Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees

Zuozhi Wang
UC Irvine
Irvine, United States
zuozhiw@ics.uci.edu

Shengquan Ni
UC Irvine
Irvine, United States
shengqun@ics.uci.edu

Avinash Kumar
UC Irvine
Irvine, United States
avinask1@ics.uci.edu

Chen Li
UC Irvine
Irvine, United States
chenli@ics.uci.edu

ABSTRACT

A computing job in a big data system can take a long time to run, especially for pipelined executions on data streams. Developers often need to change the computing logic of the job such as fixing a loophole in an operator or changing the machine learning model in an operator with a cheaper model to handle a sudden increase of the data-ingestion rate. Recently many systems have started supporting runtime reconfigurations to allow this type of change on the fly without killing and restarting the execution. While the delay in reconfiguration is critical to performance, existing systems use epochs to do runtime reconfigurations, which can cause a long delay. In this paper we develop a new technique called Fries that leverages the emerging availability of fast control messages in many systems, since these messages can be sent without being blocked by data messages. We formally define consistency in runtime reconfigurations, and develop a Fries scheduler with consistency guarantees. The technique not only works for different classes of dataflows, but also works for parallel executions and supports fault tolerance. Our extensive experimental evaluation on clusters show the advantages of this technique compared to epoch-based schedulers.

PVLDB Reference Format:
Zuozhi Wang, Shengquan Ni, Avinash Kumar, and Chen Li. Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees. PVLDB, 16(2): 256 - 268, 2022.
doi:10.14778/3565816.3565827
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Texera/Fries-Flink.

1 INTRODUCTION

Big data systems are widely used to process large amounts of data. Each computation job in these systems can take a long time to run, from hours to days or even weeks to finish. Applications that require timely processing of input data often use pipelined dataflow execution engines [1, 8, 9], for example, in the scenarios of processing real-time streaming data, or answering queries progressively to provide early results to users. In these applications, when a long running job continuously processes ingested data, developers often need to change the computing logic of the job without disrupting the execution, as illustrated in the following example.

Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565827

Consider a data-processing pipeline for payment-fraud detection shown in Figure 1. This simplified dataflow resembles many real-world applications [12, 29]. A stream of payment tuples is continuously ingested into the dataflow, with each tuple containing payment information such as customer, merchant, and amount. The dataflow uses two machine learning (ML) operators *FC* and *FM* to detect fraud based on customer and merchant information.

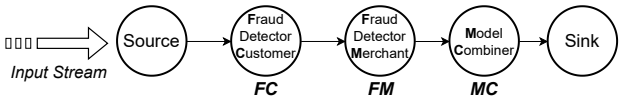

Figure 1: An example data-processing pipeline for fraud detection processing continuously ingested data.

Consider two example use cases in this dataflow. *Use case 1: fixing loopholes in operators.* After observing unexpected tuples from the Sink, the user identifies a loophole in the operator *FM*. She wants to update this operator to incorporate new rules to fix the loophole, without stopping the execution. *Use case 2: handling surges of data arrival rate.* Suppose the data arrival rate suddenly increases, and as a result, the end-to-end processing latency becomes larger. The user finds that the ML operator *FM* is the bottleneck. To reduce the latency, she wants to "hot-replace" the expensive ML model (e.g., a neural network) with a lightweight model (e.g., a decision tree) to improve its performance, thus reduce the processing latency. Again, she wants to make the change without stopping the execution. These examples show the importance of allowing developers to change the dataflow execution "on the fly." We call such changes *runtime reconfigurations*. This problem has gained a lot of interest in the research areas of software engineering [28], mobile computing [18, 30], and distributed systems [19, 21]. Recently, users of dataflow systems also show the need for runtime reconfigurations [14, 15, 29] and more systems start supporting this important feature [7], such as Amber [20], Chi [22], Flink [32], and Trisk [23].

Naturally there is a delay from the time a user requests a reconfiguration to the time its changes take effect in the target operators. This delay is critical to the performance of the system. For example, in use case 1, the user wants to fix the loophole as soon as possible since a large reconfiguration delay can cause financial losses. In use case 2, a large delay in mitigating the surge can cause the system to suffer longer in terms of long latency and wasting of computing resources. Thus we want this delay to be as low as possible.

A main limitation of existing systems supporting runtime reconfigurations is that they could have a long reconfiguration delay. In these systems, after a reconfiguration request is submitted, they need to wait for all the in-flight tuples to be processed by those

target reconfiguration operators, as well as those earlier operators in the dataflow, before the requested changes can be applied on the target operators. This delay could be very long, when there are many in-flight tuples, or some of these operators are expensive, especially for operators using advanced machine learning models and those implemented as user-defined functions (UDF's).

In this paper, we develop a novel technique, called "Fries," to perform runtime reconfigurations with a low delay. It leverages the emerging availability of fast control messages in many systems recently. A *fast control message*, "FCM" for short, is a message exchanged between the controller in the data engine and an operator without being blocked by data messages. Figure 2 shows an example of handling a reconfiguration request of two operators *FM* and *MC* using FCM's. Upon a reconfiguration request, the controller sends an FCM to each of the two operators, and each of them applies the new configuration immediately after receiving the message. Since FCM's are sent separately from data messages, these changes can reach the target operators much faster.

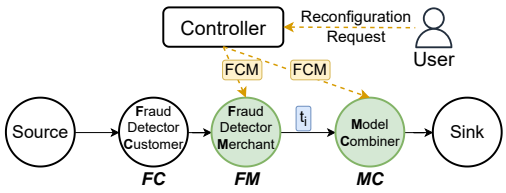

Figure 2: Handing a runtime reconfiguration of operators *FM* and *MC* using fast control messages (FCM's).

We will show in Section 4.1 that the naive way of using FCM's can cause consistency issues in Figure 2. It has unexpected side effects, e.g., producing incorrect results on the output tuples, or even causing the operator *MC* to crash. This example shows several challenges in developing Fries: 1) What is the meaning of "consistency" in this reconfiguration context? 2) How to ensure this consistency while reducing their delay? 3) How to deal with different types of operators and support parallel executions? We study these challenges and make the following contributions.

- We analyze epoch-based reconfiguration schedulers and show their limitations (Section 3).
- We formally define consistency of a reconfiguration based on transactions (Section 4).
- We first consider a simple class of dataflows that have one-to-one operators only, and develop a Fries scheduler that guarantees consistency (Section 5).
- We then consider the general class of dataflows with one-to-many operators, and extend the Fries scheduler (Section 6).
- We extend Fries to more general cases, such as dataflows with blocking operators and multiple workers (Section 7).
- We conduct an extensive experimental study to evaluate Fries in various scenarios and show its superiority compared to epoch-based schedulers (Section 8).

1.1 Related Work

Reconfiguration systems. Recently, many data-processing systems have started to support reconfigurations. Flink [8] supports reconfiguration by taking a savepoint [13], killing the running job, then restarting the job with the new configuration. This approach is disruptive to the dataflow execution. Spark Streaming [2, 36] uses a mini-batch-based execution strategy and supports reconfiguration between mini-batches. Chi [22] enables runtime reconfiguration by propagating epoch markers in its data stream. Trisk [23] provides an easy-to-use programming API for reconfigurations. The approaches in these systems are all based on epochs, which can have a long reconfiguration delay, as analyzed in Section 3. Fries relies on FCM's to perform reconfigurations with a low delay. Noria [16] uses dataflows to incrementally maintain materialized views. It supports reconfigurations of view definitions, which require the new views to be recomputed from entire base tables. In Fries, an update of a dataflow only affects the future tuples. The input tuples that are already processed by the dataflow are not affected.

Re-scaling systems. Some systems [11, 17, 25] support updating the dataflow for re-scaling. For example, Megaphone [17] based on timely dataflow [26] supports a fine-granularity re-scaling and Rhino [25] based on Flink supports re-scaling with very large states. Fries focuses on reconfiguring the computation functions of operators, which is different from re-scaling.

Transactions in dataflow systems. S-Store [24] and the work in [5] are systems that allow streaming dataflows and OLTP workloads to access a shared mutable state. Although both systems do not support reconfigurations directly, we could map a reconfiguration to these systems. S-Store defines transactions on the processing of each input batch on a single operator. This model cannot express our consistency requirements in reconfigurations. The work in [5] treats a dataflow as a black box, thus it has the limitation of not being able to utilize the properties of the dataflow and its operators to reduce the reconfiguration delay. Fries can do so to achieve this reduction. Both earlier systems are only on a single node, while the Fries scheduler can run on a distributed engine on a cluster.

Transactions in database systems. Transactions are widely studied in traditional database systems (e.g., [3, 4, 34]). A uniqueness in transactions in our work is that they treat operations in a reconfiguration as a separate transaction, which is handled differently from data transactions. In addition, Fries does optimizations by utilizing special properties in our problem setting, including the DAG shape of a dataflow, and types of operators, e.g., one-to-one and one-to-many. Moreover, the Fries scheduler uses FCM's and epoch markers to schedule transactions without locking.

