Incremental Exception Resolution in
Tuplex

Benjamin Givertz

Advisor: Malte Schwarzkopf
Reader: Shriram Krishnamurthi

Department of Computer Science
Brown University
Providence, RI
May 2022
# Contents

Acknowledgements 3

Abstract 4

1 Introduction 5
   1.1 Tuplex Overview 6
   1.2 Data Science Workflow 7
   1.3 Problem Statement 8
   1.4 Proposed Solution 8

2 Background 9
   2.1 Schema Exceptions 10
   2.2 UDF Exceptions 11
   2.3 Debugging 11

3 Incremental Exception Resolution 12
   3.1 Caching 12
   3.2 Pipeline Comparison 13
   3.3 Merging 18

4 Experimental Evaluation 24
   4.1 Methodology 24
   4.2 Zillow 25
   4.3 Synthetic 30

5 Discussion 34
   5.1 Further Research 35

6 Conclusion 35

References 38
Acknowledgements

Thank you to Leonhard Spiegelberg for bringing me on board Tuplex in the spring of 2021. Over the past three semesters, I have had the amazing opportunity to work with Leonhard and learn about data science, systems, and research as a whole. Leonhard has offered guidance, support, and expertise on every step of this thesis, and I am so grateful for his mentorship.

Thank you to my thesis advisor, Malte Schwarzkopf, for supporting me throughout my time working on Tuplex. Malte’s attention to detail, prodding questions, and interest in my work helped me grow and kept me engaged throughout this process.

Thank you to my reader, Shriram Krishnamurthi, for offering his systems expertise in reviewing my research. During my first semester at Brown in the fall of 2018, Shriram’s excitement, ability to engage, and passion catalyzed my interest in computer science when I took his introductory course.

Thank you to my parents, Joanne and Michael, my siblings Sam and Hannah, and all of my wonderful friends who have supported me throughout my undergraduate experience at Brown and made these past four years so memorable.
Abstract

As data scientists scale their workloads, the cost of development scales accordingly because the time it takes to execute a single query increases with the size of the input data. Due to dirty data and unexpected input values, data scientists must test and iterate upon their pipelines to make them resilient to failures. Each time they update their pipeline, data scientists must re-execute the entire workload even if the change only affects a small number of error-causing rows. Throughout this process, data scientists bear the costs of unnecessary computations that state-of-the-art systems, such as Apache Spark, repeatedly perform over the majority of rows that do not produce failures. In this thesis, we present an efficient workflow for incrementally resolving errors that occur due to malformed input or unexpected values. To do so, we build upon Tuplex, a data analytics framework for Python. By removing unnecessary computation steps, incremental exception resolution allows data scientists to resolve exceptions that occur during pipeline execution efficiently. Experimentally, our implementation demonstrated a 37.06% end-to-end runtime savings over the current version of Tuplex when compared using a real-world dataset and pipeline. The efficiency improvements of incremental exception resolution correspond to significant impacts on time and money savings as well as a reduction in the environmental impact of data science development.
1 Introduction

From manufacturing plants to elementary school classrooms, data-driven decision-making has rapidly risen to prominence over the past decade. By analyzing vast quantities of data, industries across the globe have significantly improved their productivity, efficiency, and effectiveness [3, 11]. As the demand for data-driven decision-making increases, so does the demand for data collection, processing, and analytics. As businesses grow their operations, products, and users, the amount of data they collect and process scales accordingly. Thus, data scientists require powerful tools and compute engines that can keep up with the ever-growing supply of data.

In 2016, engineers at Meta set out to solve a scaling issue of their own. As the amount of data they needed to process for a single analytics query increased to over 60 terabytes, they determined that the 60-80 hour execution time of their current Hive-based system was no longer sustainable. Instead, they sought to switch their compute engine to Apache Spark hypothesizing that the framework could reduce overall processing time. However, developing a Spark system was not a straightforward task. As is often the case, the engineers found that "running a single Spark job for such a large pipeline didn’t work on the first try, or even on the 10th try" [4]. As they scaled up the amount of input data, their workload would fail due to unexpected input values from malformed or missing data entries. Throughout multiple development and debugging cycles, the engineers slowly scaled up the amount of input data from a few gigabytes to the full 60 terabyte workload. Each time they increased the size of the input data, they updated their pipeline with resolving logic to account for novel failures. In the end, the engineers reduced the execution time of their query to 10-20 hours.

The case study from Meta highlights the significant incentives that data scientists have to experiment and iterate over their existing workflows to achieve substantial speedups. However, it also demonstrates the arduous development process that takes place behind the scenes. When implementing systems at such a large scale, data scientists work iteratively by starting with a small amount of data and making improvements and bug fixes until the full workload can be processed. As they increase the size of their testing data, the time it takes to determine success also increases. While an optimized data analytics framework may reduce the overall execution time of a finished and polished pipeline, the time consumed by the development process is often overlooked.

When queries take hours or days to complete, failures that occur along the way are costly. However, when working with complex pipelines and large amounts of data, failures are all too common due to dirty datasets– those that contain malformed, inconsistent, or otherwise erroneous data [17]. During pipeline development, data scientists must make assumptions about the expected format of their input data. While certain violations of these assumptions may be simple to account for, unexpected edge cases can arise as the size of the input data scales. With many frameworks, a single runtime error may result in the entire workload
failing even if the pipeline could have successfully processed every other row. Thus, data scientists must debug the failure and reprocess the entire query when exceptions occur.

In this thesis, we explore the cost that exceptions and exception resolution impose on the data science development cycle and how that cost can be mitigated. To answer this question, we introduce Tuplex, a data analytics framework that uniquely handles and resolves exceptions, and justify why it is well-positioned to tackle this challenge. Building upon its existing capabilities, we implement a novel system for efficiently resolving exceptions that occur throughout the development process using incremental exception resolution. Finally, we evaluate the performance of the system experimentally with a real-world use case and dataset.

1.1 Tuplex Overview

With easy-to-learn syntax, powerful expressive capabilities, and a vast community of active developers, Python is by far the most widely used language for data scientists today. However, Python code can be orders of magnitude slower to execute than equivalent code in a compiled language such as C [10, 21]. While this overhead may not be significant for small tasks, as a data science workflow scales, interpreted Python code alone is no longer a viable option. At the same time, translating a pipeline from Python into hand-optimized and compiled native code is time-consuming and opens the door for new bugs.

