that are descendants of the worker that is responsible for tag σ. Each mailbox that receives the heartbeat updates its timers and clears its buffers as if it has received an event of $\langle \sigma, t, v \rangle$ without adding the heartbeat to the buffer to be released to the worker process.

4.5 Proof of Correctness

We show that *any* implementation produced by the end-to-end framework is correct according to the semantics of the stream processing API. First, Definition 4.1 formalizes the assumptions about the input streams outlined in Section 4.1, and Definition 4.2 defines what it means for an implementation to be correct with respect to a sequential specification. Our definition is inspired by the classical definitions of distributed correctness based on observational trace semantics (e.g., [35]). However, we focus on how to interpret the independent input streams as a sequential input, in order to model possibly synchronizing and order-dependent stream processing computations.

Definition 4.1. A *valid input instance* consists of k input streams (finite sequences) `u_1`, `u_2`, ..., `u_k` of type `List(Event | Heartbeat)`, and an order relation O on input events and heartbeats, with the following properties. (1) *Monotonicity:* for all i, `u_i` is in strictly increasing order according to $<_O$. (2) *Progress:* for all i, for each input event (non-heartbeat) `x` in `u_i`, for every other stream j there exists an event or heartbeat `y` in `u_j` such that `x` $<_O$ `y`.

The semantics of a specification given by the stream processing API is a function `spec: List(Event) -> Set(Out)`. The output specified by `spec` is produced incrementally (or *monotonically*): if `u` is a prefix of `u'`, then `spec(u)` is a subset of `spec(u')`. Define the *sort* function `sort`$_O$ `: List(List(Event | Heartbeat)) -> List(Event)` which takes k sorted input event streams and sorts them into one sequential stream, according to the total order relation O, and drops heartbeat events.

Definition 4.2. Given a specification `spec: List(Event) -> Set(Out)`, a distributed implementation is *correct* with respect to `spec` if for every valid input instance O, `u_1`, ..., `u_k`, the set of outputs produced by the implementation is equal to `spec(sort`$_O$`(u_1, ..., u_k))`.

We show that our framework is correct according to Definition 4.2 (due to space limitations, the complete proof is contained in an extended technical report).

5 EXPERIMENTAL EVALUATION

In this section we carry out a series of experiments to investigate the limitations of data-parallelism in existing stream processing systems. For applications where some form of synchronization is required, what benefits do DGS (our streaming API) and synchronization plans (the execution model) offer? We conduct our investigation on Apache Flink [9, 17], a representative high-performance dataflow-based stream processing system For our evaluation we design three applications that exhibit different forms of dependencies, which we describe in Section 5.1. In the rest of the section, we aim to evaluate the following questions:

- **Q1** For computations requiring synchronization, what are the throughput scalability limits of the parallelism exposed by existing stream processing APIs?
- **Q2** Can synchronization plans be used to achieve concrete throughput improvements (through additional parallelism) in existing stream processing systems?
- **Q3** What is the scalability of the synchronization plans that are generated automatically by our implementation of DGS? What throughput is achieved and what is the latency cost of synchronization?
- **Q4** Finally, for these differing methods of achieving data parallelism with synchronization, what sacrifices are made in terms of high-level design?

To answer **Q1** (Section 5.2), we investigate whether Flink's API can be used to achieve scalable data-parallel implementations for the three applications. We show that Flink reaches a scalability limit due to lack of support for arbitrary communication among parallel instances. In particular, it can only scale one of the three applications, which we verify by measuring the throughput as the number of distributed nodes increases.

To answer **Q2** (Section 5.3), we manually implement synchronization plans generated using our framework for the above applications on top of Flink. We compare their throughput with the maximum throughput achieved by the implementations from Q1 in the cases where Flink was unable to scale automatically. We show that synchronization plans can achieve higher throughput (x4-x8), which provides evidence that there are opportunities for better scalability in existing state-of-the-art systems.

To answer **Q3** (Section 5.4), we implement and evaluate DGSStream, an end-to-end prototype that implements DGS in Erlang [7]. Given a specification, DGSStream generates and deploys a synchronization plan. We write a specification for each of the three applications and we show that given a single specification our prototype can automatically produce implementations that scale with increased parallelism. We also show that our implementation pays only the expected linear cost in latency with the amount of parallelism.