2 PROBLEM SETTINGS

2.1 Data-Processing Model

A data-processing system runs a computation dataflow job represented as a directed acyclic graph (DAG) of operators. Each operator receives tuples from its input edges, processes them, and sends tuples through its output edges. An operator contains a computation function f represented as

$$f : (s, t) \rightarrow \big(s', \{(t'_1, o'_1), \ldots, (t'_n, o'_n)\}\big).$$

The function processes a tuple t at a time with a state s of the operator, produces a set of zero or more output tuples $\{t'_1, \ldots, t'_n\}$, where each tuple t'_i has a receiving operator o'_i. The operator also

updates its state to s'. The system has a module called *controller* that manages the execution of the job, handles requests from the user, and exchanges messages with operators during the execution.

For simplicity, we first focus on dataflows under the following assumptions. (1) A dataflow contains pipelined operators only, such as selection, projection, union, and other tuple-at-a-time operators. We consider a class of join operators where the operator first collects all the tuples from one input (e.g., the "build" input of a hash join), then starts processing tuples from the other input (e.g., the "probe" input of a hash join). We consider the processing of tuples from the second input of join. (2) Each operator has a single worker. We relax these assumptions in Section 7.

As an example, consider a data-processing pipeline for payment-fraud detection shown in Figure 1. The example dataflow uses two machine learning (ML) operators for fraud detection. The first one, denoted as *FC*, keeps a state of the 5 recent tuples of each customer. For each input tuple, *FC* updates the state and feeds the 5 recent tuples of the customer into an ML model. The predicted probability $p_c(5)$ is attached as a new column of the tuple. The second one, denoted as *FM*, keeps a state of the 5 recent tuples of each merchant. Similarly, it uses an ML model to generate a predicted probability $p_m(5)$, and attaches it as a new column of the tuple. Finally, the model combiner *MC* uses $p_c(5)$ and $p_m(5)$ of each tuple to compute the final average probability with the weights $[0.4, 0.6]$.

2.2 Runtime Reconfiguration

Definition 2.1 (Runtime reconfiguration). During the execution of a dataflow, an update to the computation functions of its operators is a *runtime reconfiguration* of this execution.

Formally, a reconfiguration $\mathcal{R}$ is a set of operators with a function update $\mu(o_i)$ for each operator o_i, i.e.,

$$\mathcal{R} = \{(o_1, \mu(o_1)), \ldots, (o_n, \mu(o_n))\}.$$

Each operator o_i has a *function-update operation* $\mu(o_i)$. This operation applies a pair $\langle f'_{o_i}, \mathcal{T}_{o_i} \rangle$ to the operator, where f'_{o_i} is a new computation function of the operator. $\mathcal{T}_{o_i}$ is a state transformation that converts the operator's original state s to a new state $s^* = \mathcal{T}_{o_i}(s)$, which can be consumed by f'_{o_i}. In this paper, we consider the case where there is one reconfiguration at a time.

In the running example, suppose the user identifies a flaw in the dataflow and wants to reconfigure the two operators *FM* and *MC*. Specifically, the user wants to change *FM* to output an additional probability value $p_m(10)$, which is predicted using the 10 recent tuples of each merchant. The operator *MC* needs to be updated to combine all three probabilities ($p_c(5)$, $p_m(10)$, and $p_m(5)$) with the new weights $[0.4, 0.4, 0.2]$. Table 1 shows the old and new configurations of the two reconfiguration operators.

Table 1: Operator executions during a reconfiguration.

	FM's output	*MC* weights
Old configuration	$p_m(5)$	[0.4, 0.6]
New configuration	$p_m(10)$, $p_m(5)$	[0.4, 0.4, 0.2]

Note that the new configuration of an operator can require a state different from that of the old configuration. In this case, the reconfiguration can use a state transformation to migrate the old state to the new one. For example, the old configuration of operator *FM* keeps the last 5 payment tuples for each merchant. However, the new configuration of *FM* needs the last 10 tuples for each merchant. The user provides a state transformation $\mathcal{T}$ for operator *FM*, to instruct the system in transferring operator *FM*'s old state to the new one. In this example, the user chooses to fill the new state with the 5 tuples from the old state and 5 additional *null* values.

3 EPOCH-BASED RECONFIGURATION SCHEDULERS AND LIMITATIONS

In this section, we explain epoch-based reconfiguration schedulers and show their limitation of long delays.

3.1 Epoch-Based Schedulers

Dataflow epoch. A stream of tuples processed by the system can be divided into consecutive sets of tuples, where each set is called an *epoch* [6]. One way to create epochs is to use epoch markers. At the start of a new epoch, an epoch marker is injected to each source operator. The epoch marker is propagated along the data stream using the following protocol [6]. When an operator receives an epoch marker from an input channel, it performs epoch alignment by waiting for all its inputs to receive an epoch marker, then sends the marker downstream. Figure 3 shows two epochs during the execution of the fraud-detection dataflow. An epoch marker injected between t_4 and t_5 divides the input stream into two epochs. The epoch marker indicates the end of epoch 1 and the start of epoch 2.

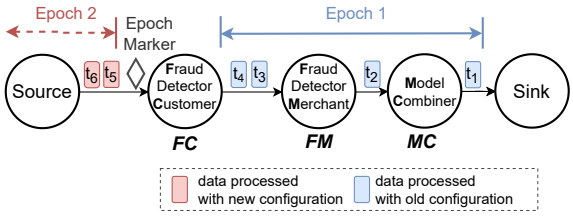

Figure 3: An epoch-based reconfiguration scheduler in Chi [22]. It uses an epoch barrier to apply the new configuration to operators *FM* and *MC* at the start of Epoch 2.

Epoch-based reconfiguration schedulers. An epoch-based reconfiguration scheduler handles a reconfiguration request by applying the new configuration of an operator between two epochs. Considering the aforementioned method to generate epochs, the following is an implementation adopted by Chi [22]. We call this implementation "Epoch Barrier Reconfiguration" scheduler, or "EBR" in short. Upon a reconfiguration request, the controller starts a new epoch and piggybacks the reconfiguration in the epoch marker. When a reconfiguration operator receives epoch markers from all its inputs, it applies the new configuration. The operator then processes the input tuples in the next epoch using the new configuration. Figure 3 shows the process of handling a reconfiguration of operators *FM* and *MC* using the EBR scheduler. When operator *FM* receives the epoch marker, it applies the new configuration, and propagates the marker to operator *MC*. When operator *MC* receives the epoch marker, it also applies the new configuration.

3.2 Limitations: Long Reconfiguration Delays

A major limitation of epoch-based reconfiguration schedulers is a long reconfiguration delay, which is from the time a request is submitted to the time the new configuration takes effect in the target operators. In particular, the system needs to process all the in-flight tuples before the new epoch. Take the EBR scheduler in Figure 3 as an example. Operator *FM* needs to finish processing the in-flight tuples t_3 and t_4. In general, this delay could be long due to the following reasons. First, the dataflow can contain multiple expensive operators that make the processing of an epoch slow. Second, the number of in-flight tuples could be large, especially when the system is under high workload. We may want to reduce the number of in-flight tuples by decreasing the buffer size. However, a smaller buffer can be easily filled by a minor fluctuation in the input ingestion rate. When the buffer is full, the system triggers back-pressure, which can decrease the throughput. Moreover, a small buffer size causes the networking layer to transmit data in small batches, which introduces additional transmission overhead.

4 SCHEDULING RECONFIGURATIONS USING FAST CONTROL MESSAGES

In this section, we introduce a new type of reconfiguration schedulers based on fast control messages (FCM's). We present a naive scheduler and show its issues. We then formally define consistency of a reconfiguration.

Definition 4.1 (Fast Control Message). A *fast control message*, "FCM" for short, is a message exchanged between the controller and an operator without being blocked by data messages.

There are many ways to implement fast control messages. One approach is to set up a new communication channel between the controller and an operator. The channel is separate from existing data channels, and the FCM can bypass data messages. Another way is to transmit the FCM using existing data channels, but assigning a higher priority to the FCM. The FCM is first sent to a source operator of the workflow, then propagated along the edges to the target operator , and it bypasses data messages in each data channel.

4.1 FCM-based Schedulers

Naive FCM scheduler. A main benefit of using FCM's to schedule reconfigurations compared to epoch-based schedulers is that FCM's have a much smaller delay. A naive scheduler leverages this benefit as follows. The controller sends an FCM directly to each reconfiguration operator. When an operator receives an FCM, it applies the new configuration immediately after finishing the processing of its current tuple. We use Figure 2 to explain how the naive scheduler works in a reconfiguration of two operators *FM* and *MC*. Using this scheduler, the controller sends an FCM directly to each of the two operators *FM* and *MC*. The FCM carries the new function f' and the state transformation $\mathcal{T}$ of the corresponding operator. These operators update their configuration after receiving their FCM.