Tuplex is a parallel data analytics framework that allows data scientists to write data processing pipelines in pure Python and compiles them into optimized native code for efficient evaluation [18]. Experimentally, Tuplex outperforms the processing speed of related frameworks, such as Apache Spark and Dask, on large, real-world datasets by a factor of 5-91x. Tuplex uses a novel dual-mode execution model in order to accomplish this. By sampling rows from the input data, Tuplex determines a normal case schema to aggressively optimize for and compiles LLVM bytecode to process these rows. During its initial pass over the dataset, Tuplex employs a lightweight check to determine whether each row conforms to the normal case. Rows that do not match the normal case schema assumption are pooled together as temporary exceptions for later processing on more generalized code paths. By separating the input data into separate cases based on their schema, Tuplex achieves efficient processing for the majority of input data while still allowing for schema violations to be resolved later.

Furthermore, Tuplex is uniquely able to handle exceptions that arise during pipeline execution. There are two ways an exception can be raised. The first, as mentioned previously, is when a row does not conform to the normal case schema assumption. The second is when a user-defined function (UDF) raises a Python exception during runtime such as a `TypeError` or `ZeroDivisionError`. In both cases, Tuplex attempts to resolve these exceptions by processing the violating rows in either a code path typed for a more general schema or, as a last resort, the Python interpreter. Moreover, Tuplex allows data scientists to write custom resolving logic within their pipeline, which specifies a behavior to apply if a certain
exception occurs at any stage of pipeline execution. By separating the core pipeline logic from exception resolution logic, Tuplex further optimizes the efficiency of the normal case. With these tools, data scientists can maximize the amount of data that can be processed from a dataset and do not need to clean it prior to pipeline execution.

However, Tuplex’s exception handling has a significant flaw. After executing a pipeline, observing what exceptions have occurred, and adding in resolving logic, data scientists must re-execute their entire workload to resolve the exceptions. In doing so, Tuplex reprocesses the entirety of the input data even though only the previous exceptions will produce novel output.

1.2 Data Science Workflow

Data scientists are tasked with implementing complex pipelines to process and analyze large amounts of data. Whether it be for statistical analysis, data cleaning, or general transformation, the goal of a data science pipeline is to consume raw input data, transform it, and output information in a more simplified, organized, or understandable way [9]. This process is illustrated in Figure 1.

![Figure 1: Example of a data science workload that parses a CSV file, transforms it, and writes its output to a new CSV file.](image)

While the goal is straightforward, the pipeline development process often requires data exploration, iteration, and substantial debugging. Data scientists must develop their pipelines to accept input data of a specific format. However, in practice, input data is not uniform. As the amount of data grows, errors and inconsistencies arise in datasets producing dirty data. Dirty data manifests itself in a variety of forms. Some entries may not contain information for every field resulting in null values, while other entries may not contain the correct number of fields. Even if an entry appears to conform to the expected schema, unexpected values may also cause exceptions to be thrown in UDFs. For example, this may occur when a value of zero is used for division or a non-numeric string is cast to an integer.

In some cases, exceptions may still be salvageable as valid output, and it is up to the data scientist to write pipelines that are accommodating to irregularities in the dataset. Creating a finalized pipeline is an inherently iterative design process. Data scientists have no way of knowing what exceptions will occur from their input data until the data is actually processed. Even if all exceptions are accounted for within a single dataset, many pipelines are designed
to be repeatedly run on new data sources over time, which opens the possibility for new classes of exceptions.

This results in a common design pattern for data scientists. After executing an initial query over a dataset, the data scientist observes what types of exceptions occurred. Then, they add custom resolving logic into their pipelines or UDFs in order to salvage some of the exceptions. This is an iterative process. Each time, the data scientist adds additional resolving logic to their pipeline until little to no exceptions remain when the entire dataset is processed. By doing so, the data scientist maximizes the amount of data that is included in the result of their query.

1.3 Problem Statement

The advantage of this approach is to utilize as much of the original dataset as possible. After developing an exception-resistant pipeline over the original dataset, the pipeline is more likely to generalize to new, unseen data sources as well. By developing robustness over exceptions, data scientists minimize wasted data entries. As data is such a valuable commodity, businesses and researchers are highly incentivized to utilize as much of their existing sources as possible [19].

However, there is a glaring problem with the efficiency of this approach. After the data scientist makes an initial pass over the input data, they now have a collection of both valid output rows and exceptions that occurred during execution. In an iterative workflow, each time resolving logic is added to the pipeline, the data scientist must reprocess the entire workload in order to see how many exceptions were resolved and whether any new exceptions have occurred. This introduces redundant processing for data that was able to execute successfully the first time on the initial unresolved pipeline. Since resolving logic should only affect the exception rows, any valid output from the previous execution will produce the same result as before. If exceptions are expected to only represent a small percentage of the overall input data, a vast majority of the dataset is being unnecessarily processed even if only a small number of exception rows would be affected by the change. At the scale of gigabytes or terabytes of data, the costs involved with reading, processing, and writing the redundant output become substantial. This issue ultimately poses the following question:

| Can data scientists reduce their development time by eliminating the unnecessary reprocessing that occurs when exceptions are resolved in an iterative workflow? |

1.4 Proposed Solution

In this paper, we present an implementation for efficient incremental exception resolution in Tuplex. The feature allows data scientists to write highly exception-resilient pipelines for their input data while reducing the cost, time, and computation power of the development
In order to improve efficiency, incremental exception resolution removes redundant processing that takes place between pipeline executions. Our key insight is that resolving logic only affects exception rows and not the vast majority of normal rows. By reusing the normal row result from a previous pipeline execution and only performing new work to resolve any exceptions that occurred, the newly resolved exceptions only have to be merged back into the original output and reported back to the data scientist. Thus, when resolving logic is added, it is no longer necessary to read the input data and do an initial processing pass since the result has already been computed previously.

The technical aspects of this system are broken down into three sequential stages. The first of these stages is results caching. When Tuplex executes a novel pipeline, it must cache necessary information about the pipeline and its result in order to allow for the possibility of incremental exception resolution down the line. The specific information that needs to be cached depends on the structure of the pipeline and user-specified options. The second stage is pipeline comparison. After Tuplex has cached the result of a previous pipeline’s execution, it must be able to detect when it is safe to reuse those results by determining whether or not only exceptions would be affected by this pipeline change. If it determines that incremental exception resolution is safe, it may proceed to the third stage. In this final stage, Tuplex must reprocess only the exceptions that were produced during the previous execution and merge the results with the previous execution’s normal output. An overview of this system is illustrated in Figure 2.

