# Reactive Dataflow for Inflight Error Handling in ML Workflows

Abhilash Jindal
IIT Delhi, India
ajindal@iitd.ac.in

Kaustubh Beedkar
IIT Delhi, India
kbeedkar@iitd.ac.in

Vishal Singh
IIT Delhi, India
tmibvishal@gmail.com

J. Nausheen Mohammed
IIT Delhi, India
mdjnausheen786@gmail.com

Tushar Singla
IIT Delhi, India
tusharsingla078@gmail.com

Aman Gupta
IIT Delhi, India
amangupt01@gmail.com

Keerti Choudhary
IIT Delhi, India
keerti@iitd.ac.in

## ABSTRACT

Modern data analytics pipelines comprise traditional data transformation operations and pre-trained ML models deployed as user-defined functions (UDFs). Such pipelines, which we call ML workflows, generally produce erroneous results due to data errors inadvertently introduced by ML models. Model errors are one of the main obstacles to improved accuracy of ML workflows. In this paper, we present Popper, a dataflow system—for expressing ML workflows—that natively supports inflight error handling. Users can extend ML workflows expressed in Popper by plugging in error handlers to improve accuracy. We propose reactive dataflow, a novel cyclic graph-based dataflow model that provides convenient abstractions for interleaving dataflow operators with user-defined error handlers for detecting and correcting errors on the fly. We also propose an efficient execution strategy amenable to pipeline parallel execution of reactive dataflow. We discuss open research challenges for making error handling a first-class citizen in dataflow systems and present preliminary evaluation of our prototypical system, which shows the effectiveness and benefits of inflight error handling in ML workflows.

**ACM Reference Format:**
Abhilash Jindal, Kaustubh Beedkar, Vishal Singh, J. Nausheen Mohammed, Tushar Singla, Aman Gupta, and Keerti Choudhary. 2024. Reactive Dataflow for Inflight Error Handling in ML Workflows. In *Workshop on Data Management for End-to-End Machine Learning (DEEM 24), June 9, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3650203.3663333

## 1 INTRODUCTION

Recent integration of pre-trained ML models into traditional data processing pipelines has brought transformative changes, enabling diverse applications such as traffic planning, autonomous driving, text question answering, and automating tedious manual tasks.

At the core of modern data analytics applications lies data processing pipelines, which are composed of complex data transformations and ML models. We refer to these pipelines as *ML workflows*. ML models are often deployed in workflows as user-defined functions (UDFs) interleaving with other data transformation UDFs and relational operations such as projection, join, or aggregation. For example, Figure 1 shows an ML workflow for traffic data analysis. The workflow processes stored traffic videos to first extract vehicles' number plates and combine them with data from the vehicle registration database to count the number of vehicles for each fuel type (petrol, diesel, CNG, or electric) within a time window. Similar ML workflows are useful for improving traffic planning, debugging autonomous vehicle models, and automating tedious manual tasks.

Data processing frameworks such as Spark [54], Flink [12], Naiad [39] among others [1, 13, 29] have expanded their scope from handling traditional relational workloads to more complex and diverse workloads involving UDFs. These frameworks have allowed for streamlined development and efficient execution of ML workflows over large datasets.

Yet, developing effective ML workflows remains a challenge. The building blocks of these workflows, which are the ML models, can silently introduce errors. For example, the ML model for optical character recognition in Figure 1, to convert the number plate image to a machine-readable text may produce erroneous data [30]. ML models deployed in workflows are bound to make errors, which arise for diverse reasons, including changes in data distribution between training and deployment data, incomplete training data, or noisy inputs [47]. Model errors can significantly impact the end-to-end accuracy of ML workflows [33]. Inflight error handling is therefore crucial for effective ML workflow development.

In this paper, we discuss three open research challenges from a dataflow system perspective to make inflight error handling a first-class citizen: abstractions for interleaving user-defined error handlers with dataflow operators; cyclic-graph based execution model for inflight modifications; and intuitive APIs for workflow developers for specifying ML workflows with error handlers.

To address the above challenges, we present Popper, a prototypical dataflow system for developing ML workflows. We introduce reactive dataflow, a new programming model, which extends the existing dataflow model with user-defined error handlers as first-class citizens and allows expressing incremental changes to inflight data by downstream error handlers, i.e., error handlers can detect errors in the dataflow and correct the output of upstream operators. We also propose an execution model based on directed cyclic graphs that allows for efficient execution of reactive dataflow. The key aspect of our execution model is categorization of graph edges based on certain transformation properties of dataflow operations. This categorization enables efficient propagation of corrections applied to upstream operators. We also propose novel techniques that enable pipeline parallel execution of reactive dataflow and present Popper's staged execution and architecture.


*DEEM 24, June 9, 2024, Santiago, AA, Chile*

ACM ISBN 979-8-4007-0611-0/24/06
https://doi.org/10.1145/3650203.3663333

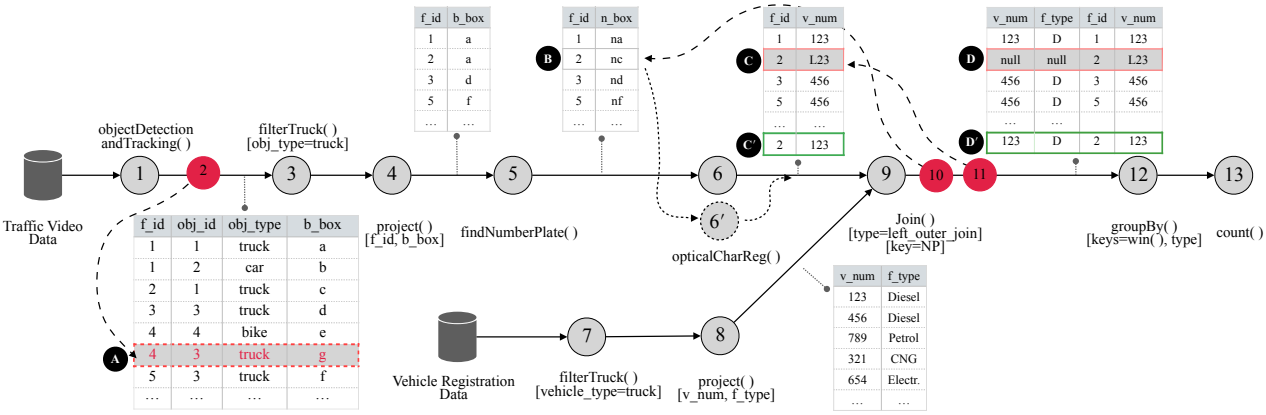



**Figure 1: An example traffic analysis ML workflow that counts the number of trucks for each fuel type within a time window. ML models like object detection, finding number plate, and optical character recognition can introduce mistakes. Error handlers 2, 10, and 11 catch mistakes and apply corrections in upstream operators' outputs.**

Our preliminary experimental study using real-world ML workflows shows that inflight error handling improves F1 scores by up to 0.44 which translates into up to 39.4% reduction in human-in-the-loop cost, measured by the number of post-facto manual corrections needed by the workflows. The results also show that Popper can execute dataflow programs efficiently.

## 2 INFLIGHT ERROR HANDLING

We start with a motivating scenario to illustrate the need for inflight error handling in ML workflows. We then discuss the status quo for error handling in dataflow systems before outlining the research challenges.

### 2.1 Motivation

Consider a traffic analysis application that runs a Machine Learning (ML) workflow[1], which processes traffic videos to extract vehicles' number plates and combines it with data from the vehicle registration database to count the number of trucks for each fuel type (petrol, diesel, CNG, and electric) within a time window. More concretely, Figure 1 illustrates the ML workflow for the above use case. Ignore the red circles, shaded boxes, and dashed arrows for now.

In our example workflow, first, each video frame `f_id` is processed using an object detection and tracking model `odt()` to determine the type `obj_type` of object (e.g., car, truck, bike, etc.), an identifier `obj_id` for objects, and their bounding boxes `b_box`[2]. Then, the frames containing trucks are processed using ML models: `findNP()` to determine the bounding box of the number plate `n_box` and an optical character recognition model `ocr()` to determine the vehicle's number plate `v_num`. Lastly, this data is joined with data from the vehicle registration database that stores the fuel type `f_type` information of each vehicle before aggregating the data for each vehicle type and time window. Figure 1 shows the data processing operations for the above workflow, where we also show excerpts of some intermediate data (tuples) between different workflow operations.