While this naive scheduler has a low reconfiguration delay, it could generate an undesirable reconfiguration schedule. Notice that the scheduler does not coordinate the updates to these two operators that run independently. Consider the in-flight tuple t_i, which is processed by *FM* using its old configuration. Suppose the *MC* switches to the new configuration before the arrival of t_i. Then tuple t_i is processed by *MC* using its new configuration. The tuple contains two probability values $p_c(5)$ and $p_m(5)$, but the new configuration of *MC* expects three probability values. This schema mismatch could have unexpected side effects, such as producing an incorrect result, or even causing the operator *MC* to crash. This example shows the importance for the reconfiguration to be performed in a synchronized manner. In particular, we want a tuple to be processed by the two reconfiguration operators either using the old configuration or using the new configuration.

FCM multi-version scheduler. To ensure a tuple is processed by the same configuration of multiple operators, we can use the following FCM-based multi-version scheduler that maintains multiple configurations of an operator at the same time. The controller first sends an FCM to each reconfiguration operator. Each operator keeps both the old configuration and the new one. After all operators have received the FCM, each source operator increments its version number, which is tagged to each source tuple. For each input tuple, an operator checks the tuple's tagged version number, chooses the corresponding configuration version to process the tuple, and tags the same version number to the output tuples. As an example, in Figure 4, after the new configuration is sent to operator *E*, the source operators then tag subsequent output tuples t_3 and t_4 with the new version v_2.

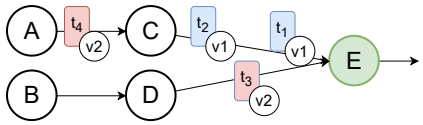

Figure 4: Using an FCM multi-version scheduler, an operator processes a tuple based on its version tag.

This scheduler has two problems. First, each reconfigured operator may need to keep two sets of states for two configurations, and these states could be very large (e.g., large hash tables or machine learning models). Second, this scheduler still suffers from a possible high reconfiguration delay. In particular, similar to the case of the EBR scheduler, there can be a large amount of in-flight tuples that are already tagged with the old version and they still need to be processed with the old configuration (e.g., t_1 and t_2 in Figure 4).

4.2 Reconfiguration Consistency

We formally define the consistency requirements in this context. At a high level, we treat the processing of a single source tuple by multiple operators as one *transaction*, and a reconfiguration as another transaction. We use conflict-serializability to define the consistency of a schedule of a reconfiguration.

Definition 4.2 (Scope of a source tuple). The *scope* of a source tuple t of a dataflow W, denoted as $\mathcal{S}(W, t)$, is a pair $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$, where $\mathcal{S}$ is a set of tuples and $\preccurlyeq_{\mathcal{S}}$ is a partial order on $\mathcal{S}$, defined as follows:

(1) The source tuple is in $\mathcal{S}$.
(2) For each tuple s in $\mathcal{S}$, if an operator processes the tuple s and produces zero or more output tuples $\{s'_1, \ldots, s'_n\}$, all the produced tuples are also in $\mathcal{S}$. For each tuple s'_i, we have the order $s \prec s'_i$ in $\preccurlyeq_{\mathcal{S}}$.

For instance, in Figure 5, a source tuple t is ingested into the dataflow from the source operator A and processed by operators C, D, E, F, and H. The scope of t includes the tuples on the highlighted edges and their partial order defined as their edges on the DAG.

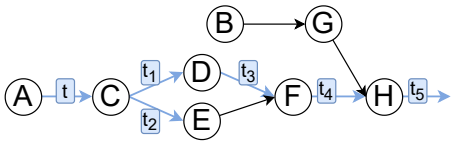

Figure 5: Scope of a source tuple in a dataflow.

Definition 4.3 (Data operation). The *data operation* of a tuple s is the processing of s by its receiving operator o, denoted as $\phi(s, o)$.

Definition 4.4 (Data transaction). For a dataflow W and a source tuple t in W, let $(\mathcal{S}, \preccurlyeq_{\mathcal{S}})$ be the scope of t. The *data transaction* of t is a pair $(\Phi, \preccurlyeq_{\Phi})$, where Φ is the set of data operations of the tuples in $\mathcal{S}$, and $\preccurlyeq_{\Phi}$ is a partial order on Φ. For two data operations $\phi(t_i, o_i)$ and $\phi(t_j, o_j)$ in Φ, we have $\phi(t_i, o_i) \prec \phi(t_j, o_j)$ in $\preccurlyeq_{\Phi}$ if and only if $t_i \prec t_j$ is in $\preccurlyeq_{\mathcal{S}}$.

For instance, in Figure 2, tuple t has the following data transaction T_1:

$$T_1 : [\phi(t, FC), \phi(t, FM), \phi(t, MC)].$$

In the data transaction, "$\phi(t, FC)$" is a data operation representing the processing of this tuple t by the FC operator.

Definition 4.5 (Function-update transaction). The *function-update transaction* of a reconfiguration $\mathcal{R} = \{(o_1, \mu(o_1)), \ldots, (o_n, \mu(o_n))\}$ on a dataflow W is the set $\{\mu(o_1), \ldots, \mu(o_n)\}$, where each $\mu(o_i)$ is a function-update operation in $\mathcal{R}$.

For instance, the reconfiguration in Figure 2 has the following function-update transaction T_2:

$$T_2 : \{\mu(FM), \mu(MC)\}.$$

In the function-update transaction, "$\mu(FM)$" is a function-update operation representing that the operator FM switches to the new configuration. Note that the order of different operations in a function-update transaction does not matter because they update different operators and are independent of each other.

Definition 4.6 (Conflicting operations). A data operation $\phi(t, o)$ and a function-update operation $\mu(o')$ are said to be *conflicting* if $o = o'$, i.e., they are on the same operator. They are said to be *not conflicting* if $o \neq o'$.

For instance, in Figure 2, operations $\phi(t, FM)$ and $\mu(FM)$ are conflicting because they are on the same operator. Operations $\phi(t, FC)$ and $\mu(FM)$ are not conflicting as they are on different operators.

Definition 4.7 (Schedule). A *schedule* of a set of transactions $T_1, \ldots, T_k$ is the set of all the operations in those transactions with a *partial order*. The schedule is called *serial* if for each pair of transactions T_i and T_j, T_i's operations in the schedule are either all before those in T_j or all after those in T_j.

In this paper we only consider schedules that include one function-update transaction and many data transactions.

Definition 4.8 (Conflict-equivalence). Two schedules S_1 and S_2 of the same set of transactions are said to be *conflict-equivalent* if $\forall o_i, o_j \in S_1$, if o_i and o_j are conflicting, and o_i is before o_j in S_1, then o_i is also before o_j in S_2.

Definition 4.9 (Conflict-serializable). A schedule is said to be *conflict-serializable* if it is conflict-equivalent to a serial schedule of the same set of transactions.

In the rest of the paper, when a partial order of a data transaction or a schedule defines a total order, for simplicity, we just show the transaction or the schedule as a sequence. We use the running example in Figure 1 to explain these concepts.

- S_1 is a schedule of the two transactions T_1 and T_2:

  $S_1 : [\phi(t, FC), \mu(FM), \phi(t, FM), \mu(MC), \phi(t, MC)]$.
- S_2 is a serial schedule of the two transactions:

  $S_2 : [\mu(FM), \mu(MC), \phi(t, FC), \phi(t, FM), \phi(t, MC)]$.

  In particular, all T_2's operations in this schedule are before those in T_1.
- S_1 and S_2 are conflict-equivalent. For example, for the conflicting pair $\mu(FM)$ and $\phi(t, FM)$, the former is before the latter in both schedules.
- S_1 is conflict-serializable because it is conflict-equivalent to the serial schedule S_2.
- S_3 is not a conflict-serializable schedule:

  $S_3 : [\phi(t, FC), \phi(t, FM), \mu(FM), \mu(MC), \phi(t, MC)]$.

  We can show that S_3 is not conflict-equivalent to any serial schedule. Intuitively, it has two pairs of conflicting operations, namely $[\phi(t, FM), \mu(FM)]$ and $[\mu(MC), \phi(t, MC)]$, and their corresponding transaction orders are different.

S_3 is the "bad" schedule generated by the naive FCM scheduler in Section 4.1, in which tuple t is processed using the old configuration of FM and the new configuration of MC. Schedule S_1 is a "good" schedule since t is processed entirely using the new configurations of both operators FM and MC and the aforementioned schema-mismatch issue does not happen.

Consistency of epoch-based schedulers. Consider the example in Figure 1. The aforementioned schedule $S1$ in Section 4.2 is produced by the EBR epoch-based scheduler, where the epoch marker is propagated before tuple t. We can show that the EBR approach can always produce a conflict-serializable schedule. We can also show that in general, an epoch-based scheduler always produces conflict-serializable schedules.

5 DATAFLOWS WITH ONE-TO-ONE OPERATORS ONLY

In this section, we consider the case where a dataflow contains one-to-one operators only. We propose a scheduler called Fries, which uses FCM's to achieve low reconfiguration delay and still guarantees conflict-serializability of produced schedules.

Definition 5.1 (One-to-one operator). An operator is called *one-to-one* if its processing function emits at most one (tuple, receiving operator) pair for each input tuple.