![Diagram](image-url)

**Figure 2**: Overview of the incremental exception resolution architecture and flowchart in Tuplex.

## 2 Background

Multiple projects have been developed that provide simple to use Python APIs for data scientists to process and analyze large quantities of data efficiently. While the exact implementations vary between projects, the frameworks employ a common strategy to bypass Python’s slow interpreter by either compiling or transpiling as much of a data scientist’s pipeline as possible. By serializing the input data to memory, processing it through a compiled code path, and deserializing it back into a Python object, these methods achieve significant
speedups over pure Python processing [13, 2]. Since frameworks perform this work in the background, the data scientist can develop and process their data analytics solely using Python without needing to worry about how the framework functions behind the scenes.

In practice, most frameworks are difficult to use. Due to Python’s dynamic typing and the limitations of type inferencing, some data analytics libraries require data scientists to input additional type annotations when developing their pipeline and UDFs [6]. Another issue is maintaining feature parity between third-party libraries and compiled code generation. While frameworks can support code compilation for some of the most popular libraries, it is infeasible to generalize compilation to all possible third-party imports. Thus, frameworks are further limited by the subset of Python and common libraries they can generate compiled code for. Finally, due to the possibility of heterogeneous input data and uncertain behavior that can arise within pipeline UDFs, frameworks must develop a strategy for dealing with exceptions that occur during runtime.

Tuplex stands out among data analytics frameworks in its ability to process malformed data and handle runtime errors that occur during pipeline execution. In this section, we compare how exception handling is implemented in other data analytics frameworks. One of the closest framework comparisons to Tuplex is Apache Spark, which is one of the most popular frameworks for big data analytics with over 1,800 contributors and 32,600 stars on GitHub [8]. Spark is a language-independent compute engine with APIs for many major languages including Python. Despite its widespread usage and substantial development effort, Spark fails to come close to Tuplex’s versatility in handling exceptions.

2.1 Schema Exceptions

Many frameworks use load-reject files to handle schema violations. When Spark encounters a schema violation when loading in a file, it dumps the row into a separate “bad records” file on disk [5]. Along with Spark, Vertica and PostgreSQL also implement this technique [20, 15]. A problem with this approach is that once the schema violations have been collected into a file, it is up to the data scientist to read in the file and process or resolve the exceptions with a different pipeline than the one that generated the exceptions originally.

In addition to its load-reject files, Spark allows data scientists to specify one of three behaviors to apply when it encounters a schema violation during CSV reading. The default is "Permissive" mode, which inserts null values for fields that do not parse. However, this approach creates an ambiguity problem with null values as there is no way to determine whether a field contained an exception or a true null value. "Drop Malformed" mode ignores the entire row if a field cannot be parsed. While this mode allows for normal rows to be processed, the ignored rows are never reported to the data scientist and are thus unable to be debugged and resolved. Finally, "Fail Fast" mode stops execution if it encounters any malformed data, which means that a single row can cause all previous computations to be wasted [7].
In comparison, given the same schema violation, Tuplex will automatically store the violating rows as internal exceptions and report them after the pipeline execution finishes. Data scientists may also add custom resolving logic to their pipelines in anticipation of what exceptions may occur. When this is specified, Tuplex will attempt to resolve any internal exceptions on a separate code path.

2.2 UDF Exceptions

Exceptions that are raised within UDFs during runtime have many of the same concerns with its implementation in Spark. By default, if a runtime exception occurs within a Spark UDF, the entire job will fail to execute. This results in the same loss of valid work as seen previously with “Fail Fast” mode. If the data scientist wishes to manually intervene, they can instead return a null value to the UDF when an exception occurs, which results in the same null value ambiguity [6].

In contrast, if an exception occurs within a UDF in Tuplex, it will ultimately be reported back to the data scientist and ignored in the final output. If the data scientist wishes, they can add resolving logic using Tuplex’s resolve and ignore operators without being forced to return an ambiguous null value. By not suspending execution by default, Tuplex enables developers to determine trends in exceptions that occur and work to resolve them efficiently.

2.3 Debugging

In addition to their performance concerns, these exception handling practices create a challenging debugging and user experience for data scientists. With load-reject files, frameworks do not group or organize their exceptions in any way. In addition to the added friction of loading the rejection data separately, data scientists are also not able to analyze patterns in the types of exceptions that occurred without performing additional analytics or sorting themselves. The way Spark handles exceptions that occur in UDFs is similarly difficult to debug. Because Spark suspends execution when a runtime exception is encountered, data scientists are only able to resolve and debug one exception at a time. When working with data at scale, this method is infeasible and hides any trends or commonalities that may be causing exceptions to arise. Until version 3.1 was released in April of 2021, Spark did not even report the reason for why a UDF exception occurred [12].

In Tuplex, all exceptions are aggregated and reported back to the user including information about what the exception type was and where within the pipeline it occurred. By grouping exceptions by these characteristics, developers have a better sense of where to start debugging and are able to target specific parts of their pipeline with resolving logic.

By comparing the exception capabilities and debugging experience of Tuplex with similar frameworks, it is clear that many of them lack the same level of flexibility that Tuplex has. Because of the way these frameworks handle exceptions, implementing an efficient
incremental exception resolution system would require overhauling a large portion of their underlying architecture and codebase.

3 Incremental Exception Resolution

In this section, we delve into incremental exception resolution’s architecture and algorithms. To implement, test, and benchmark this feature, we added over 5,000 lines of C++ and Python code to the existing codebase. As described in Section 1.4 and illustrated in Figure 2, the implementation can be broken down into three distinct steps: results caching, pipeline comparison, and merging. With these steps, Tuplex supports efficient incremental exception resolution by eliminating duplicate IO and pipeline processing work when it is safe and practical to do so.

3.1 Caching

The first step of incremental exception resolution is to cache the results of a pipeline after it has been executed. In order to make the development experience as streamlined and straightforward as possible, users should not have to specify whether or not they will be resolving exceptions before a pipeline is executed. After all, there is no way of reliably determining whether or not exceptions will be produced before an arbitrary pipeline is executed. Thus, Tuplex must cache the results of every pipeline execution that occurs in a single development session by default in order to reuse the results later down the line.

To start developing with Tuplex, a user first initializes a Context object, which is responsible for specifying the options and parameters that a pipeline will execute within. When initializing a Context, users may configure an assortment of options such as the number of executors and memory, optimization strategies, and other parameters. From this, users can build and generate multiple pipelines, which are represented as DataSet objects. A DataSet is a collection of logical operators such as file input, output, maps, filters, aggregates, joins, and others. By composing these operators and providing UDFs, users can create complex and powerful data analytics and transformation pipelines. Tuplex lazily evaluates each DataSet, which means that it defers execution until a user adds an output operator such as collect, cache, or tocsv, to its logical plan.

When a user enables incremental exception resolution by a Context option, Tuplex will always cache the output and results of executed pipelines. Tuplex stores this cache in the Context within an IncrementalCache object, which maps a pipeline in the form of its logical plan to the results that were produced by it being executed. Figure 3 illustrates the architecture for incremental exception resolution’s caching strategy.