[1]In this paper, we consider ML workflows that comprise ML inference tasks.
[2]We note that bounding boxes are typically represented by four values (coordinates); for brevity we represent them by one value in Figure 1.

ML workflows are typically expressed as dataflows (as shown in Figure 1), where the workflow is modeled as a directed graph of data flowing between operations. Dataflow systems such as Spark [54], Flink [12], or Naiad [39] among others [29] are a natural choice to express ML workflows as they offer convenient abstractions with well-defined operator semantics and support for user-defined functions (UDFs). For instance, one could easily express the workflow in Figure 1 using user-defined `Map`, `Filter`, `Join`, and `ReduceByKey` operations that dataflow systems' API provides.

Developing effective ML workflows is challenging as the building blocks of these workflows, the ML models, often produce erroneous outputs. Errors may arise for diverse reasons, including changes in data distribution between training and deployment data, incomplete training data, or noisy inputs. For example, in the workflow shown in Figure 1, operations such as `odt()`, `findNP()`, or `ocr()` may produce erroneous outputs. For instance, `odt()` may fail to detect a truck, i.e., when the model detected the same truck (`obj_id=3`) in frames 3 and 5, but not in frame 4. We show this missing detection (row labeled Ⓐ) by a shaded box in the output of `odt()`. Likewise, the ML model used in `findNP()` operation may incorrectly identify the bounding box of the vehicle's number plate (label Ⓑ), or the optical character recognition model in `ocr()` may output an incorrect vehicle number (label Ⓒ), for example, "L23" instead of "123".

Model errors can significantly impact the end-to-end accuracy of ML workflows. In Figure 1, for example, a missing object detection or incorrectly identifying a vehicle's number plate may lead to inaccurate aggregates *i.e.,* the workflow will incorrectly compute the number of diesel trucks for the window comprising frames 1–5. Therefore, handling data errors in ML workflows is crucial.

### 2.2 Status quo

Existing dataflow systems lack native support for inflight error handling. We note that effective error handling requires both error detection and error correction. Recent work [33] has shown that ML model errors can be detected using so-called *model assertions*, i.e., by having "special" operations that check for the correctness of a model's output. For example in the workflow of Figure 1, one could implement a stateful `Map` operator to detect objects missed

by `odt()` in consecutive frames, before outputting the detection. However, such an approach has limitations in that it cannot be used for handling errors that may only be evident later, i.e., in the output of some downstream operation. In our ongoing example, for instance, an error in the output of `findNP()` or `ocr()` is only evident after an error is detected in the output of the `Join` operation. For example, the workflow developer can check for a null value in left `v_num` to detect an error in the output of either or both previous operations 4 and 5—as every vehicle must be registered.

Detecting and debugging data errors (e.g., null values) in a dataflow can also be achieved using data debugging techniques proposed in [16, 23, 24, 28, 35]. For example, using the backward tracing technique employed in data debugging, one can debug that the null values in a certain output row of operator 9 results from the output rows B and/or C. However, correcting the erroneous upstream rows remains a challenge. The workflow developer, for instance, may want to use a different OCR engine only for erroneous tuples (e.g., row B) without impacting the other inflight data.

Overall, current approach in dataflow systems to handle errors is to "drop" the erroneous tuples. While this may improve the precision of the workflow, it worsens the recall. Instead, a desirable approach is to be able to go "back in time" and correct the output of the upstream operator that caused the error. For example, in the workflow of Figure 1, fixing the null values in the output of `Join` requires correcting the output of rows B and/or C.

## 2.3 Research Challenges

From a dataflow system's perspective inflight error detection and correction entails three major challenges:

*Dataflow Model.* We require a dataflow model based on directed cyclic graphs that enables inflight modifications to upstream data by downstream operators (i.e., error handlers) and re-propagate updates downstream. This requires incremental and cyclic computations across dataflow operations. For example, in Figure 1, the downstream operator 10 upon detecting an error updates the upstream data (C to C'), by correcting the output of `ocr()` in the above example using an alternative `ocr` (operator 6′), but only for erroneous rows detected in the output of operator 9.

Current state-of-the-art dataflow systems can not address this problem sufficiently. For instance, while Spark [54] adopts a directed acyclic graph based model, Flink [12] require barrier synchronization between loop iterations, leading to inefficiencies in propagating incremental inflight updates. Naiad [39] supports incremental cyclic computations, but does not support unstructured loops (as in Figure 1). Moreover, current dataflow systems and data debugging approaches do not support arbitrary inflight updates to upstream intermediate output. For instance, while correcting the output of `odt()` requires appending a new row to its output, that of `findNP()` and `ocr()` requires editing a row (i.e., deleting the incorrect one followed by appending a corrected one).

*Dataflow Execution.* Cyclic graph-based execution model requires meticulous data synchronization between operators. For instance, the error handler (operator 10; Figure 1), upon detecting an erroneous tuple, can lead to re-processing of row B via an alternative `ocr()` engine (operator 6′). This conceptually requires propagating a deleted tuple C along with a new corrected tuple C' downstream, which eventually requires deleting the erroneous tuple D and appending the tuple D'. The challenge here lies in pipeline parallel execution of the dataflow operators in presence of such inflight updates to intermediate upstream data.

*Error Handling Abstractions.* We require convenient abstractions for error handling operators that can seamlessly be interleaved with existing dataflow operators. For instance, how can the workflow developer specify error handlers (such as operators 2, 10, and 11). Current state of the art dataflow systems do not offer such abstractions. Moreover, interfaces proposed by data debugging systems do not lend themselves to user-defined error correction functions.

Overall, none of the existing dataflow systems provide an effective and efficient way for inflight error handling in ML workflows.

# 3 REACTIVE DATAFLOW

We now present reactive dataflow, directed cyclic dataflow graphs for supporting inflight error handling, our preliminary execution engine that can efficiently run reactive dataflows, and our abstractions for defining error handlers. We also outline open challenges.

## 3.1 Directed-cyclic dataflow graphs

A cyclic dataflow graph-based programming model allows data to "flow" back to an upstream operator. This is crucial for inflight error handling, where an error might be only evident in the output of some downstream operator. Let $G = (\mathcal{O}, \mathcal{E}_f \cup \mathcal{E}_b)$ be a directed cyclic graph, where $\mathcal{O}$ is a set of operators (vertices), and $\mathcal{E}_f$ and $\mathcal{E}_b$ are sets of forward and backward dataflows (edges). For example, Figure 2 shows the reactive dataflow for example workflow of Section 2 with error handlers (for now ignore the edge labels). Here, for instance $o_{10} \rightarrow o_5$ is a backward edge while $o_9 \rightarrow o_{10}$ is a forward edge. A forward edge $o_i \rightarrow o_j \in \mathcal{E}_f$ denotes that output of the operator $o_i$ is consumed by $o_j$.

In reactive dataflow, backward edges $o_j \rightarrow o_i \in \mathcal{E}_b$ denote that the operator $o_j$ modifies the output of upstream operator $o_i$. Let $R$ denote the rowset (i.e., the output) of an operator $o$. Each backward edge in $\mathcal{E}_b$ is labelled with an update operation $op$, which can be an append (denoted by +) or a delete (−)[3]. A backward edge with label $op \in \{+, -\}$, denoted $o_i \xrightarrow{op} o_j$ implies that the edge updates the output $R_j$ as $R_j = R_j + \Delta^{op} R_j$. In other words, the backward edge $o_i \xrightarrow{op} o_j$ updates the output $R_j$ of $o_j$ by $\Delta R_j$ (either by appending and/or deleting a row). For example, while the edges $o_{10} \rightarrow o_5$ and $o_{11} \rightarrow o_6$ edit the output rows of operators $o_5$ and $o_6$, respectively, the edge $o_2 \rightarrow o_1$ appends row(s) to the output of $o_1$.

Our goal is to efficiently propagate these inflight updates downstream. To effectively and efficiently propagate inflight updates, it is important to capture the transformation properties of the operators with respect to handling changes in their inputs. In what follows, we formally define these properties for the forward edges.

Inflight updates (appends and deletes) may not propagate incrementally through operators. For instance, an `OrderBy` operator assigning an absolute rank to each input row might require a full pass over all the rows upon appends and deletes in its input. We

[3]We do not explicitly consider edit operation(±) here as they are essentially deletes followed by appends, i.e., $\Delta^{\pm} R = \Delta^{-} R + \Delta^{+} R$.