Lastly, to answer **Q4** (Section 5.5), we first identify a set of criteria with which to evaluate high-level design trade-offs. We identify the following criteria. (i) *Parallelism independence:* is the program developed without regard to the number of parallel instances or does the behavior of the program change depending on the number of parallel instances? (ii) *API compliance:* does the program violate any assumptionsmade by the stream processing API? We then compare all of the programs from Q1, Q2, and Q3 in Flink and DGS with respect to these criteria. We show that using existing streaming APIs a user has to choose between (i) high throughput with the cost of violating the above criteria (by manually implementing a parallel implementation such as synchronization plans), or (ii) development convenience but low throughput (by developing their program using existing stream processing APIs). In contrast, DGS can achieve the best of both worlds, producing scalable parallel implementations in the form of synchronization plans without violating the above criteria.

Experimental Setup. We conduct the experiments in this section in a distributed execution environment consisting of a set of AWS EC2 instances. We use m6g.medium instances (1 core @2.5GHz, 4 GB) in the same region (us-east-2) and we increase the number of instances for the scalability experiments. In order to evaluate the true streaming performance of Flink we turn off fault tolerance and batching. This is accomplished in Flink by disabling checkpointing and dynamic adaptation, and setting buffer-timeout to 0. Communication between different nodes is managed by each respective system (the system runtime for Flink, and Erlang for DGSStream).

5.1 Applications Requiring Synchronization

We consider three applications that exhibit different processing dependencies and therefore require different forms of synchronization. All three of the applications do not perform CPU-heavy computation for each event so as to expose communication and underlying system costs in the measurements. The conclusions that we draw are still valid since a computation heavy application that would exhibit similar dependencies would scale even better with the addition of more processing nodes. The input for all three of these applications is synthetically generated.

Event-based Windowing: An *event-based window* is a window whose start and end is defined by the occurrence of certain events.

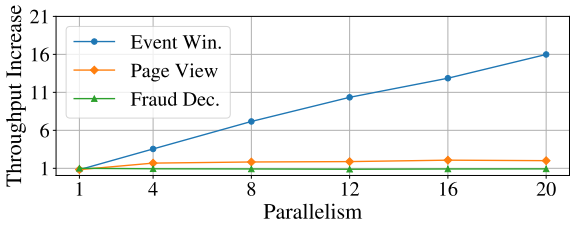
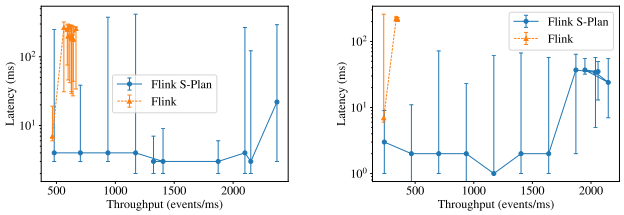


Figure 7: Flink implementations maximum throughput increase with respect to the sequential maximum throughput for the three applications that require synchronization.



(a) Page-view Join. **(b) Fraud Detection.**

Figure 8: Throughput (x-axis) and 10th, 50th, 90th percentile latencies on the y-axis for increasing input rates (from left to right) and 12 parallel nodes. Flink represents the parallel implementation produced automatically by Flink, and Flink S-Plan represents the synchronization plan implementation in Flink.

This results in a simple synchronization pattern where parallel nodes must synchronize at the end of each window. For this application, we generate an input consisting of several streams of integer *values* and a single stream of *barriers*. The task is to produce an aggregate of the values between every two consecutive barriers, where *between* is defined based on event timestamps. We take the aggregation to be the sum of the values. The computation is parallelizable if there are sufficiently more value events than barrier events. In the input to our experiments, there are 10K events in each value stream between two barriers.

Page-view Join: The second application is a example of a streaming join. The input contains 2 types of events: *page-view* events that represent visits of users to websites, and *update-page-info* that are used to update information and metadata for a particular website and also output the old info and metadata when processed by the program. All of these events contain a unique identifier identifying the website and the goal is to join page-view events with the latest metadata of the visited page to augment them for a later analysis. An additional assumption is that the input is not uniformly distributed among websites, but a small number of them receive most of the page-views. To simulate this behavior all but 2 pages receive a negligible number of views in our experiment input.