This type includes operators such as projection, filter, map function, equi-join on key attributes, and union.

Definition 5.2 (One-to-many operator). An operator is called *one-to-many* if its processing function can emit more than one output (tuple, receiving operator) pair for an input tuple.

This type includes operators such as join on non-key attributes and flatten function. In the rest of this section, we consider dataflows where all operators in the dataflow are one-to-one.

5.1 Conflict-Serializable Schedules Produced by the Naive FCM-based Scheduler

Section 4.1 shows an example dataflow and a reconfiguration where the naive FCM-based scheduler produces a non-conflict-serializable schedule. Next we use an example to show that the naive scheduler can still guarantee conflict-serializability for some types of dataflows and reconfigurations.

Example 5.3. Suppose we want to use the naive FCM-based scheduler to handle a reconfiguration of the two operators C and D as shown in Figure 6. Operator X is a one-to-one operator that splits the output tuples to operators C and D. In this case, we have a data transaction $T_3 = [\phi(t_1, X), \phi(t_1, C])$, another data transaction $T_4 = [\phi(t_2, X), \phi(t_2, D])$, and a function-update transaction $U = [\mu(C), \mu(D)]$. The controller sends two separate FCM's to C and D. Consider a possible schedule with T_3, T_4, and U:

$$S_4 : [\phi(t_1, X), \mu(C), \phi(t_1, C), \phi(t_2, X), \mu(D), \phi(t_2, D)].$$

Schedule S_4 is conflict-serializable because it is conflict-equivalent to the serial schedule $[U, T_3, T_4]$. Interestingly, we can show that all schedules produced by the naive FCM-based scheduler in Figure 6 are conflict-serializable.

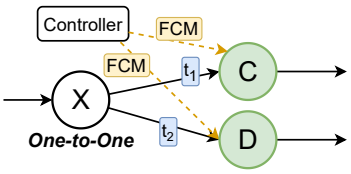

Figure 6: An example dataflow with a reconfiguration on operators C and D. The naive FCM-based scheduler always produces a conflict-serializable schedule.

One might wonder why the two examples in Figure 2 and Figure 6 are different in the conflict-serializability of the produced schedules. The main reason is that in Figure 2, a tuple can be processed by operators *FM* and *MC*, and both of them are in the reconfiguration. But there is no synchronization between the data operations and the function-update operations, causing the non-conflict-serializability. While in Figure 6, a tuple is processed by only one of the two paths through either C or D. On each path, there is a single operator in the reconfiguration, thus the data operations and the function-update operations are always synchronized.

Next, we introduce a concept called "minimal covering sub-DAG," which is used to represent the synchronization components. We then describe the Fries scheduler using this concept, and prove that this scheduler can always produce a conflict-serializable schedule.

5.2 Minimal Covering Sub-DAG (MCS)

Definition 5.4 (Minimal covering sub-DAG). Given a DAG $G = (V, E)$, and a set of vertices $M \subseteq V$, a minimal covering sub-DAG $G' = (V', E')$ is defined as follows:

(1) $M \subseteq V'$;
(2) $\forall A, B \in M$, if there is a path from A to B, then all the vertices and edges on the path are in V' and E', respectively;
(3) G' is minimal, i.e., no proper sub-DAG of G' can satisfy the above two conditions.

Figure 7 shows the minimal covering sub-DAG for the dataflow graph in Figure 5 and the set of operators $\{C, F, G\}$ in the reconfiguration. The sub-DAG is: $V' = \{C, D, E, F, G\}$ and $E' = \{C \to D, C \to E, D \to F, E \to F\}$. In general, we can show that there is a unique MCS given a DAG and a set of vertices, and we can compute the MCS using an algorithm with an $O(V + E)$ time complexity.

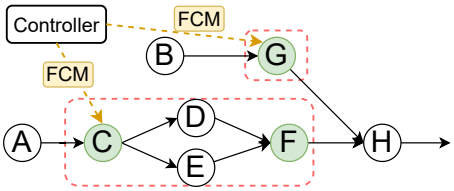

Figure 7: Two components of the minimal covering sub-DAG used in the Fries scheduler are highlighted in red.

5.3 The Fries Scheduler

The Fries scheduler uses components of the MCS to schedule the reconfiguration. A *component* is a maximal sub-DAG of the MCS where every pair of vertices in the component are connected by a path, ignoring the direction of edges. For example, the sub-DAG in Figure 7 has two components, each marked in a red box. The components of the MCS can be also computed using an algorithm [10] with an $O(V + E)$ time complexity.

The Fries scheduler is formally described in Algorithm 1. We first construct the minimal covering sub-DAG from the original dataflow DAG and operators in the reconfiguration (lines 1 and 2). We compute the components within the MCS (line 3). For each component in the MCS, the controller sends an FCM to the "head" operators, i.e., those with no input edges in the component. The head operators then start propagating an epoch marker within the component (lines 4 to 6). Specifically, when an operator receives an epoch marker, it performs marker alignment on the input edges in its component. An operator sends an epoch marker only to its downstream operators in its component.

As an example, in Figure 7, the controller sends an FCM to operator C, which is the only head operator of the first component. The controller also sends an FCM to operator G, which is the only head operator of the second component. When C receives the FCM, it applies the new configuration and starts propagating an epoch marker to operators D and E. These operators then forward the marker to operator F. When F receives the marker from both D and E, it applies the new configuration and stops the marker propagation. When operator G receives the marker, it applies the new

Algorithm 1 The Fries Scheduler (for dataflows with one-to-one operators only)

Input: $G = (V, E)$
Input: $\mathcal{R} = \{(o_1, U_1), \ldots, (o_n, U_n)\}$
1: $M \leftarrow \{o_1, \ldots, o_n\}$
2: $G' \leftarrow findMCS(G, M)$
3: $C_1, \ldots, C_p \leftarrow findComponents(G')$
4: **for** each $C \leftarrow C_1, \ldots, C_p$ **do**
5: send an FCM to the each head operator in C
6: start propagating an epoch marker within C

configuration and does not send out an epoch marker. We can show that the Fries scheduler can always produce a conflict-serializable schedule. Due to space limitations, we refer interested readers to the extended version [33] for the full proof.

The reconfiguration delay of the Fries scheduler is decided by the size of each MCS component, which is the number of edges in the component. Compared to the EBR scheduler, the FCMs sent to the head of each MCS component are not blocked by the processing of data by the upstream operators. Within each MCS component, the Fries scheduler still relies on epoch markers. In the extreme case where the MCS covers the entire dataflow graph, the Fries scheduler essentially becomes the epoch-based scheduler, where the FCMs are sent to all source operators and the epoch markers need to be propagated through the entire dataflow.

6 DATAFLOWS WITH ONE-TO-MANY OPERATORS

In this section we consider dataflows with one-to-many operators.

6.1 Challenges

Figure 8 shows a part of a dataflow with a one-to-many Join operator, which joins each input tuple with the Merchants table. When a tuple contains purchases from multiple merchants, Join generates multiple output tuples. For instance, the tuple t_1 joins with three merchants and produces the tuples t_2, t_3, and t_4. The Split operator splits the stream based on merchant information and sends different tuples to the two merchant fraud-detector operators FMX and FMY. The prediction results are combined by a Union operator.

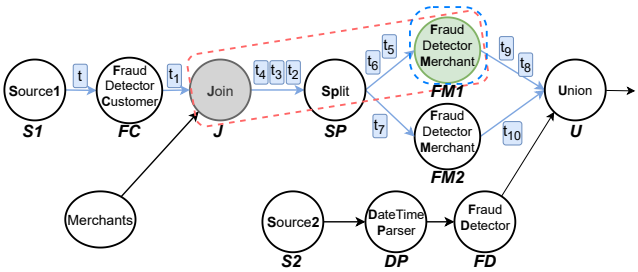

Figure 8: Reconfiguration of operator $FM1$ in a dataflow with a one-to-many Join operator. An incorrect MCS generated by Algorithm 1 is highlighted in blue. The correct MCS generated by Algorithm 2 is highlighted in red.

Based on Definition 4.4, t has the following data transaction T_5:

$$\Phi \; in \; T_5 : \{\phi(FC, t), \phi(J, t_1), \phi(SP, t_2), \phi(SP, t_3), \phi(SP, t_4), \phi(FMX, t_5), \phi(FMX, t_6), \phi(FMY, t_7), \phi(U_1, t_8), \phi(U_1, t_9), \phi(U_1, t_{10})\}.$$

We use an example to show that when reconfiguring a dataflow with one-to-many operators, a naive adoption of the Fries scheduler in Algorithm 1 can produce a non-conflict-serializable schedule. Consider a reconfiguration of operator FMX in Figure 8.Algorithm 1 adds the only reconfiguration operator FMX to the set M and computes the MCS with one component, which contains the operator FMX and no other edges. Algorithm 1 ignores the Join operator because it is not in the reconfiguration. The method sends an FCM to FMX. This operator does not propagate the FCM to its downstream operators because it is the only operator in the MCS component. Suppose the FCM sent to operator FMX arrives *after* the tuple t_5 and before the tuple t_6 in the same transaction. Then this scheduler produces the following schedule with a total order of the data operations and the function-update operations:

$$S_5 : [\phi(FC, t), \phi(J, t_1), \phi(SP, t_2), \phi(SP, t_3), \phi(SP, t_4), \boldsymbol{\phi(FMX, t_5)}, \boldsymbol{\mu(FD_1)}, \boldsymbol{\phi(FMX, t_6)}, \phi(FMY, t_7), \phi(U, t_8), \phi(U, t_9), \phi(U, t_{10})].$$

We can show that the schedule S_5 is not conflict-serializable. Intuitively, as indicated in the operations in bold, tuple t_5 is processed by FMX with the old configuration, and tuple t_6 in the same transaction is processed by FD_2 with the new configuration.