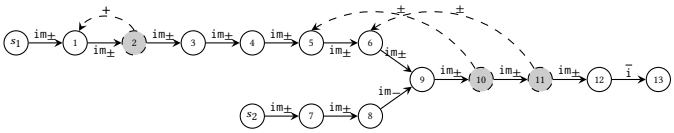



**Figure 2: Reactive dataflow for the example workflow of Section 2. Solid vertices are dataflow transformations and gray vertices are error handlers.**

say that a forward edge $o_i \rightarrow o_j$ is *non-incremental* if

$$R_i = R_i + \Delta^{op} R_i \rightsquigarrow R_j = o_j(R_i + \Delta^{op} R_i) \quad (1)$$

In contrast, other operators may be able to incrementally handle appends and/or deletes. However append in inputs can turn into deletes in outputs and vice-versa upon forward propagation. To see this, consider a forward edge $o_i \rightarrow o_j$ where the operator $o_i$ outputs a rowset of integers and $o_j$ computes the minimum. For $R_i = \{2, 4, 3, 5\}$, $o_j$ will output $R_j = \{2\}$. Further, assume that a backward edge updated $R_i$ by deleting 2, i.e., $R_i + \Delta^- R_i = \{4, 3, 5\}$. This update in $R_i$ requires updating $R_j$ by deleting 2 and appending 3, which may further lead to propagating deletes and appends, and so on. Therefore, we also carefully capture the *monotonicity behavior* in our characterization of incremental forward edges.

We say that a forward edge $o_i \rightarrow o_j$ is *incremental and monotonic* if an append (delete) in $R_i$ leads to appends (deletes) in $R_j$, i.e.,

$$R_i = R_i + \Delta^{op} R_i \rightsquigarrow R_j = R_j + \Delta^{op} R_j \quad (2)$$

Likewise, a forward edge $o_i \rightarrow o_j$ is *incremental and non-monotonic* if an append or delete in $R_i$ leads to both appends and deletes in $R_j$, i.e.,

$$R_i = R_i + \Delta^{op} R_i \rightsquigarrow R_j = R_j + \Delta^{+} R_j + \Delta^{-} R_j \quad (3)$$

Based on equations (1)–(3), we label each forward edge $o_i \rightarrow o_j$ based on the edge type and update operation as shown in Table 1.

The reactive dataflow in Figure 2 shows each forward edge along with its label. For example, $o_5 \xrightarrow{\texttt{im}_\pm} o_6$ denotes that $R_5 = R_5 + \Delta^+ R_5 \rightsquigarrow R_6 = R_6 + \Delta^+ R_6$ and $R_5 = R_5 + \Delta^- R_5 \rightsquigarrow R_6 = R_6 + \Delta^- R_6$, i.e., both appends and deletes on the output of $o_5$ can be incrementally propagated and that the transformation applied by $o_6$ is monotonic. As another example, $o_6 \xrightarrow{\texttt{im}_-} o_9$ denotes that $R_6 = R_6 + \Delta^- R_6 \rightsquigarrow R_9 = R_9 + \Delta^- R_9$ but $R_6 = R_6 + \Delta^+ R_6 \rightsquigarrow R_9 = o_9(R_6 + \Delta^+ R_6)$, i.e., while deletes on the output of $o_6$ can be incrementally propagated, appends cannot, and the incremental handling of deletes by $o_9$ is monotonic.

*Open challenges.* Note that the label of a forward edge $o_i \rightarrow o_j$ is a property of the operator $o_j$ in terms of handling updates from $o_i$. While we provide default properties for several standard dataflow operators, shown in Appendix A, automatically inferring these properties using static code analysis will be a challenge. For now, UDFs manually declare these properties for themselves.

## 3.2 Execution engine

*Execution planning.* Our prototypical system Popper does pipeline parallel execution of reactive dataflows. To plan pipelines, it first builds an *auxiliary graph* that captures blocking and non-blocking behaviors of operators with respect to changes in their inputs. Auxiliary graph separates operator $o$ into operators $o^+$ and $o^-$ that append ($\Delta^+ R$) and delete ($\Delta^- R$) rows in $o$'s output $R$. The blocking and non-blocking behavior follows directly from the edge properties in Table 1.

| Property | $R_i + \Delta^+ R_i \rightsquigarrow$ | $R_i + \Delta^- R_i \rightsquigarrow$ | Label |
|---|---|---|---|
| Non-incremental | $o_j(R_i + \Delta^+ R_i)$ | $o_j(R_i + \Delta^- R_i)$ | $\overline{\texttt{i}}$ |
| Incremental monotonic | | | |
| Only appends | $R_j + \Delta^+ R_j$ | $o_j(R_i + \Delta^- R_i)$ | $\texttt{im}_+$ |
| Only deletes | $o_j(R_i + \Delta^+ R_i)$ | $R_j + \Delta^- R_j$ | $\texttt{im}_-$ |
| Both | $R_j + \Delta^+ R_j$ | $R_j + \Delta^- R_j$ | $\texttt{im}_\pm$ |
| Incremental non-monotonic | | | |
| Only appends | $R_j + \Delta^+ R_j + \Delta^- R_j$ | $o_j(R_i + \Delta^- R_i)$ | $\texttt{i}\overline{\texttt{m}}_+$ |
| Only deletes | $o_j(R_i + \Delta^+ R_i)$ | $R_j + \Delta^+ R_j + \Delta^- R_j$ | $\texttt{i}\overline{\texttt{m}}_-$ |
| Both | $R_j + \Delta^+ R_j + \Delta^- R_j$ | $R_j + \Delta^+ R_j + \Delta^- R_j$ | $\texttt{i}\overline{\texttt{m}}_\pm$ |

**Table 1: Incremental and monotonic edge properties and their labels.**

Due to page limit, we skip a more formal treatment of building auxiliary graph and demonstrate it with an example in Figure 3b corresponding to reactive dataflow in Figure 3a. We denote non-blocking behavior of an operator with respect to changes in its input using solid black edges, i.e., when operators can be pipelined, and blocking behavior by dashed red edges.

For example, since $o_1 \xrightarrow{\texttt{im}_\pm} o_2$, operator $o_2$ is non-blocking with respect to both appends and deletes in its input $R_1$. Since $o_2 \xrightarrow{\texttt{i}\overline{\texttt{m}}_+} o_3$, operator $o_3$ is non-blocking with respect to appends in its input $R_2$, but is blocking with respect to deletes in $R_2$. Further observe that an append ($\Delta^+ R_2$) in $R_2$ will lead to both appends ($\Delta^+ R_3$) and deletes ($\Delta^- R_3$) in $R_3$. Likewise, since $o_6 \xrightarrow{\overline{\texttt{i}}} o_7$, operator $o_7$ is blocking with respect to both appends and deletes in its input.

Using the auxiliary graph, we identify operator pipelines starting from either a source or an operator that consumes dataflow from a blocking edge, and ends at either a sink or a blocking edge. Pipelines in reactive dataflow may contain cycles due to backward edges introduced by error handlers. In Figure 3b, for example, there are four pipelines, as shown in dashed gray boxes, with Pipeline 3 having backward edges. Since pipelines can overlap, we further assign operators to non-overlapping stages; operators within each stage can form a pipeline. In Figure 3b, for example, there are four stages shown in solid yellow boxes. Stages have dependencies on one another, following the dependencies in the auxiliary graph. Stages are executed in order based on their dependencies; for example, first, stage 1 and 2 are executed, followed by stage 3, and finally, stage 4. If a stage is dependent on itself, then a stage may be run multiple times e.g., stage 3 in Figure 3c.