Fraud Detection: Finally, the third application is a bare-bones version of the ML fraud detection application (section 2), that has the same dependencies but the computation is simplified. The input contains *transaction* events and *rule* events both of which are integer values. On receiving a rule, the program outputs an aggregate of the transactions since the last rule and a transaction is considered fraudulent if it is equal modulo 1000 to the sum of the previous transactions (simulating the model retraining) and the last rule event. Similarly to event-based windowing there are 10K transactions in each transaction stream between every two rules.

5.2 Implementations in Flink's API (Q1)

In this section we investigate how Flink's API can be used to develop scalable parallel implementations for the aforementioned applications. We ran an experiment where we increased the worker nodes (and Flink work queues) and measured the maximum throughput of the resulting implementation for each application. The maximum throughput increase is shown in Figure 7.

Event-based Windowing: Flink's API supports a `broadcast` construct that can be used to send barrier events to all parallel instances of the implementation, therefore being able to scale with an increasing parallel input load. By transforming these barriers to Flink watermarks that indicate window boundaries, we can then aggregate values of each window in parallel, and finally merge all windows to get the global aggregate.

Page-view join: The input of this application allows for data parallelism across keys, in this case websites, but also for the same key since some keys receive most of the events. Unfortunately, existing streaming APIs (including Flink's) can produce implementations that parallelize the processing of disjoint keys (using sharding), but cannot produce an implementation that parallelizes both across and in the same key. Therefore we implemented this application using a standard keyed join, ensuring that the resulting implementation will be parallel with respect to keys.

Fraud Detection: As mentioned in Section 2, streaming APIs cannot support cross instance dependencies and therefore we can only develop a sequential implementation of this application using Flink's API.

Take-away (Q1): The streaming API of Flink, a state-of-the-art stream processing system cannot automatically produce implementations that scale throughput-wise for all applications that have synchronization requirements. More precisely, they cannot process events of the same key in parallel, and they cannot produce a parallel implementation for the ML fraud detection application shown in Section 2.2.

5.3 Implementation using Synchronization Plans (Q2)

We focus on the two applications that Flink cannot produce parallel implementations for, namely **page-view join** and **fraud detection**. We write a specification for these applications and we use our generation framework to produce a synchronization plan for a specific parallelism level (12 nodes). We then manually implement these synchronization plans in Flink and measure their throughput and latency compared to the parallel implementations that the systems produced in section 5.2. The results for both applications are shown in Figure 8.

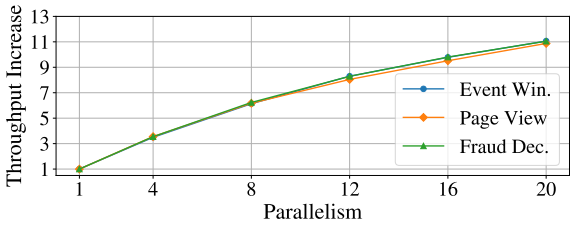


Figure 9: DGSStream maximum throughput increase with respect to the sequential maximum throughput for the three applications that require synchronization.

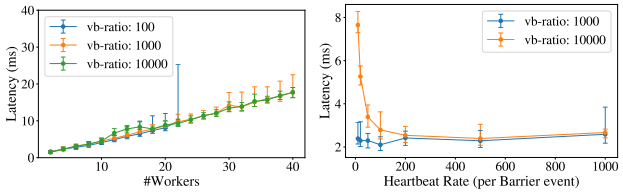


Figure 10: DGSStream latency (10/50/90 percentile) on the event-based windowing application for various configurations: Synchronization impact on latency (10/50/90 percentile) for various configurations: (a) Varying ratios of value to barrier events (different lines) and number of parallel nodes (x-axis). Heartbeat ratio is 1/100 the vb-ratio. The blue line stops after 22 workers. (b) Varying ratios of value to barrier events (different lines) and heartbeat rates (x-axis). Number of parallel nodes is fixed to 5.

Page-view join: The synchronization plan that we implement for this application is a forest containing a tree for each key (website) and each of these trees has leaves that process page-view events. Each time an update event needs to processed for a specific key, the responsible tree is joined, processes the event and then forks the new state back.

Fraud Detection: The synchronization plan that we implement for this application is a tree that processes rule events at its root and transactions at all of each leaves. The tree is joined in order to process rules and is then forked back to keep processing transactions.