The exact representation of the results will vary based on the structure of the pipeline and the user’s merging strategy, which is detailed later in Section 3.3. For some configurations, the cache will only need to store metadata about a pipeline’s output in order to support incremental exception resolution. However, certain configurations will require the cache to
store all of the output rows themselves. Depending on the size of the output data, this may cause the cache to exceed its ability to store results in main memory and will require the IncrementalCache to dump some of its stale results to file. Utilizing the shared nature of a Context among multiple DataSet’s, Tuplex is able to successfully cache the results that are produced by executing a pipeline for later use in incremental exception resolution.

### 3.2 Pipeline Comparison

Once the IncrementalCache has been populated with results from novel pipeline executions, the second step is to determine whether or not it is safe to reuse the results from a given cache entry for efficient incremental exception resolution. Whenever a user executes a pipeline, Tuplex compares the current pipeline with all of the pipelines that are currently stored in the IncrementalCache. If the current pipeline returns a match, Tuplex can reuse the results of the previous pipeline. However, if the current pipeline fails to find a match in the IncrementalCache, it must execute in its entirety and save its results.

Before describing the algorithm used to compute the comparison of two pipelines, we will first establish how pipelines are built by the user and internally represented. Utilizing the API operations provided by Tuplex, users create pipelines to load, transform, and output their data. Some operations, such as a map, require the user to provide a UDF in order to determine the behavior. When compiling the pipeline into LLVM bytecode, Tuplex must infer the input and output types of each logical operator and UDF in order to generate a valid code path.

The following code demonstrates a basic data transformation pipeline in Tuplex. The pipeline reads in data from the ‘animals.csv’ file and automatically detects the column and UDF types by sampling the input. It creates a new ‘title’ column by concatenating the ‘name’ and ‘species’ columns found in the dataset. The rows are then filtered according to the ‘age’, and finally, the pipeline outputs its result to a new ‘output.csv’ file.
Along with basic transformation operations, Tuplex allows users to add custom resolving logic to their pipeline without having to incorporate it directly into their UDFs. This separation of concerns allows Tuplex to process the normal case rows without needing to worry about the overhead that resolving logic may cause. If exceptions arise during execution of the normal case code path, Tuplex pools and reprocess them on a slower code path that incorporates the resolving logic. For example, the third row in the following dataset will cause a `TypeError` when executed through the pipeline as it will attempt to string-concatenate with a `None` value.

<table>
<thead>
<tr>
<th>name</th>
<th>species</th>
<th>age</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Buddy&quot;</td>
<td>&quot;dog&quot;</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>&quot;Yaz&quot;</td>
<td>&quot;cat&quot;</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>&quot;dog&quot;</td>
<td>2</td>
<td>Unknown the dog</td>
</tr>
<tr>
<td>&quot;Gordy&quot;</td>
<td>&quot;gerbil&quot;</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

In order to resolve this exception, users can add a `resolve` or `ignore` operator to their pipeline. An `ignore` operator will simply ignore any rows that throw a specified error, which will increase efficiency as it will not attempt to reprocess them in a slower code path. With a `resolve` operator, a user can provide an additional UDF to attempt to resolve a row that throws a specified error. For example, the user can resolve the `TypeError` with the following resolving logic:

The `resolve` operator will now allow the third row to be processed as normal resulting in a final output of the previous dataset written to `output.csv` as such:

<table>
<thead>
<tr>
<th>name</th>
<th>species</th>
<th>age</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Yaz&quot;</td>
<td>&quot;cat&quot;</td>
<td>8</td>
<td>&quot;Yaz the cat&quot;</td>
</tr>
<tr>
<td>None</td>
<td>&quot;dog&quot;</td>
<td>2</td>
<td>Unknown the dog</td>
</tr>
<tr>
<td>&quot;Gordy&quot;</td>
<td>&quot;gerbil&quot;</td>
<td>4</td>
<td>&quot;Gordy the gerbil&quot;</td>
</tr>
</tbody>
</table>

Internally, Tuplex computes a logical operator graph in order to represent the structure of a pipeline. Each logical operator has two main components: operator type and input and
output schema. When evaluating the validity of a pipeline, Tuplex requires that the output schema for one operator matches the input schema to the following operator. These schemas are inferred from the previous operator and return types of the provided UDFs. In practice, the graph is a tree with the output operator serving as the root that flows back to the data input operators. It is a tree rather than a simple list as join operators will produce a branching effect. Figure 4 illustrates examples of a logical plan graph.

Figure 4: Example of Tuplex code being translated into a tree of logical operators.

For incremental exception resolution, Tuplex computes a comparison of logical operator graphs to determine if the current pipeline matches a previous one that has been stored in the cache. At a high level, a pipeline comparison should only return as a match if the user has reprocessed a previous pipeline and added zero or more resolve and ignore operators to the current pipeline that are guaranteed to only affect exceptions that previously occurred. Tuplex does pipeline comparison in the logical optimizer before any data is processed. Thus, it is a lightweight check and adds negligible overhead to the framework.

When computing this comparison, Tuplex traverses the logical plans of the previous and current pipeline simultaneously in order to compare their corresponding logical operators to determine a match. Two logical operators are equal if and only if they have the same type, input schema, and output schema. As a special case, input and output operators are only equal if, in addition to these requirements, they also point to the same file path as determined by a string comparison. Figure 5 shows an illustration of this algorithm. There are four exhaustive cases to consider:

1. Neither the previous nor current pipelines have resolve/ignore operators.
2. Only the previous pipeline has resolve/ignore operators.
3. Only the current pipeline has resolve/ignore operators.

4. Both the previous and current pipelines have resolve/ignore operators.

Case 1

If neither pipeline has resolve or ignore operators, the comparison should only match if they are indeed the exact same pipeline. To achieve this behavior, Tuplex traverses both trees beginning with the file output operator on both. Tuplex confirms the equality of these operators before continuing on to the operator’s child. If an operator has two children due to a join, the algorithm can recur on both of the subtrees. This process will occur until it reaches the base of the tree, which will be the file input operator. Because all of the logical operators matched up, Tuplex considers the pipelines safe to reprocess. Conceptually, when no resolvers are added to a pipeline, incremental exception resolution should act as an automatic caching or memoization feature that simply returns the previously computed results without any additional work.

Case 2

If the previous pipeline contains resolve or ignore operators and the current pipeline does not, they are not compatible. Because resolving logic has been removed from the previous pipeline, the current pipeline will produce a subset of the results that were output in the previous pipeline. Conceptually, this is the opposite process of incremental exception resolution, which does not need to be supported by the implementation.