*Execution.* Figure 4 shows the internals of Popper. It spawns stateless processes, including a driver process and worker processes to perform computation. It maintains shared states in an in-memory data store to coordinate among processes. Driver adds all the operators in ready stages to task sets maintained in the data store. Workers keep polling the task sets to pick an operator and perform work for them. For each operator instances $o_i^+$ and $o_i^-$, Popper maintains an output stream $\Delta^+ R_i$ and a deletion stream $\Delta^- R_i$ in

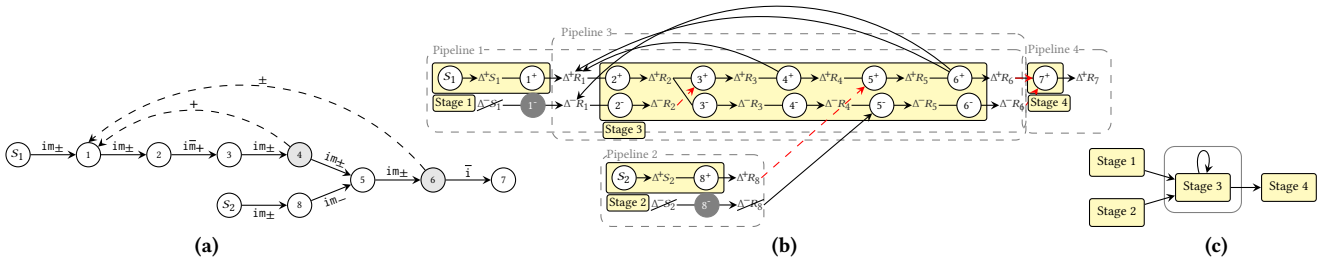



**Figure 3: (a) An example reactive dataflow; (b) An auxiliary dataflow: gray nodes denote dead nodes; dashed red arrows denote blocking edges; dashed boxes denote operator pipelines; and yellow boxes denote stages; (c) Staged execution.**

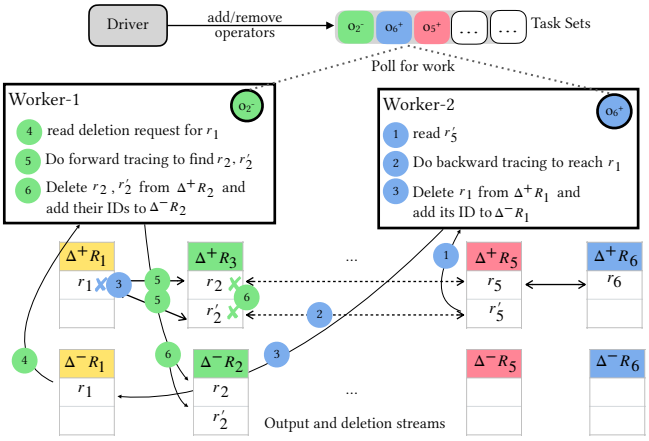



**Figure 4: Popper internals.**

the data store. When executing an operator, workers read from one or more streams of parent operators as per the auxiliary graph and write to the operator's output and deletion streams.

Error handlers, such as $o_6^+$, apply corrections using instance-level lineage pointers stored in the in-memory data store. For each row, these pointers are maintained as two separate sets: backward pointers identify immediate parent(s) of the row and are used for doing backward tracing, and forward pointers identify immediate children of the row and are used for incrementally propagating deletions. Figure 4 shows an example of how inflight updates are applied. We omit further details due to space constraints.

*Open challenges.* Currently, Popper only works with static reactive dataflow graphs. In a real-world scenario, programmers can be allowed to add new error handlers on-the-fly as and when they observe new errors in production workflows. Allowing such reconfigurations at runtime will be challenging.

Popper works on a single machine and maintains all the information, such as output rows and lineage, in a common data store. Sharding this common data store and scaling Popper to a distributed fault-tolerant setup is an open challenge.

## 3.3 Abstractions for building reactive dataflow

Our programming interface is based on the SCOPE language [13], which provides an easy way to express ML workflows, which are expressed as transformations over rowsets. *Transformations* take rowset(s) as input and output a rowset. Transformations are based on five primitive SCOPE operators: (i) *extractor* (for parsing and constructing rows from a source like a file); (ii) *processor* (for row-wise processing); (iii) *reducer* (for group-wise processing); (iv) *combiner* (for combining rows from two rowsets); and (v) *outputter* (for writing rows to a data sink). These primitives allow us to offer a rich API comprising well-known dataflow transformations such as `map`, `filter`, `groupBy`, and `join`, among others, and also allow users to define their own. For example, the `odt()` operator in Figure 1 is a user-defined processor, i.e., it row-wise processes each row (video frame) to output a rowset of object detections. We refer readers to [13] for an overview and formal semantics of SCOPE's operators.

To make inflight error handling a first-class citizen, Popper provides two operators: a `rowErrorHandler` and a `rowSetErrorHandler`. Error handlers can be composed with other SCOPE dataflow transformations to detect and correct errors on the fly. In what follows, we will describe the row error handler interface; the semantics of the rowset error handler are similar.

A row error handler allows users to specify a condition on a row to detect an error, which when holds, executes a user-defined correction function to "edit" the output row(s) of some upstream operator. More formally, `rowErrorHandler` interface provides three functions that the user needs to implement.

►`detect(f:row)→ bool`: It is used for error detection. The input is a UDF that receives a row and outputs true or false depending on the condition it evaluated on the row.

►`correct(f:row)→ None`: It is used for error correction. The input is a UDF that receives a row and corrects it by either appending or deleting or both any of its ancestor rows, i.e., output rows of upstream operators from which the current row was derived.

►`eventually(f:row)→ {PASS,DROP}`: It is used for specifying the behavior of the error handler upon detecting an error in the row a second time.

In the following, we give example of error handling operator 10 in workflow of Figure 1. Examples of error handling operators 2 and 11 can be found in Appendix B.

**Example 1:** *Implementing an error handler (operator 10) that detects null values in the output of join in Figure 1, and corrects the output of the upstream* `findNP()` *operator.*

```
class JoinErrorHandler1(RowErrorHandler):
 def detect(self, row):
  return (not row["r_v_num"] or not row["f_type"])

 def correct(self, row):
  b_box = row["n_box"]
  row.edit({"n_box": findNP2(b_box)})

 def eventually(self, row): return PASS
```

The above user-defined error handler first checks for the null values (line 3) in columns `r_v_num` and `f_type`, and if present, corrects the output of the upstream `findNP()` operator. Observe that here the user-defined error correction function (lines 6 and 7) first fetches the `b_box` from the upstream operator and then attempts to corrects `n_box` by applying an alternative `findNP2` function. Our API includes convenient methods to track the lineage in order to fetch and correct outputs of upstream operators. Also note that the eventual behavior of the error handler is set to `PASS` (line 9), which means that the handler will forward the erroneous tuple if it again detects an error the next time it receives it. A later error handler, operator 11, will correct the erroneous tuple.

*Open challenges.* Our current API requires the workflow developer to know the schema-level lineage, i.e., upon catching an error, which upstream column(s) need to be corrected. A friendlier development environment can be devised that uses static analysis to identify lineage to simplify writing error handlers.

## 4 PRELIMINARY EXPERIMENTS

Our goal in developing Popper is to support error detection and error correction in ML workflows. We develop a few ML workflows to investigate the following questions:

- ▸ Can error detection and error correction provided by Popper improve both precision and recall?
- ▸ Can Popper reduce the cost of running ML workflows? We evaluate two types of costs: (1) runtime cost measured by the end-to-end execution time; and (2) human-in-the-loop cost, measured by the number of post-facto manual corrections, needed by the workflows.
- ▸ How does the end-to-end runtime of Popper compare with other dataflow systems? Wherever possible, we implement error corrections by following the approach to support iterations in other dataflow systems.
- ▸ How does the runtime of Popper grow as we increase the percentage of errors in data?

### 4.1 Setup

We use six real-world ML workflows shown in Figure 5 and extend them with error handlers (shown in gray nodes). Workflows 1–3 are taken from HuggingFace [26] and they have the same structure.

**1)** hfp1: Text question answering workflow receives text contexts and questions about contexts from SQuaD 2.0 [43] dataset. The workflow invokes TinyBERT [31] to provide answers to questions.
**2)** hfp2: Visual question answering workflow receives images and questions about the images from VQAv2 [22] dataset. The workflow invokes ViLT-b32 [34] model to provide answers to questions.
**3)** hfp3: Image classification workflow uses the DeiT tiny [49] model to classify images from ImageNet-1K dataset [18] .

We extend each of these pipelines with an error handler that detects an error if the confidence of the inference is below a threshold and corrects it by running a heavier ML model. Error handler corrects errors using RoBERTa [38], BLIP [36], and ViT [50] for hfp1, hfp2, and hfp3 respectively.

**4)** ObjTrack workflow inspired from [21] uses a light-weight `CSRT` tracker [37] to track an object in input videos from the OTB2015 [51] dataset. Error handler verifies every sixth frame with a heavier model `SeqTrack` [14]. If the bounding boxes from the two models have an IoU that is below a threshold, the error handler updates the bounding box with the box found by the `SeqTrack` model.