Implementation in Flink: In order to implement the synchronization plans in Flink we need to introduce communication across parallel instances. We achieve this by using a centralized service that can be accessed by the instances using RMI. Synchronization between a parent and its children happens using two sets of semaphores J, F. A child releases its J semaphore and acquires its F semaphore when it is ready to join, and a parent acquires its children's J semaphores, performs the event processing, and then releases their F semaphores.

Take-away (Q2): As can be seen in Figure 8, synchronization plans can be used in applications that require synchronization to achieve higher throughputs (x4-x8 for 12 parallel nodes) than the parallel implementations produced by Flink using its API.

5.4 Implementation in DGS (Q3)

In order to evaluate the implementations that can be automatically generated using DGS we implement DGSStream, an end-to-end prototype in Erlang, and we develop a specification for each of the above applications. We then measure the throughput scalability of the implementations produced DGSStream.

Event-based Windowing + Fraud Detection: The specification for event-based windowing contains: (i) a sequential update function that adds incoming value events to the state, and outputs the value of the state when processing a barrier event, (ii) a dependence relation that indicates that all events depend on barrier events, and (iii) a `fork` that splits the current state in half, together with a `join` that adds two states to aggregate them. The specification for fraud detection is the same with the addition that the `fork` also duplicates the sum of the previous transaction and last rule modulo 1000.

Page-view Join: In addition to the sequential update function, the specification in DGS indicates that events referring to different keys are independent, and that page-view events of the same key are also independent. The `fork` and `join` are very similar to the ones in Figure 3 and just separate the state with respect to the keys.

Throughput Scalability: We measured the throughput scalability of the implementations produced by DGSStream for the above specifications by increasing the number of parallel nodes. The results are shown in Figure 9. Note that all three specifications scale very similarly even though they have exhibit widly varying dependencies and synchronization requirements.

Synchronication latency: We also conducted an experiment to evaluate the latency of synchronization plans. We studied three factors that affect latency: (i) the depth of the synchronization plan, (ii) the rate of events that are processed at non-leaf nodes of the plan, and (iii) the hearbeat rate. We ran the event-based windowing application with various configurations and the results are shown in Figure 10.

Take-away (Q3): DGS can be used to produce parallel implementations that scale throughput wise. The latency costs of these implementations scale linearly as long as synchronization does not happen too frequently.

5.5 Application Development Tradeoffs (Q4)

In this section we analyze all of the programs that we implemented in this section, assessing what tradeoffs each one of them makes. First, we analyze whether the programs contains any references to any information about the parallel instances (shards) of the implementation. Parallelism independence allows decoupling the program specification from its parallel implementation and is one of the main reasons why people use stream processing APIs. Second, we analyze whether the programs violate any assumptions made by the streaming API. An example of such an assumption is that there is communication across parallel instances—either via an external service or directly. Violating an assumption of the API leads to undefined behavior, possibly leading to an unexpected behavior from the underlying system. Finally, we check whether the

	Event Win.		Page-View			Fraud Det.		
Dev. Tradeoffs	F	DGS	F	FS	DGS	F	FS	DGS
Parallelism Indep.	✓	✓	✓	✗	✓	✓	✗	✓
API compliance	✓	✓	✓	✗	✓	✓	✗	✓
Throughput (12 nodes)	x10	x8	x2	x9	x8	x1	x9	x8

Table 1: Development tradeoffs for each program for each application. F refers to the Flink API implementation, FS refers to the synchronization plan implementation in Flink, and DGS refers to programs in our API.

programs correspond to efficient sequential implementations. This is important since it showcases whether a single implementation is flexible to perform well sequentially and in parallel, or whether the user needs to have a different efficient sequential and non-sequential implementation. Table 1 shows all the different tradeoffs that need to be made for each of the programs in this section together with the throughput increase for 12 nodes.

Take-away (Q4): DGS does not sacrifice parallelism independence or API compliance while still being able to produce scalable implementations.

6 CASE STUDIES

In this section, we evaluate DGS in the context of realistic applications. We consider the following questions:

(1) How does the performance achieved by the DGS implementation compare with handcrafted implementations?
(2) What is the additional programming effort to achieve automatic parallelization when using the DGS API?