6.2 Extending the Fries scheduler

We extend the Fries scheduler Algorithm 1 to produce a conflict-serializable schedule for a dataflow with one-to-many operators and a function-update transaction. Intuitively, for a one-to-many operator, each of its descendant operators could receive multiple input tuples that belong to the same data transaction. In Figure 8, operator SP receives three tuples (t_2, t_3, and t_4), and operator FMX receives two tuples (t_5 and t_6) in the same data transaction.

Consider a reconfiguration that includes the operator FD_1. The function-update operation $\mu(FD_1)$ can be conflicting with the data operations of tuples t_5 and t_6 (in the same data transaction) in the same operator. To guarantee a conflict-serializable schedule, these two data operations must synchronize with $\mu(FMX)$ to ensure that both data operations are either before $\mu(FMX)$ or after $\mu(FMX)$. In other words, $\mu(FMX)$ cannot be scheduled in the middle of these two data operations. Notice that the Join operator is the earliest ancestor one-to-many operator of the reconfiguration operator FMX. If an FCM is sent to an operator O after the Join operator, since the operator O could possibly generate multiple data operations for the same data transaction, the FCM can be injected in the middle of these data operations, causing the schedule to be not conflict-serializable. Based on these observations, to guarantee the conflict-serializability, we can start the synchronization from the Join operator using an epoch marker. Recall that the Fries scheduler starts the epoch marker propagation from the head operators of a component in the MCS. The MCS is constructed using a set of operators M, which includes the reconfiguration operator FMX. To make sure the Join operator is treated as a head operator in a component, we add the operator to M before computing the MCS.

Algorithm 2 The Fries Scheduler (for general dataflows with one-to-many operators)

Input: A dataflow $G = (V, E)$
Input: A reconfiguration $\mathcal{R} = \{(o_1, U_1), \ldots, (o_n, U_n)\}$
1: $M = \{o_1, \ldots, o_n\}$
2: **for** each reconfiguration operator o_i in $\{o_1, \ldots, o_n\}$ **do**
3: $\quad\mathcal{A} \leftarrow$ set of ancestor one-to-many operators of o_i
4: $\quad\mathcal{E} \leftarrow computeEarliestAncestors(\mathcal{A})$
5: $\quad M \leftarrow M \cup \mathcal{E}$
6: ...same as Algorithm 1 line 2-6

Algorithm 2 shows the extended Fries scheduler, with the part in the box showing the differences compared to the original Fries scheduler in Algorithm 1. When constructing the MCS, apart from adding the operators in the reconfiguration to M (line 1), we also add to M all the earliest one-to-many ancestor operators of each reconfiguration operator o_i (lines 2 to 5). This step is done by first finding the set of ancestor one-to-many operators of o_i, denoted as $\mathcal{A}$, then finding the earliest ones in $\mathcal{A}$. Notice that a reconfiguration operator could have more than one earliest ancestor one-to-many operator. For example, in Figure 8, suppose the operators *FMX* and *FMY* are the only one-to-many operators in the dataflow. Then the reconfiguration operator U has both *FMX* and *FMY* as its earliest ancestor one-to-many operators according to the partial order of the DAG. We do the modification in the box because we want to start the synchronization from these one-to-many operators with the reconfiguration operators using epoch markers. The remaining steps are the same as in Algorithm 1.

As an example, in Figure 8, the only one-to-many operator is the Join operator J. Because the reconfiguration operator *FMX*'s earliest ancestor one-to-many operator is J, we add J to M when constructing the MCS. The resulting MCS includes a single component with operators J, *SP*, and *FMX*, together with their edges. The controller injects an FCM to operator J, which propagates an epoch marker within the component to operator *FMX*. We can show that the extended Fries scheduler still guarantees conflict-serializability of its produced schedule. The full proof can be found in [33].

6.3 Reducing delay by MCS pruning

For dataflows with one-to-many operators, the reconfiguration delay can be long when there are many intermediate operators between the head of an MCS component and a reconfiguration operator in the component. To address this limitation, we improve the Fries scheduler in Algorithm 2 by using pruning rules to remove one-to-many operators that do not need to be synchronized. Algorithm 3 shows the addition of a pruning step. In line 4, we call a function pruneAncestors that applies pruning rules to each of the ancestor one-to-many operators to decide it can be pruned.

Next, we introduce two pruning rules that are used in the improved Fries scheduler.

1. Edge-wise one-to-one pruning rule. Figure 9 (I) shows a part of a dataflow with a Replicate operator, denoted as *RE*. This operator replicates each input tuple to produce two output tuples and sends each of them to operators *C* and *D*. *RE* is a one-to-many operator by Definition 5.2. Suppose all other operators in this dataflow are one-to-one operators. Using Algorithm 2, the Fries scheduler includes

Algorithm 3 The Fries Scheduler with a Pruning Process

1: $M = \{o_1, \ldots, o_n\}$
2: **for** each reconfiguration operator o_i in $\{o_1, \ldots, o_n\}$ **do**
3: $\quad\mathcal{A} \leftarrow$ set of ancestor one-to-many operators of o_i
4: $\quad$***pruneAncestors*($\mathcal{A}$)**
5: $\quad\mathcal{E} \leftarrow computeEarliestAncestors(\mathcal{A})$
6: $\quad M \leftarrow M \cup \mathcal{E}$
7: ...same as Algorithm 1 line 2-6

operators *RE*, *C*, and *E* in the MCS, as shown in the red box in Figure 9 (I). This is because *RE* is the earliest one-to-many ancestor operator of the reconfiguration operator *E*.

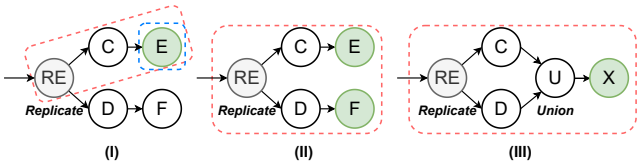

Figure 9: Example reconfigurations on dataflows with a replicate operator. (I): The MCS can be pruned. (II) and (III): the MCS's cannot be pruned.

Although operator *RE* is a one-to-many operator, for an input tuple, the operator outputs a single tuple on each edge. For the reconfiguration operator *E*, it only receives a single tuple in each data transaction. Therefore, there is no need for operator *E* to synchronize with operator *RE*. The MCS only contains operator *E*, as shown in the blue box in Figure 9. Figure 9 (II) and (III) show dataflows where the MCS with a replicate operator cannot be pruned. In Figure 9 (II), for each tuple processed by operator *E*, the corresponding replicated tuple must be processed by the same version of operator *F*. We can achieve the goal by starting the synchronization from *RE*. In Figure 9 (III), operator *X* receives all the replicated tuples in each data transaction. Therefore we also need to start the synchronization from the one-to-many operator *RE*.

Next, we formally describe the pruning rule. We prune an ancestor one-to-many operator A of a reconfiguration operator o_i if the following conditions are true. (1) On each of its output edges, A emits at most one tuple for each input tuple. (2) The A has only one output edge e connected to a downstream reconfiguration operator, and this output edge e is connected to o_i. Intuitively, condition (1) ensures that A behaves like a one-to-one operator on each of its output edges. Condition (2) ensures that the reconfiguration transaction of o_i affects only one output tuple of A sent on edge e. As analyzed in Section 6.2, a one-to-many operator O needs to be included in the MCS to ensure multiple output tuples of O are processed using the same configuration. In this case, only a single output tuple of A is affected by the reconfiguration. Therefore, A can be pruned from the set of operators used to construct the MCS.

2. Uniqueness pruning rule.

Next, we show another example of pruning an one-to-many operator. In Figure 10, operator *E* is reconfigured. Each input tuple is first replicated by operator *RE*. The replicated tuples are sent to operators *C* and *D*. They are then combined to a single tuple using

a Self-Join operator SJ on the primary key. Algorithm 2 computes the sub-DAG from operator RE to operator E as the MCS, as shown in the red box in Figure 10. However, operator SJ ensures that it generates at most one output tuple for input tuple from the source. Therefore, RE does not need to be synchronized and the MCS can only contain E without RE, as shown in the blue box. In general, we prune an ancestor one-to-many operator A of a reconfiguration operator o_i if on each path from A to o_i, there exists an operator O that has the following uniqueness property: operator O generates at most one output tuple for each data transaction. In the running example, SJ is such an O operator and RE can be pruned.