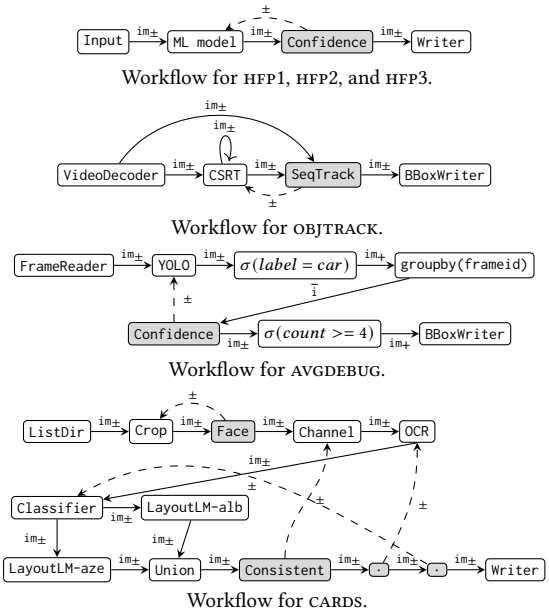



**Figure 5: ML workflows used for experiments.**

**5)** AVDebug workflow identifies frames containing four or more cars from a video feed of the front camera of a moving car. Such workflows are used by autonomous vehicle researchers looking for specific traffic scenarios [8, 33]. The workflow runs `YOLO` [45] on the camera images to find objects and their bounding boxes. If YOLO found less than four cars with high confidence, the `RowSetErrorHandler Confidence` detects an error and invokes `SECOND` model [53] on the corresponding LIDAR data to correct the boxes. The input is given from the NuScenes [11] dataset.

**6)** Cards workflow extracts 'first name', 'surname', 'date of birth', 'date of issue', 'date of expiry', 'gender', and 'place of birth' from rotated card images from the MIDV 2020 [10] dataset. The workflow first `crops` the card by finding a rectangle in the image. `RowErrorHandler Face` detects an error if a face cannot be detected in the cropped card image. It corrects the error by trying other orientations until it can find a face. Since cards often have colorful backgrounds, `Channel` removes background color channels to make the card amenable to `OCR`. `OCR` runs EasyOCR [19] to get text from images. A ResNet `classifier` identifies the country of the card and routes it to appropriate `LayoutLM` model [52] fine-tuned for that particular country's card. `Consistent` is a sequence of three error handlers which check that fields are not null and that dates are well formatted. When they detect errors, they first try to select other color channels, then try Tesseract OCR engine [48], and finally retry with the second prediction from the `Classifier`. This order is decided by the observed error rate of each upstream operator *i.e.,* changing channels fixes more errors than changing OCR engine which fixes more errors than changing card class. See Appendix C for details.

All experiments are done on a server with Intel Xeon Gold 6330 and an Nvidia A40 GPU. Popper is implemented in 13KLOC and uses Redis as its data store. We spawn 10 Popper workers.

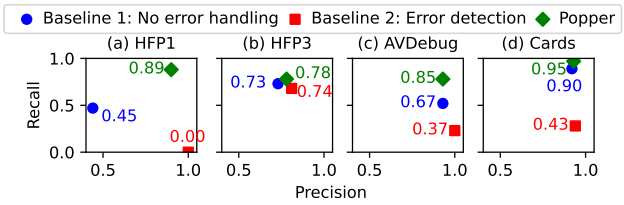



**Figure 6: Precision and recall without error handling, with error detection, and with error correction. Numbers in the figure show the F1 score.**

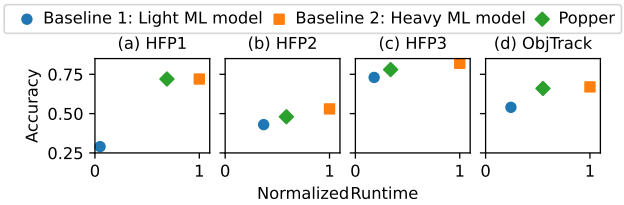



**Figure 7: Normalized runtime and accuracy from using just the heavy models, just the light models, and using the heavy model to correct errors made by the light model.**

## 4.2 Effectiveness of Inflight Error Handling

In our first set of experiments, we evaluate the effectiveness of inflight error handling considering precision and recall of ML workflows. We compared Popper against two baselines. baseline 1 is a workflow without any error handler, and baseline 2 has an error handler that only does error detection, i.e., the error handler simply drops data upon detecting an error. We show the effectiveness results in Figure 6 for workflows hfp1, hfp3, avdebug, and cards. The numbers in the Figure denote F1 scores.

We observe that baseline 2 improves precision but achieves a lower recall over baseline 1. This is because detecting errors and dropping erroneous data turns false positives (incorrect values) to false negatives (no value). We also observe that for all workflows, dropping erroneous data reduced F1 score. This is most pronounced in the Cards workflow, where F1 score drops from 0.90 to 0.43.

It is worth nothing that for all workflows, error detection and correction always leads to a higher recall. However, we observe that with error correction, precision can reduce compared to baseline 2. Precision is affected by the specified "eventual" behavior of the error handler (recall Section 3.3). For hfp1, hfp3, and avdebug, the eventual behavior of error handlers are set to pass *i.e.,* if the error handlers detect an error again after making a correction, they forward the row. This explains slightly lower precision of these workflows. We refer to Appendix D for further discussion on limitations of inflight error handling. Overall, we observe an improvement in F1 score from 0.05 up to 0.44 across different workflows due to inflight error handling.

*We conclude that inflight error handling can improve both precision and recall thereby increasing the effectiveness of ML workflows.*

## 4.3 Cost Improvements with Error Handling

*Human-in-the-loop cost.* Improvements in accuracy directly reduce the human-in-the-loop (HIL) cost, i.e., the number of times a human has to correct errors in workflows. We conduct two supplementary experiments that measured the HIL cost for hfp1 and cards workflows. For hfp1 (text question answering) we split the dataset into chatbot-like conversations. A conversation is a group of questions about the same text context. If the workflow misses the answer to any of the questions then it has to be sent to a human for examination. Likewise, for the cards workflow, if any of the seven fields are extracted incorrectly then the card has to be sent to a human for manually correcting incorrect fields.

For the hfp1 workflow, error handling reduces HIL cost by 26.3%: from 2030 to 1496 conversations required to be sent to human for correction. Similarly, for cards workflow it reduces HIL cost by 39.4%: from 307 to 186 cards required to be sent to human.

*Infrastructure cost.* In our next set of experiments, we evaluate the impact of error handling and the trade-off it offers considering accuracy and runtime—which correlates with infrastructure costs.

We compare ML workflows hfp1, hfp2, hfp3, and objtrack with two baselines: baseline 1 that uses "light" ML models; and baseline 2 that uses "heavy" ML models. In general, light ML models trade off accuracy for faster inference time than heavy ML models. Indeed, we observe in Figure 7 that baseline 1 (with light ML models) is 2–10× faster, but achieves 9–43% lower accuracy than baseline 2 (with heavy ML models).

Extending baseline 1 with a carefully designed error handler offers a better trade off. In particular, error handling allows improving accuracy of workflows with light ML models by only invoking the heavy ML model after detecting an error. For example, for the ObjTrack workflow in Figure 7, error handling can improve the accuracy of the workflow with light ML model from 0.54 to 0.66, closely matching the accuracy of the baseline 2 workflow with heavy ML model, which is 0.67. Moreover, it does so at just half the runtime of the baseline 2 workflow with heavy ML model.

For HFP2 and HFP3, workflow with error handling achieves an accuracy that is slightly lower compared to baseline 2 (with heavy ML model). This is because light ML models sometimes output wrong answers with high confidence and thus such errors were not detected by error handlers.

*Overall, combining light and heavy models with Popper's error handling capabilities offer a better trade off between accuracy and runtime, saving both HIL and infrastructure costs.*

## 4.4 System Efficiency

We now evaluate Popper's efficiency with respect to handling inflight dataflow updates. In particular, we evaluate Popper's pipeline parallel execution that allows to pipeline dataflow modifications via backward edges. We compare Popper's staged execution to that of Flink's [20] and Naiad's [39] iteration approaches as they also allow executing cyclic dataflows. It is worth noting that workflows such as those in Figure 5 cannot be expressed directly in Flink, Naiad and other dataflow systems as they lack native support for inflight error handling. Therefore for this evaluation, we simulate Flink's and Naiad's iterative computation in Popper.