Case 3

If the current pipeline contains resolving logic and the previous pipeline does not, the pipelines are compatible if the current pipeline would be the same as the previous pipeline if all the resolving logic has been removed. To determine this, the algorithm can simply follow the same recursive process as the first case, but skip over any resolve or ignore operators that it encounters while traversing the current pipeline until it reaches a non-resolving operator to continue the comparison with the previous pipeline. This case represents a developer’s first iteration with incremental exception resolution as they first executed a pipeline and then added resolving logic to attempt to reduce the number of exception rows.

Case 4

When both the previous and current pipelines have resolving logic, the algorithm becomes a bit more complicated. This situation occurs past the first iteration of incremental exception resolution when the user decides to add additional logic to resolve more of the exceptions after their first attempt. A requirement of this case is that new resolving logic must occur later in the pipeline than any previous resolving logic. If this order is violated, the new logic
has the chance to affect the resolved exceptions from a previous incremental resolution step, which would not preserve the cached output. Thus, in this comparison, Tuplex compares the logical graph of both pipelines up until a resolver is encountered in the previous pipeline. During this stage, the algorithm acts as if it were the third case and skips over any resolvers that occur in the current pipeline. Once it reaches a resolver in the previous pipeline, it must switch to enforcing a strict comparison between the pipelines and check for one-to-one equality until it reaches the end of the tree.

Figure 5: Example of pipeline comparison performed over the animal pipeline. The top row is the initial pipeline and the bottom row represents the first incremental resolution step.

Tuplex performs this logical plan traversal due to the fact that resolving logic is separated from the normal case logic. In comparison, if users had to write resolving logic in the form of try/catch blocks directly into their UDFs, this comparison would require comparing arbitrary blocks of Python code to try and extract heuristics of equality. By generating a full logical operator graph of a pipeline and extracting key information such as input and output types, our work achieves this with a simple tree comparison algorithm.

However, this algorithm comes with some drawbacks and limitations. Because the equality check on logical operators only enforces schema matching, it is possible that the UDF functionality of a logical operator is not actually the same between pipelines when a match occurs. These false positives are mitigated by enforcing that the input and output data sources must be the same between matching pipelines. With further implementation and research efforts, this algorithm can be improved by traversing the ASTs of Python UDFs.

Although it is not a perfect system, utilizing some form of automatic pipeline comparison is necessary to streamline the development process for users. Without the need to worry about enabling and disabling incremental exception resolution manually, users can focus on the more important aspects of their code.
3.3 Merging

After a pipeline comparison has determined that it is safe to reuse the cached results of a previous execution, Tuplex can proceed with the efficient incremental exception resolution instead of processing the pipeline as normal. A normal execution in Tuplex requires the following steps to take place: read the input data, process normal case rows and pool together exceptions, resolve and merge exceptions back into the output, and write the results to disk. Incremental exception resolution skips the first two steps as this processing will remain the same due to the fact that resolvers do not affect the normal case rows. Instead, incremental Tuplex will jump right into the resolve and merge stage.

The first part of this process is simple: a new resolving code path is generated from the current and updated pipeline. Then, Tuplex retrieves the exceptions that occurred during the previous execution from the IncrementalCache and processes them through the new pipeline. Each of these exceptions will either be resolved into a valid output, ignored, or produce the same or a different exception. In the case that the exception is ignored, there is no more work to be done. Similarly, in the case that a new exception is produced, it is pooled together and cached for future incremental resolution steps. However, when Tuplex resolves some exceptions into a valid output, the next step is to merge these results with the valid output from the previous execution.

When merging the results of the current and previous pipeline, there are two cases: whether or not the data should be merged back into its original order from the input data. Both of these two modes are important for Tuplex to support. On the one hand, merging exceptions back into their original order is intuitive for users who are used to working with other libraries such as Pandas or Spark which preserve the order of data by default. However, maintaining the order of rows comes at a computational cost, which is demonstrated in experimental evaluation. Thus, Tuplex also allows users to disable merge in order functionality to improve the performance of their workloads. This case is justified further as some pipeline operators carry the assumption that Tuplex will not preserve order. For example, Tuplex implements a join operator using a hash table, which will destroy the order of the dataset. In order to restore this, Tuplex would need to compute an additional sort over the joined dataset, which negates the need to merge exceptions back into their original order. Tuplex automatically distinguishes between these cases and utilizes the efficiency of not merging in order when it is not necessary. Figure 6 illustrates an overview of this process.

Out-of-order

When Tuplex can merge data out-of-order, the implementation of incremental exception resolution is more straightforward. In order to detail how this is implemented, we first illustrate how Tuplex internally manages and processes data through its dual-mode execution framework.
Tuplex splits its input data into three distinct groups: normal rows, general rows, and fallback rows. Normal rows are the majority schema case as detected by a sampling of the input data. Depending on options that the user can configure, Tuplex can aggressively optimize for the common case in order to generate a code path that is able to process the majority of rows as fast as possible. For example, if a column of a dataset contains 90% integers and 10% null values, the common case may type the column as a non-null integer. Thus, a map operator that operates on the column does not need to worry about the null case as rows that contain a null value will never execute over the code path in the first place. When a row violates the normal schema, but may still conform to a less aggressive classification, it is considered a general case row. In the previous example, while the normal case may type the column as a non-null integer, the general case will type it as an optional integer and allow for null values. In Tuplex, normal case rows can always be upcast to the general case schema. Thus, general case rows are generated during data input when a row violates the normal schema or when a normal row throws an exception during processing in its optimized code path. General rows are then executed in a slower code path that contains code-generated logic from any resolve or ignore operators that are added to the pipeline. Fallback rows contain data that violate both the normal and the general case and are stored as pickled Python objects. In order to process these rows, Tuplex falls back to the Python interpreter, which is the slowest code path available. No matter which code path an input row is processed in, its valid output may be classified as a normal, general, or fallback row. If a row ultimately produces an exception, it is stored in a separate group of unresolved exception rows. Figure 7 illustrates an overview of this architecture.
Figure 7: Tuplex reads in input data and classifies it as a normal, general, or fallback case. After being processed through one or more code paths, Tuplex produces normal, general, fallback, and exception rows.