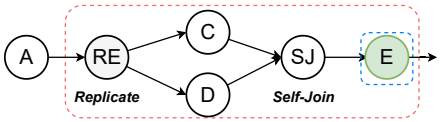

Figure 10: Operator RE can be pruned from the set of operators used to construct the MCS.

7 EXTENSIONS

In this section, we consider how the Fries scheduler handles dataflows with blocking operators and parallel dataflows with multiple workers per operator. Note that the Fries scheduler also affects checkpoint-based fault-tolerance in a parallel setting, due to limited space, we discuss how to support fault-tolerance in the extended version [33].

7.1 Dataflows with Blocking Operators

We now consider how the Fries scheduler works on dataflows containing blocking operators, such as aggregation and sort. Consider a blocking operator B. All operators before B need to run to their completion before the operators after B start to run. In other words, the operators before B and those after B never execute at the same time. Based on this observation, we can use the blocking operators in a dataflow to divide the dataflow into multiple sub-dataflows, with each of them containing pipelined operators only. Then we run Fries on each sub-dataflow during its execution.

7.2 Multiple Workers for an Operator

In a parallel execution engine, each operator can have multiple workers, with each worker processing a data partition. We map a single-worker dataflow $G = (V, E)$ to a parallel dataflow $G^* = (V^*, E^*)$, where each operator v in V is mapped to multiple parallel workers $v^1, \ldots, v^p$ in V^*, where p is the number of workers of the operator. We map a reconfiguration $\mathcal{R}$ specified on the single worker dataflow G to a new reconfiguration $\mathcal{R}^*$ on the parallel dataflow G^* of G. Figure 11 shows part of a parallel dataflow based on Figure 8, with two workers per operator. For each function update $\mu(o_i)$ on an operator o_i in R, we map it to a set of function updates on all the workers of o_i, i.e., $\{(o_i^1, \mu(o_i)), \ldots, (o_i^p, \mu(o_i))\}$ in R^*. For example, the reconfiguration on operator FMX is mapped to a reconfiguration on the corresponding workers FMX_1 and FMX_2.

Notice that the parallel dataflow G^* is also a DAG. Algorithm 3 can be directly run on G^* with $\mathcal{R}^*$. The generated MCS is highlighted in red in Figure 8. The Fries scheduler treats a worker of an operator to have the same property (one-to-one or one-to-many) as the operator in hash and range partitioning. For example, both workers of the Join operator are treated as one-to-many operators. When using the broadcast strategy, a worker replicates an output tuple to all its downstream workers. In this case, the Fries treats it as if a Replicate operator is added after the worker. The pruning techniques described in Section 6.3 can still be used.

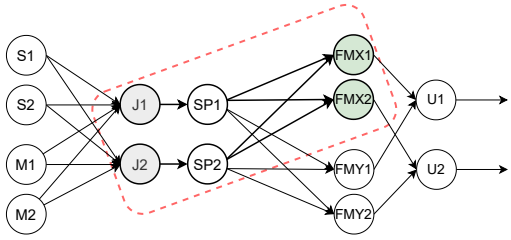

Figure 11: A reconfiguration on a parallel dataflow.

8 EXPERIMENTS

In this section, we present the results of experiments of different reconfiguration schedulers and show the benefits of Fries.

8.1 Setting

Datasets. We used three datasets shown in Table 2. Dataset 1 had 24M tuples of credit card payments with 12 attributes [27], such as the customer, merchant, date, amount, and chip usage. Dataset 2 was constructed by grouping the credit card payments per user in dataset 1. Each record had a user and a list of payments by the user. We used this dataset to utilize a one-to-many unnest operator to split a payment list into multiple records. Dataset 2 was generated using the TPC-DS benchmark [31] with a scale factor of 100.

Table 2: Datasets used in the experiments.

Dataset	Table	Attribute #	Tuple #
1	Credit card payment	12	24M
2	Credit card payment aggregated per user	2	20K
3	Catalog sales	34	144M
	Store sales	23	288M
	Web sales	34	71M

Workflows. We constructed workflows as shown in Figure 12. Workflow W_1 simulated a fraud detection application. A user-based inference operator saved 10 recent payments per use as state and used an LSTM auto-encoder [35] to predict the probability of fraud. On top of W_1, workflow W_4 and W_5 included an additional merchant-based inference operator. It saved 50 recent payments per merchant as state and used a similar LSTM auto-encoder to do inference. Workflow W_2 was constructed based on TPC-DS query 40. Workflow W_3 was constructed based on TPC-DS query 71. All the join operators in these workflows were one-to-one operators because they join a primary key with a foreign key. We only considered the pipelined sub-DAG of each dataflow. In Figure 12, we highlighted all the pipelined edges considered in the experiments in red.

Reconfigurations. For workflow W_1, we performed configurations with one operator. For the other workflows, we performed

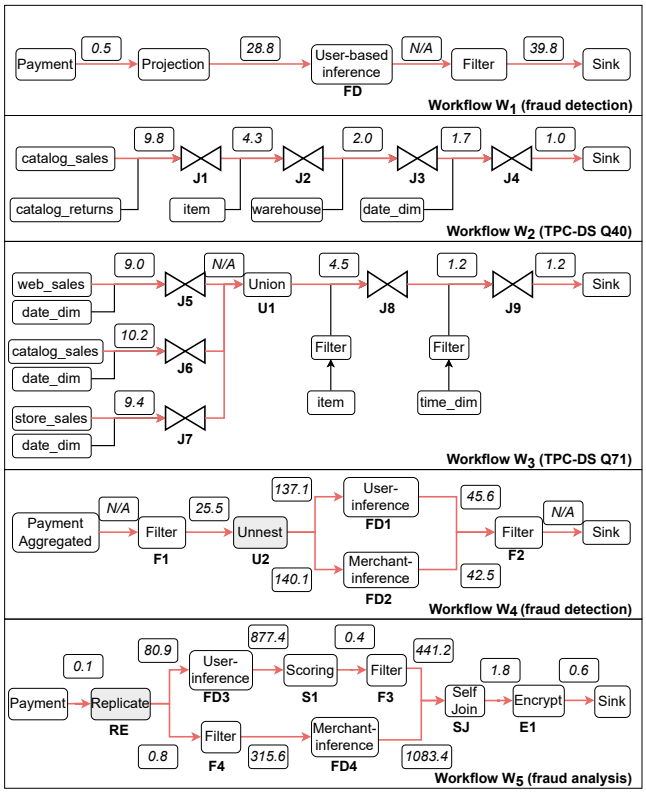

Figure 12: Workflows used in the experiments. Pipelined edges are highlighted in red.

reconfigurations with multiple operators. The methods of choosing reconfiguration operators will be described in each experiment.

Schedulers. For the epoch-based scheduler, we implemented the EBR scheduler of Chi [22] (described in Section 3). As Chi was not open source, we implemented this scheduler on top of Flink, and used Flink's aligned checkpoint barriers as epoch markers. For fair-comparison purposes, we implemented Fries also on top of Flink. In the implementation, FCM's sent from the controller to a specific worker of an operator were implemented using Flink's RPC messages. The epoch markers propagated within an MCS component were implemented on top of Flink's checkpoint barriers.

System environment. All the experiments were conducted on the Google Cloud Platform (GCP). The execution was on a GCP dataproc cluster with 1 coordinator machine and 10 worker machines. All the machines were of type n1-highmem-4 with Ubuntu 18.04. The job controller of Flink ran on the coordinator. The coordinator machine had a 2TB HDD, while each worker machine had a 250GB HDD. To separate computation and storage, we stored the datasets in an HDFS file system on another cluster with 6 e2-highmem-4 machines, each with 4 vCPU's, 32 GB memory, and a 500GB HDD. For all the schedulers, we used Flink release 1.13 and Java 8.

8.2 Choke Point Analysis of Workflows

There were various choke points in the workflows where the reconfiguration delay between two operators was very high. We analyzed these choke points in the experiment workflows by computing the average reconfiguration delay between two operators using the EBR scheduler and showed the numbers on top of each edge in Figure 12. The numbers represented the delay from the time when the upstream operator applied the reconfiguration and sent out epoch markers, to the time when the downstream operator aligned all the epoch markers and applied the reconfiguration. Some edges are marked as N/A because the two connected operators were fused to a single operator chain. The other edges perform re-partition operations, thus the two connected operators are not chained.

We had the following observations. 1) Expensive operators usually created choke points in the workflow. For example, in $W4$, both expensive inference operators had a high delay of around 140 seconds after $U1$. 2) Stragglers also created choke points. For example, in $W5$, there was a delay of 877.4s between $FD3$ and $S1$ because one of the $FD3$ workers was a straggler. Recall that due to the epoch alignment step, $S1$ had to receive all the checkpoint barriers before applying the reconfiguration. $S1$ was blocked when waiting for the straggler $FD3$ worker to finish. 3) Choke points depended on the amount of data in each operator's input data channel. For example, in both $W2$ and $W3$, the initial joins had larger delays compared to other joins. Since tuples were filtered by every join, the joins near the sink received less data and they had a lower delay.