We consider workflows cards, objtrack, and avgdebug for which Figure 8 shows the runtimes. We first observe that Flink-style and Naiad-style iterations cannot handle cards workflow

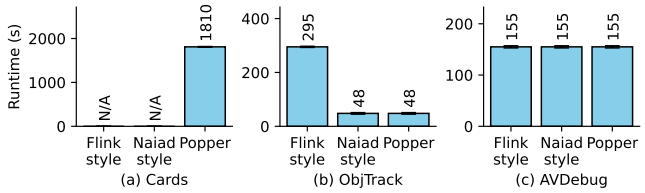



**Figure 8: Comparing runtime with error handling.**

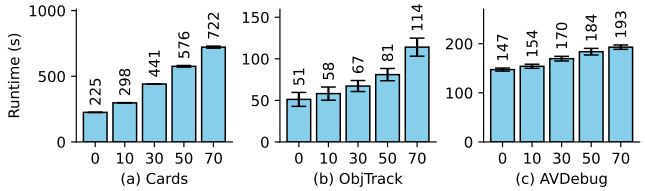



**Figure 9: Impact on runtime with increase in pct. (%) of errors.**

since they only support nested loops whereas the set of `Consistent` error handlers in the CARDS workflow make unstructured edits.

POPPER's staged execution and Naiad-style iterations outperform Flink-style iterations by 83.7% for OBJTRACK. This is due to pipeline parallel execution in POPPER, enabling faster operator pipelining between loop iterations. Conversely, Flink-style iterations necessitate barrier synchronization, resulting in a 6× slower runtime for OBJTRACK. This is because the `SeqTrack` verifier waits for the `CSRT` tracker to traverse the entire video before each update, significantly slowing down the workflow. For AVGDEBUG, all three iteration styles exhibit similar runtimes due to a pipeline breaker `groupby` $\xrightarrow{i}$`confidence`, preventing update pipelining.

*Scalability with respect to errors.* We additionally evaluated POPPER's performance by gradually increasing the percentage of erroneous inputs. Results in Figure 9 show that the runtime of each workflow increases proportionally with the increase in number of errors. For example, a 20% increase in number of errors increases the runtime by 63% for the CARDS workflow, by 37% for the OBJTRACK workflow, and by 9% for the AVGDEBUG workflow. This is expected as the number of iterations due to error handling also increases with more errors.

*In sum, we conclude that POPPER provides an efficient and versatile execution engine to run ML workflows with inflight error handling. POPPER offers good scalability with respect to errors in the data.*

## 5 RELATED WORK

We now relate the ideas put forward in this paper to existing prior work, which can be categorized into:

*Dataflow engines.* Dataflow systems such as MapReduce [17] and others [13, 29] adopt an execution model that is based on directed acyclic graphs. Hence, these systems cannot be used for inflight error handling that requires support for cycles. Spark [54] and Flink [20] support cyclic computations but requires barrier synchronization between loop iterations. As discussed in Section 4.4, this incurs overhead compared to POPPER's execution which supports pipelining between loop iterations. Timely dataflow [39] supports incremental and iterative computations, but like Spark and Flink it does not support unstructured loops such as in the CARDS workflow.

In contrast to these systems, POPPER's execution engine supports incremental and cyclic computations, specific to inflight error handling. In addition, none of the existing dataflow systems support abstractions for error detection and correction.

*Incremental computation.* Inflight error handling is reminiscent to supporting incremental updates, explored in incremental view maintenance [3, 9, 40, 55], streaming computation [5–7, 12, 15], and provenance-based selective replay [27]. These systems typically handle updates at the input, such as base table updates in incremental view maintenance or stream appends in streaming computation. These updates are controlled externally, therefore they lack visibility into inflight updates. In contrast, POPPER allows updates at intermediate operation outputs and provides visibility into the location of error handlers and type of update operations (+, −, ±) for planning stages.

*Data Debugging.* Error handling in workflows is also closely related to data debugging in dataflow systems. Systems such as Amber [35] and those based on Spark including Titian [28], BigSift [24], BigDebug [23], and TagSniff [16] support debugging primitives such as breakpoints (to pause and resume execution) and watchpoints (to inspect intermediate data). While these can be used for error detection, they offer limited support for inflight error correction. Vega based on BigDebug [23], aims to supports incremental updates, but is limited to incremental updates supported by Naiad, as it is based on differential dataflow [2], which has limitations discussed above. Overall, current debugging systems do not offer inflight error handling that involve both error detection and correction.

*Model Assertions.* [33] show how data consistency checks, called *model assertions*, can detect model errors at runtime. PTAV [21] and Focus [25] leverage heavy-weight models to detect and correct errors that are made by light-weight models in computer vision applications. PICARD [46] improves the accuracy of Text2SQL flow by detecting incorrect token predictions on-the-fly using a SQL parser. Detecting and correcting errors in such a manner have shown to improve the end-to-end accuracy. POPPER's error handler also supports such assertions, and in addition supports scenarios where errors can only be evident later in the workflow and corrections lead to updating upstream dataflows that are efficiently propagated downstream.

## 6 CONCLUSION

In this paper, we have proposed reactive dataflow, a new programming model that provides convenient abstractions for specifying user-defined error-handling operations. These error handlers integrate seamlessly with traditional data processing operations. We developed a categorization of dataflow edges based on the operators' transformation properties, which enable efficient pipeline parallel execution. We have built a prototype POPPER and have shown its efficacy in executing ML workflows, which are prone to errors introduced by ML models. Our evaluation showed that POPPER's inflight error handling capabilities allow for improving the accuracy and reducing the cost of ML workflows and that POPPER as a dataflow system is efficient. We have listed some open challenges that we plan to work on in future. Most importantly, we plan to extend POPPER to distributed shared-nothing environments.

## ACKNOWLEDGMENTS

We thank the members of the Practical Systems Lab, IIT Delhi for contributing to implementation of Popper. We also thank the anonymous reviewers for their feedback. This research was supported in part by Google India Research Awards.

## REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016), OSDI'16, USENIX Association, p. 265–283.

[2] Abadi, M., McSherry, F., and Plotkin, G. D. *Foundations of Differential Dataflow*, vol. 9034 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2015, p. 71–83.

[3] Agiwal, A., Lai, K., Manoharan, G. N. B., Roy, I., Sankaranarayanan, J., Zhang, H., Zou, T., Chen, M., Chen, J., Dai, M., Do, T., Gao, H., Geng, H., Grover, R., Huang, B., Huang, Y., Li, A., Liang, J., Lin, T., Liu, L., Liu, Y., Mao, X., Meng, M., Mishra, P., Patel, J., R, R. S., Raman, V., Roy, S., Shishodia, M. S., Sun, T., Tang, J., Tatemura, J., Trehan, S., Vadali, R., Venkatasubramanian, P., Zhang, J., Zhang, K., Zhang, Y., Zhuang, Z., Graefe, G., Agrawal, D., Naughton, J., Kosalge, S. S., and Hacigümüş, H. Napa: Powering scalable data warehousing with robust query performance at google. *Proceedings of the VLDB Endowment (PVLDB) 14 (12)* (2021), 2986–2998.

[4] Ahmad, Y., Kennedy, O., Koch, C., and Nikolic, M. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow. 5*, 10 (jun 2012), 968–979.

[5] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1033–1044.

[6] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. 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 (aug 2015), 1792–1803.

[7] Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I., and Zaharia, M. Structured streaming: A declarative API for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data* (2018), SIGMOD '18, Association for Computing Machinery, p. 601–613.

[8] Bang, J., Kakkar, G. T., Chunduri, P., Mitra, S., and Arulraj, J. Seiden: Revisiting query processing in video database systems. *Proc. VLDB Endow. 16*, 9 (may 2023), 2289–2301.

[9] Blakeley, J. A., Larson, P.-A., and Tompa, F. W. Efficiently updating materialized views. *SIGMOD Rec. 15*, 2 (jun 1986), 61–71.

[10] Bulatov, K., Emelianova, E., Tropin, D., Skoryukina, N., Chernyshova, Y., Sheshkus, A., Usilin, S., Ming, Z., Burie, J.-C., Luqman, M., and Arlazarov, V. MIDV-2020: A comprehensive benchmark dataset for identity document analysis.