To evaluate these questions we conduct two case studies on applications taken from the literature that have existing high-performance implementations for comparison. The first is a state-of-the-art algorithm for statistical outlier detection, and the second is a smart home power prediction task from the DEBS Grand Challenge competition. In answer to question (1), we are able to achieve comparable performance to the existing handcrafted implementations in both cases, in terms of throughput scalability for outlier detection and in terms of latency for the smart home task. In answer to question (2), this performance is achieved while putting minimal additional effort into parallelization: compared to 200-700 LoC for the sequential task, writing the fork and join primitives requires only an additional 50-60 LoC. These results support the feasibility of using DGS for practical workloads, with minimal additional programmer effort to accomplish parallelism.

6.1 Statistical Outlier Detection

Reloaded [42] is a state-of-the-art distributed streaming algorithm for outlier detection in mixed attribute datasets. It is a structurally similar to the fraud-detection example (see section 2.3) and therefore cannot be parallelized by existing systems. The algorithm assumes a set of input streams that contain events from the same distribution. Each input stream is processed independently by a different worker with the goal of identifying outlier events. Each worker constructs a local model of the input distribution and uses that to flag some events as potential outliers. Whenever a user (or some other event) requests the current outliers, the individual workers merge their states to construct a global model of the input distribution and use that to flag the potential outlier events as definitive outliers.

The sequential specification of the algorithm in DGS consists of approximately 700 lines of Erlang code—most of which is boilerplate to manage the algorithm data structures. The benefit of using our programming model can be seen when implementing the distributed version of the algorithm – the extension is just 50 LoC consisting of a `fork` that duplicates the global model and a `join` that aggregates the local models. In order to evaluate our generation framework we executed a network intrusion detection task from the original paper [23]. The goal of the task is to distinguish malicious connections from a streaming dataset of simulated connections on a typical U.S. Air Force LAN. Each connection is represented as an input event. In the experiment we varied the number of nodes from 1-8 and we measured the execution time speedup. We executed this experiment on a local server (Intel Xeon Gold 6154, 18 cores @3GHz, 384 GB) to be able to compare our results with the original paper. To simulate the network and measure network load we use NS3 [10]. The speedup achieved by our implementation is almost linear (7.3× for 8 nodes), while the speedup achieved by the handcrafted C++ implementation that was reported in the original paper is 7.7× for 8 nodes.

Take-away: The implementation produced by our model and framework achieves comparable speedup to a manually distributed implementation of a state-of-the-art outlier detection algorithm that cannot be parallelized by existing systems. The programming effort to achieve a distributed implementation with our framework amounts to 50LoC of a straightforward pair of `fork-join` primitives.

6.2 IoT Power Prediction

The DEBS Grand Challenge is an annual competition to evaluate how research solutions in distributed event-based systems can perform on real applications and workloads. The 2014 challenge [14, 28] involves an IoT task of power prediction for smart home plugs. As with the previous case study, our goal is to see if the performance and programmability of our model and framework can be used on a task where there are state-of-the-art results we can compare to.

The problem (query 1 of the challenge) is to predict the load of a power system in multiple granularities (per plug, per house, per household) using a combination of real-time and historic data. We developed a solution that follows the suggested prediction method, that is using a weighted combination of averages from the hour and historic average loads by the time of day. This task involves inherent synchronization: while parallelization is possible at each of the different granularities, synchronizaiton is then required to bring together the historic data for future predictions. For example, if state is parallelized by plug, then state needs to be joined in order to calculate a historic average load by household. Our specification is conceptually similar to the map from keys to counters in section 3, where we maintain a map of historical totals for various keys (plugs, houses, and households). The challenge input contains 29GB of synthetic load measurements for 2125 plugs distributed across 40 houses, during the period of one month. We executed our implementation on a subset of 20 of the 40 houses, which we sped

up by a factor of 360 so that the whole experiment took about 2 hours to complete. To compare with submissions to the challenge which were evaluated on one node, we ran a parallelized computation on one server node; to simulate the network and measure network load, we then used NS3 [10].

Programmability: In total, the sequential code of our solution amounted to slightly more than 200 LoC, and the distribution primitives (fork, join, dependence relation) were 60 LoC. We conclude the overhead to enable parallelization is small compared to the code to implement the computation itself.

Performance: Latency varied between 44ms (10th percentile), 51ms (median), and 75ms (90th), and the average throughput was 104 events/ms. These results are comparable to the ones reported by that year's grand challenge winner [40]: 6.9*ms* (10th) 22.5*ms* (median) 41.3 (90th) and 131 events/ms throughput. Note that although the dataset is the same, the power prediction method used was different in some solutions. In this application domain, our optimizer has the advantage of enabling edge processing: we measure only 362 MB of data sent over the network, in contrast to the 29 GB of total processed data.