When the order does not matter, Tuplex simply concatenates the normal, general, and fallback rows together when writing its output or reporting it to the user. Without needing to perform an in-order merge of these three cases, this is the most efficient option for incremental processing. Likewise, incremental resolution can simply reprocess the cached exceptions from the previous pipeline execution into new normal, general, and fallback rows. Because these do not need to be merged in-order, incremental resolution can simply append these results to the file that contains the previous output or dump them to a new file in the same directory. In the example illustrated in Figure 8, an initial pipeline execution produced two output files, ‘output.part0.csv’ and ‘output.part1.csv’, but still had unresolved exceptions. After the user adds resolving logic, incremental exception resolution skips the file input step and fast path processing. Instead, it only has to retrieve the exceptions that have been saved in the IncrementalCache and process them through the new pipeline. Once Tuplex resolves the exceptions into new valid output, it can simply output them to a new ‘output.part2.csv’.

Figure 8: Illustration of how out-of-order incremental exception resolution reprocesses previous exceptions and outputs them to disk efficiently.
In addition to being able to skip the initial step of reading and processing the initial file, out-of-order also benefits from not having to write out the entire output each time it is executed. Instead Tuplex only writes to disk for the new rows that were produced from resolving some of the exceptions.

There are certain assumptions and resulting limitations that accompany this approach. The first assumption is that the output directory of the pipeline is not altered between iterations of incremental exception resolution. If a user renames or moves files between executions, the resulting output may not correspond with the correct naming system or could potentially overwrite existing data on disk if its naming conventions interfere. It also does not maintain uniform file sizes. Because the last output file of the initial execution may be smaller than other files, some users may want the new data to be appended to the previous instead of generating a new file. On the other hand, the current implementation allows the user to know exactly where the results of their previous execution began and avoids editing previously produced files. Moreover, the user is free to repartition their data at the end of their development process.

**In-order**

In-order merging presents a unique challenge for Tuplex due to its separation of data into normal, general, fallback, and exception cases. At the end of pipeline processing, Tuplex must be able to recover the original order of the input dataset even though rows are dispersed throughout three valid output groups. For incremental exception resolution, a second challenge arises as exceptions that were previously not included in a dataset’s valid output are later resolved and must be merged back into their original order. In this section, we will first describe how Tuplex manages merging in-order during a single iteration of pipeline execution and show how this algorithm was extended to support incremental exception resolution.

The naive approach to preserving order across multiple groups of rows is to assign a unique sequential number to each row as it is read into memory. This row number will persist throughout Tuplex’s pipeline execution. After Tuplex processes the data, it merges the normal, general, and fallback rows back together by comparing their indices in ascending order. In the case that some input rows are not included in the output set due to an exception occurring or a filter operation, the merge algorithm only needs to output the row with the next highest row index. In the case that multiple rows are produced by a single input row, Tuplex can copy the row index, apply it to all output rows, and rely on their implicit ordering to determine the final order.

While this solution is conceptually simple and computationally makes a single pass over the output data, it requires a linear amount of additional memory to store the index of each input row. Our key insight is that while Tuplex maintains separate groupings of rows, the majority of rows are classified as normal rows if schema violations in a dataset are relatively infrequent. Thus, Tuplex employs a space-saving mechanism to support in-order merging.
Instead of storing a row index for each normal row, Tuplex relies on their implicit ordering. With this system, Tuplex continues to store an index for general, fallback, and exception rows along with their serialized data which dictates how the row should be merged back into the final dataset. This is done in a hierarchical approach. For each fallback row, Tuplex stores the number of normal rows that come before it. For each general row, Tuplex stores the number of normal rows plus the number of fallback rows. For each exception row, Tuplex stores the number of normal rows plus fallback rows plus general rows. This leads to a merging algorithm that is linear in the size of the output data, but it does not use linear space in the size of the normal case rows. Figure 9 shows pseudocode for this merging process.

```python
def merge(normal_rows, fallback_rows, general_rows):
    merged_rows = []
    normal_emitted = 0
    fallback_emitted = 0

    for each (general_row, general_index) in general_rows:
        while normal_emitted + fallback_emitted < general_index:
            if normal_emitted < index of next fallback_row:
                append next normal_row to merged_rows
                update next normal_row, normal_emitted
            else:
                append next fallback_row to merged_rows
                update next fallback_row, fallback_emitted
        append general_row to merged_rows

    for each remaining (fallback_row, fallback_index) in fallback_rows:
        while normal_emitted < fallback_index:
            append next normal_row to merged_rows
            update next normal_row, normal_rows_emitted
        append fallback_row to merged_rows

    for each remaining normal_row in normal_rows:
        append normal_row to merged_rows

    return merged_rows
```

Figure 9: Pseudocode to implement the Tuplex in-order merge algorithm between normal, fallback, and general case rows.

Because Tuplex generates the row indices for general, fallback, and exception rows when it parses the input data, there is a special consideration that must be applied during fast path processing of the normal rows in order to preserve the correct indexing. After processing the normal rows, if no exceptions or filtering occurred, the normal case output will correspond one-to-one with the input. In this case, the exceptions generated for the other cases of rows are still valid. However, if even a single normal row results in an exception or is filtered,
all subsequent general, fallback, and exception rows will contain the wrong index. In the case that a normal row produces an exception, Tuplex defers updating these indices to the resolving code path when all three cases are processed regardless. However, if a normal row is filtered, there is no trace of its existence in the output set. Thus, while processing the normal case rows, Tuplex must also update the indices of fallback, general, and exceptions rows as they occur. The algorithm to implement this uses a running counter and pointer into the groups of non-normal rows so that its runtime is proportional to the number of these rows.

A similar indexing update algorithm was implemented to support incremental exception resolution. During incremental resolution, when an exception is resolved into a normal, general, or fallback case row, it is merged into its respective list in-order. However, there is now an additional row in the output that may not be accounted for in some of the saved indices. Because the groups are indexed hierarchically, an exception that is merged back into a normal row propagates up to affect the row indices of subsequent fallback, general, and exception rows. However, an exception that is merged back as a general row will only affect the row indices of the exceptions.

In-order merging presents a significant issue for incremental mode that is not encountered with out-of-order merging. This process involves merging data into the output of the previous pipeline’s execution. With no efficient method for inserting text into a simple CSV file, incremental exception resolution needs to store or load the entire output of the previous pipeline’s execution in memory. This is a stark contrast to out-of-order merging, which never needs to process or load information from previous executions. This increases the runtime cost of incremental exception resolution for two reasons. The first factor is that performing the in-order merge in memory takes more computation time as it performs a linear amount of work over the entire dataset. The second factor is that after merging in the results, it must rewrite the entire output to disk and overwrite the results from the previous execution. Figure 10 illustrates this process.