8.3 Benefits of Short Reconfiguration Delay: Reducing End-to-end Tuple Latency

A main advantage of Fries was its short delay compared to epoch-based schedulers. To show the benefits of this advantage, we considered a scenario for W_1 as shown in Figure 13, where the developer needed to hot-replace the model in the user-based inference operator FD during the execution to deal with a sudden surge of input data. In W_1, we set the number of workers for operators (except for the source and sink) to 40. The maximum throughput of FD was around $1,600$ tuple/s. The source operator started with an initial ingestion rate of $1,000$ tuples/s. At $t = 100s$, we increased the ingestion rate to $2,000$ tuples/s. At $t = 120s$, we replaced the original LSTM model in FD with a cheaper LSTM model with fewer parameters to speed up the processing. At $t = 200s$, we further increased the rate to $9,000$ tuples/s. At $t = 220s$, we performed another reconfiguration to deploy a simple decision-tree model.

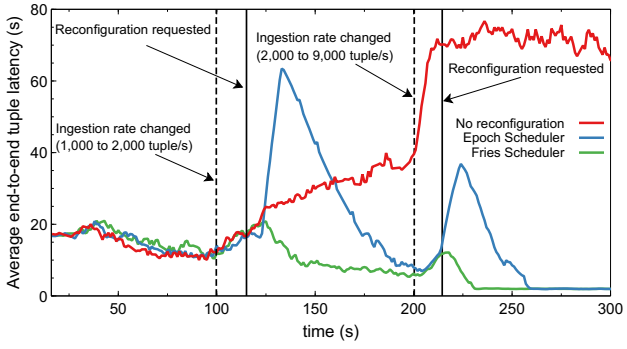

Figure 13: Effect of mitigating surges of data-ingestion rate by different schedulers (W_1 on dataset 1).

Figure 13 shows the average end-to-end latency of output tuples for every 10-second sliding window. (1) Without reconfiguration, the latency began to increase after 100 seconds because *FD* was not fast enough to process the incoming tuples. The latency increased continuously until the backpressure mechanism slows down the data ingestion rate. (2) Using the Epoch scheduler, the latency rapidly increased to above 60 seconds until around $t = 135s$ due to the surge. The main reason for the increase was the blocking in the epoch alignment step. Before the sink operator worker can process any tuple in the new epoch, it needed to wait until all 40 upstream *FD* workers completely processed all tuples in the old epoch. Note that the delay was determined by the slowest *FD* worker. There were two straggler workers that took 58 seconds and 69 seconds to finish the old epoch, respectively. The two straggler workers suffered from data skew. On average, each worker processed 35,000 tuples in the old epoch. However, the slowest worker processed 62,000 tuples. (3) Using the Fries scheduler, the latency immediately decreased after $t = 120s$, indicating that *FD* applied the reconfiguration and quickly processed the buffered tuples. Compared to Epoch, Fries required less time to mitigate the surge. In this reconfiguration, the MCS component contained operator *FD* only. Therefore, FCMs are directly sent to all *FD* workers and no epoch markers were propagated. This eliminated the aforementioned delay caused by the epoch alignment step.

8.4 Effect of Data Ingestion Rates on Reconfiguration Delays

Next we evaluated the effect of different factors on the delay. We first considered data-ingestion rate. For workflow W_1, we gradually increased the ingestion rate from 500 tuples/s to 2,500 tuples/s. After the execution of 120 seconds, we applied a dummy reconfiguration on *FD* and measured the delay. As shown in Figure 14 (with a log scale for the y-axis), when the ingestion rate increased, the delay of the Epoch scheduler also increased due to the larger amount of in-flight tuples. Since the Fries scheduler sent FCM's directly to *FD*, its delay grew slower than the Epoch scheduler.

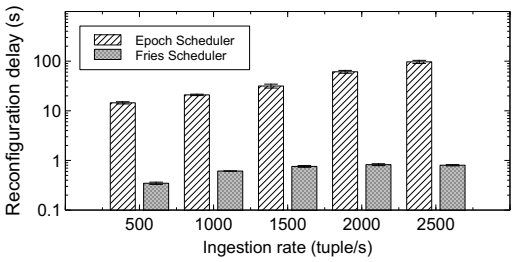

Figure 14: Effect of ingestion rate on delay (W_1 on dataset 1).

8.5 Effect of Operator Costs on Delays

To evaluate the effect of operator cost on the reconfiguration delay, for workflow W_1, we gradually increased the cost of the user-based inference operator *FD* to process each input tuple. The *FD* operator maintained a bounded queue of recent payment amounts of each user. When an input tuple was received by *FD*, the operator passed the payment amounts in the queue to its ML model. In different runs of experiments, we gradually increased the size of this queue from 10 to 50 so that the operator took more time to process each input tuple. Again, for each configuration, after the execution ran for 120 seconds, we applied a dummy reconfiguration on *FD* and measured the delay under the two schedulers. As shown in Figure 15, when the *FD*'s cost increased, the delay of the Epoch scheduler also increased because each in-flight data tuple prior to the epoch marker took more time to be processed. On the other hand, the delay of the Fries scheduler grew much slower than the Epoch scheduler.

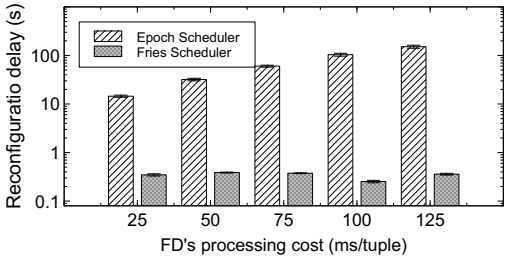

Figure 15: Effect of operator cost on delay (W_1 on dataset 1).

8.6 Effect of Reconfigurations on Delays

We wanted to evaluate the effect of reconfigurations on the delay under the two schedulers. We varied the number of reconfiguration operators in both workflows W_2 and W_3. For both workflows, we used 40 workers for each operator. For every 10 seconds, we requested a reconfiguration and measured the average reconfiguration delay. The results are shown in Table 3. For each reconfiguration, we show its operators, the MCS components generated by the Fries scheduler, the length of a longest path of each component, the delay of using the Fries scheduler, and the delay of using the Epoch scheduler. We reported the path length because it affected the delay in the Fries scheduler.

We have the following observations from the results. (1) The delay of the Fries scheduler was always significantly lower than the delay of the Epoch scheduler. For example, for the W_2 configuration on $J1$ and $J4$, the delay of the Fries scheduler was 1,702ms, compared to 12,361ms of the Epoch scheduler. (2) The delay of Fries was very low if each MCS component only had one operator. For example, for the W_3 reconfiguration on $J5$ and $J6$, the Fries scheduler had a delay of only 127ms. This low delay was because the Fries scheduler sent FCM's separately to both operators and their reconfiguration happened in parallel. (3) When the length of the longest path in a component increased, the delay also increased. For example, for the reconfiguration of $J1$ and $J3$, the longest path in their MCS had a length of 2, and the delay was 1,664ms. For the reconfiguration of $J1$ and $J4$, the longest path in their MCS had a length of 3, the delay increased to 1,702ms.

8.7 Reconfiguration Delays in Workflows with One-to-many Operators

We used workflow W_4 to evaluate the effect of different reconfiguration operators on the reconfiguration delay in workflows with

Table 3: Reconfiguration operators, corresponding MCS, and reconfiguration delay in workflows W_2 and W_3 on dataset 3. Head operators in each component are highlighted in bold.

	Reconfiguration operators	MCS components	Longest path length	Fries Scheduler delay (ms)	Epoch Scheduler delay (ms)
W_2	J1	{**J1**}	0	46	11,432
	J2	{**J2**}	0	44	11,709
	J1, J3	{**J1**, J2, J3}	2	1,664	12,339
	J1, J4	{**J1**, J2, J3, J4}	3	1,702	12,361
	J3, J4	{**J3**, J4}	1	387	13,767
W_3	J5	{**J5**}	0	87	4,127
	J5, J6	{**J5**} {**J6**}	0 0	127	8,352
	J5, J6, J7, J8	{**J5**, **J6**, **J7**, U1, J8}	3	447	19,608
	J5, J6, J7, J9	{**J5**, **J6**, **J7**, U1, J8, J9}	4	526	19,717
	J7, J8, J9	{**J7**, U1, J8, J9}	3	1,340	20,532

a one-to-many operator $U2$. This operator split all the payments of a user and sent them to both $FD1$ and $FD2$. Table 4 shows the results. We have the following observations. (1) The delay of the Fries scheduler was still always lower than the Epoch scheduler. (2) The reconfiguration of $FD1$ took a long time (47,892ms) in Fries because $FD1$ was not the head operator of its component. The epoch markers had to go through the data channels of $FD1$ (from multiple workers). Since $FD1$ processed tuples slowly, many of its input tuples were buffered in its data channels, which delayed the propagation of the epoch markers. (3) The reconfiguration of $F2$ took a long delay (221, 353ms) in Fries because its generated MCS contained every operator on the path from $U2$ and $F2$ with the one-to-many $U2$ operator and both $FD1$ and $FD2$ were slow.