7 RELATED WORK

Dataflow stream processing systems: Applications over streaming data can be implemented using high-performance, fault tolerant stream processing systems, such as Flink [9], Trill [11], Spark Streaming [49], Storm [18], Samza [41], Heron [31], MillWheel [4], and IBM Streams [24]. These systems offer APIs that do not allow for arbitrary synchronization in the context of parallelism. The need for synchronization has resulted in a number of extensions to their APIs, but they fall short of a general solution (see section 2).

Naiad [39] proposes *timely dataflow* in order to support iterative computations. Epochs in an iterative computation are a form of synchronization. However, parallel copies of a node do not communicate directly with each other even in an iterative dataflow model, so implementing our abstraction on an iterative model would require explicit messages to update and pass state, and extra nodes to enforce message ordering. We have shown that some iterative workloads (e.g. machine learning) can be programmed in DGS.

Noria [22] proposes *partially-stateful dataflow* with the goal of accelerating web applications. Noria guarantees eventually consistent state, and this in terms allows reads to be executed much faster, in parallel without blocking. Data parallelism for writes in Noria is achieved as usual using sharding. Our work is incomparable from Noria since DGS is aimed at applications where some dependencies between reads and writes need to be preserved (as specified in the dependence relation), but all independent writes can be processed in parallel even if they refer to the same key.

Distributed programming with synchronization: In the broader context of distributed systems, synchronization is a significant source of overhead for developers, and this has motivated some prior work. Some similarities with our work can be found in the domain of consistency for replicated data stores. RedBlue consistency [34] is based on classifying data store operations as either red or blue, if they require full synchronization or no synchronization, respectively. The difference of our work is that in our setting state updates are local instead of globally replicated, and the dependence relation can express more complex synchronization patterns. Another related domain is Conflict-Free Replicated Data Types [45], some of which are defined using state updates and state merges. In contrast to CRDTs, dependent events in our context are ordered, whereas in their setting all events updates must commute.

Correctness in stream processing: One feature of DGS and our generation framework compared to existing models is the correctness guarantee, which is provided if the user-provided specification is consistent (see sections 3.2 and 4.5). Researchers have previously pointed out [37, 44] that parallelization in distributed stream processing systems is not semantics-preserving, and have proposed methods to restrict parallelization so that it preserves the semantics. The first method [44] is based on categorizing operators for properties such as statefulness and selectivity; the type of parallelization that can be done then depends on these properties. The second [37] uses a type-based discipline to indicate possible parallelization of streams. The idea of dependency relations, which we use in DGS, comes from this work. However, their implementation only supports templates for commonly used patterns, and lacks a general programming model, a notion of correctness, and generation of a parallel implementation. Finally, recent work by Kallas et al. [30] proposes DiffStream, a differential testing library that can be used to test the correctness of parallel streaming applications with varying ordering requirements on their output streams.

8 CONCLUSION

We have presented DGS, a stream processing API that can be used to generate scalable parallel implementations for applications with complex dependencies and synchronization requirements. DGS offers flexible primitives to fork and join state, and a dependence relation encodes the synchronization and ordering requirements on the input. We also presented *synchronization plans* as a core execution model for synchronization, and we developed a framework that generates a provably correct and efficient implementation given a user-written DGS specification via synchronization plans. Our experimental evaluation in Flink sheds light on the limitations of sharding in dataflow-based systems to capture computations with both parallelism and synchronization in general, and shows that synchronization plans have the potential to improve throughput. Our experimental evaluation using our DGS prototype additionally shows that the synchronization plans can be generated automatically to scale with the amount of parallelism.

We view DGS as the first step in a new set of programming language and systems solutions for data processing where synchronization plays a central role. In future work we plan to develop a stream processing system that addresses implementation challenges related to our model such as (i) the management and communication of forked state in a distributed environment, (ii) fault tolerance, (iii) dynamic adaptation, and (iv) extending known techniques such as batching and backpressure in our setting.