![Diagram](diagram.png)

Figure 10: After an in-order merge occurs, incremental exception resolution must overwrite existing data on the file system as it outputs the new results.
Because of this issue, we present commit mode, an alternative implementation of incremental exception resolution when in-order merging is required. With commit mode, the user is able to specify when they wish to commit the results of their pipeline execution to disk. By not committing their results until the end of an incremental exception resolution process, a user is able to bypass the cost of constantly rewriting the file output. The user can simply reprocess the data and add new resolvers as many times as they would like, and Tuplex will store the results in memory. When they are finished working on a pipeline, the user can finally commit their changes, and Tuplex will write out all of the data to disk. While this mode avoids the process of writing data to disk, it comes at the cost of requiring active input and foresight from the user when they are ready to finish their development process. The user is not able to walk away at any point without losing their progress as they must first dump their working data to a disk. Figure 11 illustrates an overview of commit mode in Tuplex.

![Initial pipeline execution](image)

![Incremental exception resolution](image)

Figure 11: Throughout the pipeline development process, commit mode avoids writing out data to disk until the final step as specified by the user.

## 4 Experimental Evaluation

### 4.1 Methodology

We quantitatively measured the performance of incremental exception resolution across two experiments. We compared incremental mode against the version of Tuplex that does not support incremental exception resolution referred to as plain mode. As discussed in Section 2, related data analytics frameworks do not have the functionality to run an equivalent experimental setup. Moreover, since Tuplex’s single pipeline execution speed already outperforms that of Spark and Dask, any efficiency improvement on Tuplex would only increase this speedup. Thus, we benchmarked our work against only Tuplex as a baseline.

We emulated an incremental exception resolution workflow in both experiments by first processing the input data over a pipeline without any resolving logic. Then, we processed...
data was iteratively in a series of incremental steps, each time adding an additional ignore or resolve operator to the pipeline.

We measured the comparative performance of incremental and plain modes of Tuplex using execution time. Specifically, we measured wall-clock time as opposed to CPU time in order to closely represent the experience of a data scientist engaging in an incremental exception resolution workflow. We calculated the total time savings between incremental mode and plain mode as the percentage of execution time saved over the course of an entire pipeline development workflow. This was the most important metric to examine when evaluating the effectiveness of incremental exception resolution. Because efficient exception resolution seeks to remove redundant computations from a data science workflow, we measured those computations as a metric of time saved. The time correlates directly with money saved in the case of running experiments with cloud computing such as an EC2 server and energy saved in the case of the power used to run computations.

### 4.2 Zillow

**Setup**

Our first experiment emulated a real-world use case of incremental exception resolution. We used a dataset and pipeline that had been previously created by Spiegelberg et al. for the original Tuplex paper [18]. In their experiment, they scraped data from Zillow, an online real estate marketplace [16]. The dataset consisted of 38,571 housing entries that included information such as title, location, price, and specifications for each listing. For our experiment, we scaled up the dataset from 7.4 MB to 100 GB by replicating all of the entries. Because we replicated the entire dataset, the proportion of malformed entries remained the same across the original dataset and the scaled-up version.

<table>
<thead>
<tr>
<th>title</th>
<th>address</th>
<th>facts and features</th>
<th>real estate provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>House For Sale</td>
<td>7 Parker St</td>
<td>3 bds, 1 ba, 1,560 sqft</td>
<td>J. Mulkerin Realty</td>
</tr>
<tr>
<td>SOLD: $717,000</td>
<td>32 Adelaide St</td>
<td>-- bds, -- ba, -- sqft</td>
<td>NaN</td>
</tr>
<tr>
<td>Apartment For Rent</td>
<td>Brainerd Rd</td>
<td>Studio, 1 ba, 516 sqft</td>
<td>NaN</td>
</tr>
<tr>
<td>House For Sale</td>
<td>11 Melo Rd</td>
<td>3 bds, 1 ba, 988 sqft</td>
<td>Lawton Real Estate, Inc</td>
</tr>
<tr>
<td>Apartment For Sale</td>
<td>13 Buckman St</td>
<td>6 bds, 2 ba, 2,581 sqft</td>
<td>J. Mulkerin Realty</td>
</tr>
</tbody>
</table>

Figure 12: Sample of select rows and columns from the scraped Zillow real estate dataset. When run over the Zillow pipeline, rows 1 and 2 produce exceptions. Row 1 can be resolved to a 1 bedroom.

Figure 12 shows a sample of this dataset with some of the columns purposefully omitted. The data science pipeline that we used works to clean, filter, and extract information from this dataset. For example, one of the operations extracts the number of bedrooms each listing
has by searching for a string that matches "X bds" in the "facts and features" column. For rows 0, 3, and 4, this is accomplished without failure. On the other hand, the "facts and features" in rows 1 and 2 do not match the expected bedroom pattern. When run through the initial pipeline, these rows will throw an exception as the bedroom string cannot be found in the column. After we processed the entire dataset through the Zillow pipeline, 24.98% of the rows were set aside as exceptions and only 75.02% of the original dataset were fully processed and output.

In this experiment, we emulate a data scientist’s workflow by incrementally resolving or ignoring a portion of these exceptions until no exceptions remain when the dataset is run through the pipeline. For example, in the first incremental step, we add a resolve for bedroom extraction. As seen in row 2, some of the entries in the dataset list a "Studio" instead of the number of bedrooms. In order to fix this exception, we add a resolver to the pipeline that searches for the string "Studio" in the row’s "facts and features" column and returns 1 for the number of bedrooms. In the sample dataset above, the pipeline will now process row 2 successfully while row 1 will still throw an exception. This is considered a single step of incremental resolution. As a next step, we add an ignore operator after the previous resolver as row 1 has no way of extracting bedroom information, so it should be removed entirely from further processing. In total, the experiment runs the entire pipeline seven times. The first time there are no resolvers, and each subsequent time we add a single resolve or ignore operator to affect some of the exception rows. By the end of the incremental exception workflow, the pipeline resolves or ignores all of the exceptions. Table 1 lists a complete breakdown of the number of exceptions processed, resolved, and ignored in each stage of the experiment. During the initial execution of the pipeline when no exceptions are resolved, the pipeline writes 2.63 gigabytes of data to disk, which is a significant decrease from the initial 100 gigabytes of input data due to filtering that occurs within the pipeline. By the last step when all of the exceptions have been either resolved or ignored, this increases slightly to 2.67 gigabytes.

<table>
<thead>
<tr>
<th>Step</th>
<th>Resolved</th>
<th>Ignored</th>
<th>Unresolved</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>140,680,635</td>
</tr>
<tr>
<td>1</td>
<td>10,687,932</td>
<td>0</td>
<td>129,992,703</td>
</tr>
<tr>
<td>2</td>
<td>10,687,932</td>
<td>50,023,026</td>
<td>79,969,677</td>
</tr>
<tr>
<td>3</td>
<td>10,687,932</td>
<td>50,066,829</td>
<td>79,925,874</td>
</tr>
<tr>
<td>4</td>
<td>10,687,932</td>
<td>56,184,648</td>
<td>73,808,055</td>
</tr>
<tr>
<td>5</td>
<td>10,687,932</td>
<td>129,875,895</td>
<td>116,808</td>
</tr>
<tr>
<td>6</td>
<td>10,804,740</td>
<td>129,875,895</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Number of exceptions that occur in each step of the Zillow pipeline.