Table 4: Reconfiguration operators, corresponding MCS, and reconfiguration delay in W_4 on dataset 2. Head operators in each component are highlighted in bold.

Reconfiguration operators	MCS components	Longest path length	Fries Scheduler delay (ms)	Epoch Scheduler delay (ms)
F1, U2	{**F1**, U2}	1	69	151
FD1	{**U2**, FD1}	1	47,892	131,103
F2	{**U2**, FD1, FD2, F2}	5	221,353	236,153

8.8 Effect of MCS Pruning on Delays in Workflows with One-to-many Operators

We used workflow W_5 to evaluate the effect of the MCS pruning method proposed in Section 6.3 on the reconfiguration delay in workflows with a one-to-many Replicate operator and a Self Join operator. For each reconfiguration, we compared the Fries scheduler with the pruning step turned on and turned off. Table 5 shows the results. We have the following observations. (1) In general, when pruning is possible, the size of MCS components was reduced and the delay with pruning was significantly lower than the delay without pruning. For example, the reconfiguration of operator $FD4$ and the reconfiguration of operator $F3$ benefited from the edge-wise one-to-one pruning rule. (2) In the case of reconfiguring both $FD3$ and $FD4$, the pruning rules could not prune the one-to-many Replicate operator. Therefore the delays were similar. (3) The reconfiguration of operator $E1$ benefited from the uniqueness pruning rule. This reconfiguration had the largest benefit in delay because the number of edges in the MCS reduced from eight to zero, which greatly reduced the epoch maker synchronization time.

Table 5: The effect of MCS pruning on delays in W_5.

Reconfiguration operators	MCS with pruning	MCS without pruning	Fries with pruning delay (ms)	Fries without pruning delay (ms)
FD4	{**FD4**}	{**RE**, F4, FD4}	158	450,149
F3	{**F3**}	{**RE**, FD3, S1, F3}	94	383,781
F4	{**F4**}	{**RE**, F4}	10	446
FD3, FD4	{**RE**, FD3, F4, FD4}	{**RE**, FD3, F4, FD4}	661,892	663,460
E1	{**E1**}	{**RE**, FD3, S1, F3, F4, FD4, SJ, E1}	85	1,122,686

8.9 Effect of Multiple Workers on Delays

To evaluate the effect of the worker number per operator on the reconfiguration delay, we considered workflow W_2 and increased the worker number per operator from 1 to 40. After the workflow ran for 20 seconds, we requested a dummy reconfiguration of $J1$ and $J4$. We measured the reconfiguration delay of the two schedulers.

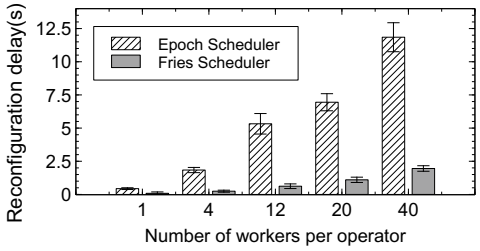

Figure 16: Effect of parallelism on delay (W_2 on dataset 3).

As shown in Figure 16, as the worker number increased, the delay increased for both schedulers. This was because between each pair of join operators, the data was shuffled and every join worker needed to receive an epoch marker from all its upstream workers. When each worker number increased, the number of epoch markers to collect also increased. The delay of Fries scheduler was again lower than the Epoch scheduler because the Fries scheduler propagated epoch markers only through the data channels between MCS workers. The number of channels between MCS workers was always less than the number of channels between all workers.

9 CONCLUSIONS

We studied the problem of runtime configurations in dataflow systems with a low delay. We showed limitations of existing epoch-based reconfiguration schedulers on the delay. We developed a new technique called Fries that uses fast control messages to do reconfigurations. We formally defined consistency in runtime reconfigurations, and developed a Fries scheduler with consistency guarantee. Our extensive experimental evaluation showed the advantages of this technique compared to epoch-based schedulers.

ACKNOWLEDGMENTS

This work was supported by the NSF IIS-2107150 award. We thank Sadeem Alsudais and Yicong Huang for participating in discussions.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[2] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[3] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[4] Philip A. Bernstein and Eric Newcomer. 1996. *Principles of Transaction Processing for Systems Professionals.* Morgan Kaufmann.

[5] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. 2012. Transactional stream processing. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, Elke A. Rundensteiner, Volker Markl, Ioana Manolescu, Sihem Amer-Yahia, Felix Naumann, and Ismail Ari (Eds.). ACM, 204–215. https://doi.org/10.1145/2247596.2247622

[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[7] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2651–2658. https://doi.org/10.1145/3318464.3383131

[8] 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, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[9] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (2014), 401–412. https://doi.org/10.14778/2735496.2735503

[10] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. 2008. *Algorithms.* McGraw-Hill.

[11] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 725–736. https://doi.org/10.1145/2463676.2465282

[12] FlinkFraudDetectionDemo [n.d.]. Advanced Flink Application Patterns Vol.1: Case Study of a Fraud Detection System, https://flink.apache.org/news/2020/01/15/demo-fraud-detection.html.

[13] FlinkSavepoint [n.d.]. Savepoints in Apache Flink, https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/savepoints/.

[14] FlinkUpdateCepPattern [n.d.]. Support dynamically changing CEP patterns in Flink, https://issues.apache.org/jira/browse/FLINK-7129.

[15] FlinkUpdateVol2 [n.d.]. Advanced Flink Application Patterns Vol.2: Dynamic Updates of Application Logic, https://flink.apache.org/news/2020/03/24/demo-fraud-detection-2.html.

[16] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Tappan Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[17] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *Proc. VLDB Endow.* 12, 9 (2019), 1002–1015. https://doi.org/10.14778/3329772.3329777

[18] Konstantinos Kakousis, Nearchos Paspallis, and George Angelos Papadopoulos. 2010. A survey of software adaptation in mobile and ubiquitous computing. *Enterp. Inf. Syst.* 4, 4 (2010), 355–389. https://doi.org/10.1080/17517575.2010.509814

[19] Fabio Kon and Roy H. Campbell. 1999. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems, May 3-7, 1999, The Town & Country Resort Hotel, San Diego, California, USA*, Murthy V. Devarakonda (Ed.). USENIX, 175–188. http://www.usenix.org/publications/library/proceedings/coots99/kon.html

[20] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *Proc. VLDB Endow.* 13, 5 (2020), 740–753. https://doi.org/10.14778/3377369.3377381

[21] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 245–255. https://doi.org/10.1145/2025113.2025148

[22] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proc. VLDB Endow.* 11, 10 (2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[23] Yancan Mao, Yuan Huang, Runxin Tian, Xin Wang, and Richard T. B. Ma. 2021. Trisk: Task-Centric Data Stream Reconfiguration. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 214–228. https://doi.org/10.1145/3472883.3487010

[24] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (2015), 2134–2145. https://doi.org/10.14778/2831360.2831367

[25] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2471–2486. https://doi.org/10.1145/3318464.3389723

[26] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738

[27] Inkit Padhi, Yair Schiff, Igor Melnyk, Mattia Rigotti, Youssef Mroueh, Pierre Dognin, Jerret Ross, Ravi Nair, and Erik Altman. 2021. Tabular transformers for modeling multivariate time series. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 3565–3569. https://ieeexplore.ieee.org/document/9414142

[28] Alireza Sadeghi, Naeem Esfahani, and Sam Malek. 2017. Ensuring the Consistency of Adaptation through Inter- and Intra-Component Dependency Analysis. *ACM Trans. Softw. Eng. Methodol.* 26, 1 (2017), 2:1–2:27. https://doi.org/10.1145/3063385

[29] StreamINGFraudDetection [n.d.]. StreamING Machine Learning Models: How ING Adds Fraud Detection Models at Runtime with Apache Flink, https://www.ververica.com/blog/real-time-fraud-detection-ing-bank-apache-flink.

[30] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. 2019. Reconfigurable Streaming for the Mobile Edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, HotMobile 2019, Santa Cruz, CA, USA, February 27-28, 2019*, Alec Wolman and Lin Zhong (Eds.). ACM, 153–158. https://doi.org/10.1145/3301293.3302355

[31] TPC-DS [n.d.]. TPC-DS http://www.tpc.org/tpcds/.

[32] UpgradeFlinkApplications [n.d.]. Upgrading Applications and Flink Versions, https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/ops/upgrading/.

[33] Zuozhi Wang, Shengquan Ni, Avinash Kumar, and Chen Li. 2022. Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees (Extended Version). arXiv:2210.10306 [cs.DB]

[34] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann.

[35] Bénard Wiese and Christian Omlin. 2009. Credit card transactions, fraud detection, and machine learning: Modelling time with LSTM recurrent neural networks. In *Innovations in neural information paradigms and applications*. Springer, 231–268.

[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 423–438. https://doi.org/10.1145/2517349.2522737