REFERENCES

[1] [n. d.]. FLIP-8: Rescalable Non-Partitioned State - Apache Flink - Apache Software Foundation. https://cwiki.apache.org/confluence/display/FLINK/FLIP-8%3A+Rescalable+Non-Partitioned+State. https://cwiki.apache.org/confluence/display/FLINK/FLIP-8%3A+Rescalable+Non-Partitioned+State

[2] [n. d.]. KTable state stores and improved semantics - Apache Kafka - Apache Software Foundation. https://cwiki.apache.org/confluence/display/KAFKA/KIP-114%3A+KTable+state+stores+and+improved+semantics.

[3] [n. d.]. Side Inputs for Local Stores - Apache Samza - Apache Software Foundation. https://cwiki.apache.org/confluence/display/SAMZA/SEP-27%3A+Side+Inputs+for+Local+Stores.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*. 601–613.

[7] Joe Armstrong, Robert Virding, Claes WikstrÃűm, and Mike Williams. 1993. Concurrent Programming in Erlang.

[8] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *Proceedings of the 2019 International Conference on Management of Data*. 1757–1772.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[10] Gustavo Carneiro. 2010. NS-3: Network simulator 3. In *UTM Lab Meeting April*, Vol. 20. 4–5.

[11] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[12] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2010. High-performance dynamic pattern matching over disordered streams. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 220–231.

[13] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

[14] DEBS Conference. 2014. DEBS 2014 Grand Challenge: Smart homes. https://debs.org/grand-challenges/2014/. [Online; accessed April 23, 2019].

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: GoogleâĂŹs globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[16] 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

[17] Apache Software Foundation. 2019. Apache Flink. https://flink.apache.org/. [Online; accessed March 31, 2019].

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

[19] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 212–223.

[20] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: The System s Declarative Stream Processing Engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1123–1134. https://doi.org/10.1145/1376616.1376729

[21] Kahn Gilles. 1974. The semantics of a simple language for parallel programming. *Information processing* 74 (1974), 471–475.

[22] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 213–231.

[23] S. Hettich and S. D. Bay. 1999. KDDCUP 1999 dataset, UCI KDD Archive. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. [Online; accessed November, 2019].

[24] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K. L. Wu. 2013. IBM Streams Processing Language: Analyzing Big Data in motion. *IBM Journal of Research and Development* 57, 3/4 (2013), 7:1–7:11. https://doi.org/10.1147/JRD.2013.2243535

[25] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

[26] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter BodÃŋk, Leana Golubchik, Minlan Yu, Victor Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. https://www.microsoft.com/en-us/research/publication/videoedge-processing-camera-streams-using-hierarchical-clusters/

[27] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, UÇğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1379–1390.

[28] Zbigniew Jerzak and Holger Ziekow. 2014. The DEBS 2014 Grand Challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 266–269. https://doi.org/10.1145/2611286.2611333

[29] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. 2005. A heartbeat mechanism and its application in gigascope. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 1079–1088.

[30] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: Differential Output Testing for Stream Processing Programs. *Proceedings of the ACM on Programming Languages* OOPSLA (2020).

[31] 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 (SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[32] Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*. 36–43.

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

[34] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 265–278.

[35] Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.

[36] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. 2017. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–708.

[37] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G. Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. (2019), 670–685. https://doi.org/10.1145/3314221.3314580

[38] Frank McSherry. 2020. Timely Dataflow (Rust). https://github.com/TimelyDataflow/timely-dataflow/. [Online; accessed September 30, 2020].

[39] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[40] Christopher Mutschler, Christoffer Löffler, Nicolas Witt, Thorsten Edelhäußer, and Michael Philippsen. 2014. Predictive load management in smart grid environments. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 282–287.

[41] 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 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[42] Matthew Eric Otey, Amol Ghoting, and Srinivasan Parthasarathy. 2006. Fast distributed outlier detection in mixed-attribute data sets. *Data mining and knowledge discovery* 12, 2-3 (2006), 203–228.

[43] Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale. 2016. SamzaSQL: Scalable fast data management with streaming SQL. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1627–1636.

[44] S. Schneider, M. Hirzel, B. Gedik, and K. Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Trans. Comput.* 64, 2 (Feb 2015), 504–517. https://doi.org/10.1109/TC.2013.221

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[46] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A universal calculus for stream processing languages. In *European Symposium on Programming*. Springer, 507–528.

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

[48] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003), 555–568.

[49] 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 (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.

1145/2517349.2522737