[11] Caesar, H., Bankiti, V., Lang, A. H., Vora, S., Liong, V. E., Xu, Q., Krishnan, A., Pan, Y., Baldan, G., and Beijbom, O. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2020), pp. 11621–11631.

[12] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*, 4 (2015).

[13] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1265–1276.

[14] Chen, X., Peng, H., Wang, D., Lu, H., and Hu, H. Seqtrack: Sequence to sequence learning for visual object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2023), pp. 14572–14581.

[15] Confluent's ksqldb. https://www.confluent.io/product/ksqldb/.

[16] Contreras-Rojas, B., Quiané-Ruiz, J.-A., Kaoudi, Z., and Thirumuruganathan, S. Tagsniff: Simplified big data debugging for dataflow jobs. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 453–464.

[17] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. OSDI'04, USENIX Association, p. 10.

[18] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[19] Easyocr. https://github.com/JaidedAI/EasyOCR.

[20] Ewen, S., Tzoumas, K., Kaufmann, M., and Markl, V. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment 5*, 11 (2012).

[21] Fan, H., and Ling, H. Parallel tracking and verifying. *IEEE Transactions on Image Processing 28*, 8 (aug 2019), 4130–4144.

[22] Goyal, Y., Khot, T., Summers-Stay, D., Batra, D., and Parikh, D. Making the V in VQA matter: Elevating the role of image understanding in visual question answering. pp. 6904–6913.

[23] Gulzar, M. A., Interlandi, M., Yoo, S., Tetali, S. D., Condie, T., Millstein, T., and Kim, M. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 784–795.

[24] Gulzar, M. A., Wang, S., and Kim, M. Bigsift: automated debugging of big data analytics in data-intensive scalable computing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), pp. 863–866.

[25] Hsieh, K., Ananthanarayanan, G., Bodik, P., Venkataraman, S., Bahl, P., Philipose, M., Gibbons, P. B., and Mutlu, O. Focus: Querying large video datasets with low latency and low cost. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (2018), OSDI'18, USENIX Association, p. 269–286.

[26] Huggingface pipelines. https://huggingface.co/docs/transformers/main_classes/pipelines.

[27] Ikeda, R., Salihoglu, S., and Widom, J. Provenance-based refresh in data-oriented workflows. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (2011), CIKM '11, Association for Computing Machinery, p. 1659–1668.

[28] Interlandi, M., Shah, K., Tetali, S. D., Gulzar, M. A., Yoo, S., Kim, M., Millstein, T., and Condie, T. Titian: Data provenance support in spark. *Proc. VLDB Endow. 9*, 3 (nov 2015), 216–227.

[29] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), pp. 59–72.

[30] Jatowt, A., Coustaty, M., Nguyen, N.-V., Doucet, A., et al. Deep statistical analysis of ocr errors for effective post-ocr processing. In *2019 ACM/IEEE Joint Conference on Digital Libraries (JCDL)* (2019), IEEE, pp. 29–38.

[31] Jiao, X., Yin, Y., Shang, L., Jiang, X., Chen, X., Li, L., Wang, F., and Liu, Q. Tinybert: Distilling BERT for natural language understanding. *CoRR abs/1909.10351* (2019).

[32] Kang, D., Raghavan, D., Bailis, P., and Zaharia, M. Model assertions for debugging machine learning. In *NeurIPS MLSys Workshop* (2018), vol. 3, p. 10.

[33] Kang, D., Raghavan, D., Bailis, P., and Zaharia, M. Model assertions for monitoring and improving ml models. *Proceedings of Machine Learning and Systems 2* (2020), 481–496.

[34] Kim, W., Son, B., and Kim, I. ViLT: Vision-and-language transformer without convolution or region supervision. In *International Conference on Machine Learning* (2021), PMLR, pp. 5583–5594.

[35] Kumar, A., Wang, Z., Ni, S., and Li, C. Amber: A debuggable dataflow system based on the actor model. *Proc. VLDB Endow. 13*, 5 (jan 2020), 740–753.

[36] Li, J., Li, D., Xiong, C., and Hoi, S. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning* (2022), PMLR, pp. 12888–12900.

[37] Li, X., Hu, W., Shen, C., Zhang, Z., Dick, A., and Hengel, A. V. D. A survey of appearance models in visual object tracking. *ACM transactions on Intelligent Systems and Technology (TIST) 4*, 4 (2013), 1–48.

[38] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. RoBERTa: A robustly optimized BERT pretraining approach. *CoRR abs/1907.11692* (2019).

[39] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13)* (2013), pp. 439–455.

[40] Nikolic, M., Dashti, M., and Koch, C. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data* (Jun 2016), ACM, p. 511–526.

[41] Popper source code. https://anonymous.4open.science/r/popper-36CA/.

[42] Qian, X., and Wiederhold, G. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering 3*, 3 (1991), 337–341.

[43] Rajpurkar, P., Jia, R., and Liang, P. Know what you don't know: Unanswerable questions for SQuAD. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Melbourne, Australia, July 2018), I. Gurevych and Y. Miyao, Eds., Association for Computational Linguistics, pp. 784–789.

[44] Redis- in-memory data store. https://redis.io/.

[45] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 779–788.

[46] Scholak, T., Schucher, N., and Bahdanau, D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings*

*of the 2021 Conference on Empirical Methods in Natural Language Processing* (Nov. 2021), Association for Computational Linguistics, pp. 9895–9901.

[47] Shankar, S., Garcia, R., Hellerstein, J. M., and Parameswaran, A. G. Operationalizing machine learning: An interview study, 2022.

[48] Smith, R. An overview of the Tesseract OCR engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)* (2007), vol. 2, IEEE, pp. 629–633.

[49] Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jegou, H. Training data-efficient image transformers and distillation through attention. In *International Conference on Machine Learning* (July 2021), vol. 139, pp. 10347–10357.

[50] Wu, B., Xu, C., Dai, X., Wan, A., Zhang, P., Yan, Z., Tomizuka, M., Gonzalez, J., Keutzer, K., and Vajda, P. Visual transformers: Token-based image representation and processing for computer vision, 2020.

[51] Wu, Y., Lim, J., and Yang, M.-H. Object tracking benchmark. *IEEE Transactions on Pattern Analysis and Machine Intelligence 37* (2015), 1834–1848.

[52] Xu, Y., Li, M., Cui, L., Huang, S., Wei, F., and Zhou, M. Layoutlm: Pre-training of text and layout for document image understanding.

[53] Yan, Y., Mao, Y., and Li, B. Second: Sparsely embedded convolutional detection. *Sensors 18*, 10 (2018), 3337.

[54] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), NSDI'12, USENIX Association.

[55] Zhuge, Y., García-Molina, H., Hammer, J., and Widom, J. View maintenance in a warehousing environment. *SIGMOD Rec. 24*, 2 (may 1995), 316–327.

# APPENDIX

## A EDGE PROPERTIES

Table 2 gives some examples of built-in transformations, their properties, and forward edge labels. For example, a map transformation is incremental and monotonic with respect to both appends and deletes in its input. An inner hash join is incremental and monotonic for both appends and deletes in its left input, but for only deletes in its right input. This means that appending a row in its right input requires re-computation of the join. As another example, the left join transformation has the same property as inner join for its left edge. However for deletes in its right input, it is now non-monotonic, i.e., deleting rows from its right input might lead to an append to its output.

## B INFLIGHT ERROR HANDLING API EXAMPLES

We provide some more examples to illustrate our error handling APIs. In particular, we show the implementation of error handling operators 11 and 2 in Figure 1.

**Example 2.** *Implementing an alternative error handler (operator 11) that detects null values in the output of join in Figure 1, and corrects the output of the upstream* `ocr()` *operator.*

```
1 class JoinErrorHandler2(RowErrorHandler):
2  def detect(self, row):
3   return (not row["r_v_num"] or not row["f_type"])
4
5  def correct(self, row):
6   n_box = row["n_box"]
7   row.edit({"v_num": ocr2(n_box)})
8
9  def eventually(self, row): return DROP
```

The above user-defined error handler tries to fix the null values by first fetching the input `n_box` and correcting `ocr()` output by using an alternate `ocr2` function (lines 6 and 7). In this example, it might be the case that `ocr2` also causes an error that eventually leads to null values in the output of join. To handle this, we define the eventual behavior of the error handler to `DROP` (line 9), which implies that the error handler will not forward the erroneous row downstream.

**Example 3.** *Implementing an error handler (operator 2) to detect and correct "flickering" objects in the output of object detection and tracking in Figure 1.*

```
1 class ODTErrorHandler(RowsetErrorHandler):
2  self.error_idxs = []
3
4  def detect(self, rowSet):
5   # rowSet is a time-sorted detections of the same obj_id
6   for idx in range(len(rowSet)-1):
7    row1, row2 = rowSet[idx], rowSet[idx+1]
8    if row1["f_id"] + 1 != row2["f_id"]:
9     self.error_idxs.append(idx)
10
11  if len(self.error_idxs) == 0: return False
12  return True
13
14 def correct(self, rowSet):
15  for idx in self.error_idxs[:-1]:
16   prev_row, next_row = rowSet[idx], rowSet[idx+1]
17   fid1, fid2 = prev_row["f_id"], next_row["f_id"]
18   bbox1, bbox2 = prev_row["bbox"], next_row["bbox"]
19   for fid in range(fid1 + 1, fid2):
20    bbox = interpolate(bbox1, bbox2, fid1, fid2, fid)
21    prev_row.duplicate({"fid": fid, "bbox": bbox})
22
23 def eventually(self, rowSet): return PASS
```

Example 3 shows how a rowset error handler can be used to detect the flickering error caused by missing object detection in workflow of Figure 1. In contrast to a row error handler, a rowset error handler allows processing a set of rows to detect and correct errors. Note that in the above example, a `groupBy(obj_id)` (not shown for brevity) precedes the error handler. Here the detection function (lines 5–12), checks successive `f_ids` for missing detection. For instance, if the model detected the same truck (e.g., `obj_id=3`) in frames 3 and 5, but not in frame 4. The correction function (lines 15–21) simply duplicates rows for missing detections, i.e., it duplicates the previous row with updated `f_id` and `bbox`.

## C ONE SIZE DOES NOT FIT ALL

In our ML workflows, we find that there is no combination of ML models and other workflow settings that works best for all inputs. For example in the Cards workflow discussed in Section 4, we found that if we use all color channels with EasyOCR without error handling, we could correctly extract 83.1% of all the fields across all the cards. But when we remove blue and green channels from the input image, we can correctly extract 4.7% new fields. We cannot always remove blue and green channels, because it misses 6.8% fields extracted by the original workflow. Similarly, changing OCR engine to Tesseract can extract 2.7% new fields.

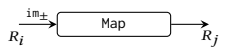
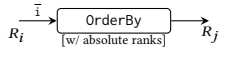
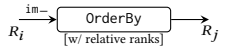
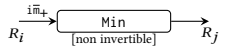
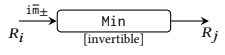
| Transformation (w/ edge labels) | Transformation property | Discussion |
|---|---|---|
|  | $R_i = R_i + \Delta^+ R_i \rightsquigarrow R_j = R_j + \Delta^+ R_j$<br>$R_i = R_i + \Delta^- R_i \rightsquigarrow R_j = R_j + \Delta^- R_j$ | *Map transformations are monotonic and can incrementally handle both appends and deletes.* |
|  | $R_{i_1} = R_{i_1} + \Delta^+ R_{i_1} \rightsquigarrow R_j = R_j + \Delta^+ R_j$<br>$R_{i_1} = R_{i_1} + \Delta^- R_{i_1} \rightsquigarrow R_j = R_j + \Delta^- R_j$<br>$R_{i_2} = R_{i_2} + \Delta^+ R_{i_2} \rightsquigarrow R_j = R_{i_1}$ HashJoin $R_{i_2}$<br>$R_{i_2} = R_{i_2} + \Delta^- R_{i_2} \rightsquigarrow R_j = R_j + \Delta^- R_j$ | *Hash join builds a hash table from the right inputs and uses it to join left inputs. Therefore, it can incrementally handle appends only in left inputs. It can incrementally handle deletes from both inputs using lineage.* |
|  | $R_{i_1} = R_{i_1} + \Delta^+ R_{i_1} \rightsquigarrow R_j = R_j + \Delta^+ R_j$<br>$R_{i_1} = R_{i_1} + \Delta^- R_{i_1} \rightsquigarrow R_j = R_j + \Delta^- R_j$<br>$R_{i_2} = R_{i_2} + \Delta^+ R_{i_2} \rightsquigarrow R_j = R_{i_1}$ LeftJoin $R_{i_2}$<br>$R_{i_2} = R_{i_2} + \Delta^- R_{i_2} \rightsquigarrow R_j = R_j + \Delta^- R_j + \Delta^+ R_j$ | *Similar to hash join with the difference that deleting from right input is non-monotonic: deleting $r_{i_2}$ from $R_{i_2}$ may delete a $(r_{i_1}, r_{i_2})$ from $R_j$ and append a $(r_{i_1}, null)$ to $R_j$.* |
|  | $R_i = R_i + \Delta^+ R_i \rightsquigarrow R_j =$ OrderBy$(R_i)$<br>$R_i = R_i + \Delta^- R_i \rightsquigarrow R_j =$ OrderBy$(R_i)$ | *Both deleting a row from input and appending a new row to input requires re-assigning new absolute rank to many rows. Hence, it is non-incremental.* |
|  | $R_i = R_i + \Delta^+ R_i \rightsquigarrow R_j =$ OrderBy$(R_i)$<br>$R_i = R_i + \Delta^- R_i \rightsquigarrow R_j = R_j + \Delta^- R_j$ | *Deleting a row from input can just delete its corresponding row from output without updating ranks of every other row.* |
|  | $R_i = R_i + \Delta^+ R_i \rightsquigarrow R_j = R_j + \Delta^- R_j + \Delta^+ R_j$<br>$R_i = R_i + \Delta^- R_i \rightsquigarrow R_j =$ Min$(R_i)$ | *Appends are incremental and non-monotonic: a new input row might delete the old minimum from the output and append a new minimum to the output.* |
|  | $R_i = R_i + \Delta^+ R_i \rightsquigarrow R_j = R_j + \Delta^- R_j + \Delta^+ R_j$<br>$R_i = R_i + \Delta^- R_i \rightsquigarrow R_j = R_j + \Delta^- R_j + \Delta^+ R_j$ | *In addition to handling appends incrementally as above, maintains a min-heap to similarly handle deletes.* |

**Table 2: Example transformations with transformation properties and forward edge labels.**

| EasyOCR | |
|---|---|
| Cards with all colors and Cards with red color | 76.3% |
| Cards with all colors but not Cards with red color | 6.8% |
| Cards with red color but not Cards with all colors | 4.7% |
| Neither Cards with all colors nor Cards with red color | 12.2% |
| **All color channels** | |
| Cards with Tesseract and Cards with EasyOCR | 36.9% |
| Cards with EasyOCR but not Cards with Tesseract | 46.2% |
| Cards with Tesseract but not Cards with EasyOCR | 2.7% |
| Neither Cards with EasyOCR nor Cards with Tesseract | 14.2% |

**Table 3: Percentage of fields correctly extracted by variations of Cards workflow (Section 4) using different workflow settings without error handling.**

## D LIMITATIONS AND DISCUSSION

Inflight error handling improves both precision and recall (Figure 6). But, we observe that even with inflight error handling, it is often not possible to obtain perfect precision and recall. We observe three reasons for this. We give examples from the Cards workflow from Section 4 to describe them.

*Correlated errors.* Using alternative models and other workflow parameters to correct errors is successful only if the errors made by these alternatives are uncorrelated. This may not always be the case. For example, all the alternatives in Cards workflow extract gender as 'F' when the ground truth is 'M' for `est_id/54.jpg`. In such scenarios where alternatives make the same errors, correcting errors is not effective. Therefore, we expect ML models trained using the same training dataset will not make up for effective error correction as they may have correlated errors.

*Imperfect assertions.* For `aze_passport/68.jpg`, the initial Cards workflow extracts name as 'DURNA DILAVAR QIzI' when ground truth is 'DURNA'. Running the other OCR engine correctly extracts the name but it does not get to run for this card. This is because the erroneous extraction also passes the error detection logic which just checks for a non-empty name. Detecting all errors may not always be possible.

*Failure to apply corrections.* After catching a mistake made by the `Crop` step, `Face` tries various orientations and model thresholds to find an orientation in which it can find a face. But it fails to find such an orientation for 5 cards, such as `esp_id/12.jpg`. This is because the face detection model itself fails to find the face even in the correct orientation.