We ran each experimental condition on a multi-core machine using 64 executors with 6 GB of memory each. We ran the experiment over 10 trials with the results averaged uniformly.
across all of them. In order to most closely model a real-world workflow, we cleared the OS buffer cache before each trial of the experiment, but it remained usable throughout all steps of exception resolution.

**Out-of-Order Experiment**

In the first condition, we compared the execution time of plain and incremental modes when exceptions could be merged out-of-order. Figure 13 graphs the end-to-end execution times for each step of the pipeline. During the first step of pipeline execution, both plain and incremental modes executed in the same amount of time in just under 200 seconds. This is an order of magnitude higher than any subsequent step for both plain and incremental modes, which demonstrated a gradual reduction in execution time between steps 1 through 6. While both showed a reduction in execution time, incremental mode was consistently faster than plain mode. In total, plain mode took 358.31 seconds to complete all 7 steps of pipeline execution while incremental mode took 225.51 seconds. Thus, after processing the same workload, incremental mode resulted in a 37.06% time savings over plain mode.

![Figure 13: Total execution time of out-of-order plain and incremental Tuplex running an incremental exception resolution workflow over 100 GB of Zillow data.](image)

We broke down the execution time for both plain and incremental modes into three stages of computation that make up the bulk of the processing time. The first of these stages is fast path processing. This involves reading in the input data, processing all normal case rows on the fast path, and emitting any exceptions that occur due to malformed data or runtime errors thrown in Python UDFs. While both plain and incremental modes must perform this step on the first pipeline execution step, incremental mode is able to skip this step in future iterations as it does not need to reprocess the normal rows. The second stage is slow path processing. This involves processing the exceptions that occurred in the fast path over the generated resolve code path or Python interpreter as a last resort. Both plain and incremental modes
must complete this step. The final stage is file output. While plain mode must write out the entire output after each execution, incremental mode only writes out the newly resolved exception rows to disk. Figure 14 graphs this breakdown for out-of-order merging.

From the graph, both modes showed an identical time breakdown for step 0 of the pipeline with the fast path processing accounting for around 190 seconds, slow path accounting for 0 seconds, and write times accounting for 4 seconds. During steps 1 through 6 of the experiment, plain and incremental modes showed differing results. For plain mode, the total time it took it to process the fast path decreased significantly to between 15 and 20 seconds in comparison to step 0 with a gradual decrease over each additional step. The time total time that plain mode spent writing the data to disk increased to around 8 seconds and remained fairly constant over the remaining steps. Finally, the total time that plain mode spent in the slow path increased from 0 to 5 seconds between steps 0 and 1 and decreased rapidly over the remaining steps to around 1 second. For incremental mode, the fast path and write times during steps 1 through 6 decreased to negligible levels. Instead, the entire execution time consisted of slow path processing as incremental mode only needs to reprocess exception rows.

The results of this experiment support our expectations about how incremental mode would compare to plain mode based on the implementation of both. Both modes produced the same total execution and time breakdowns in step 0 of the experiment. Because this is the first time a pipeline has been executed and contains no resolvers, incremental mode is expected to perform the same work as plain mode as it requires an initial result to be produced. The drop in total execution time by plain mode is at first unintuitive as once a resolve operator has been added, more work has to be performed to attempt to resolve the exception rows in the slow path. However, as seen in the time breakdown, this additional slow path cost is offset by a reduced fast path cost. The reason that the fast path shows a significant reduction in execution time is because, by the second iteration, the input data has already
been loaded into the file cache for efficient access. Thus, as the fast path must first read in and parse the input data, the total execution time is significantly reduced. Furthermore, both modes show a reduction in slow path execution time between steps 1 and 6 of the experiment. The execution speed of the slow path scales with the number of exceptions that it must process. Because steps 2, 3, 4, and 6 contain resolve operators, there is a significant reduction in the number of exceptions that are pooled for slow path processing in comparison to steps 1 and 5. Finally, incremental mode spends an insignificant amount of time writing in comparison to plain mode during steps 1 through 6. This is because while plain mode has to output the entire result set each iteration, incremental mode only writes out new data from the exceptions it resolved. Ultimately, the total time savings demonstrated by incremental mode in this experiment can be attributed to avoiding the costs of fast path processing and write times.

**In-order Experiment**

In the second condition, we compared the execution time of plain and incremental modes when exceptions had to be merged back in their original order. In this experiment, we compared three modes against each other: plain, incremental, and commit. With commit mode, Tuplex does not write data until the last step of the pipeline execution and instead stored in cached memory. Because incremental mode must merge its exceptions back in-order, it must rewrite the entire output each time an iteration occurs. On the other hand, commit mode writes them out once at the end.

![Figure 15](image)

Figure 15: Total execution time of in-order plain and incremental Tuplex running an incremental exception resolution workflow over 100 GB of Zillow data.

Figure 15 graphs the end-to-end execution times for each step of the pipeline comparing plain, incremental, and commit modes. In total, plain mode took 356.39 seconds to complete all 7 steps of the experiment. Incremental mode took 271.76 seconds resulting in a 23.75%
savings by processing the same workload. Commit mode took 229.70 seconds resulting in a 35.55% savings by processing the same workload. In step 0, both plain and incremental modes take the same amount of time, while commit takes slightly less. This is because both plain and incremental modes have the additional cost of writing their output to disk whereas commit mode does not have to write any output. Figure 16 shows the total breakdown of how each mode spent its execution time. Commit mode performs significantly better than incremental mode as its computation is no longer proportional to the size of the entire input data. There is a single fixed cost of outputting the data during the last step of the pipeline.

![Figure 16: Time breakdown of in-order Tuplex running an incremental exception resolution workflow over 100 GB of Zillow data.](image)

Figure 16 presents a similar story is presented in out-of-order mode. Both incremental and commit modes gain speedups over the plain execution by not processing exceptions through the fast path. Commit mode goes further than this by also not writing out its data until the final step of the pipeline.

### 4.3 Synthetic

**Setup**

In addition to the Zillow experiment, we tested incremental exception resolution on a set of synthetic datasets. Our motivation for this experiment was to explore how incremental
exception resolution trends when the proportion of exceptions in the dataset varies. In the Zillow experiment, the dataset produced around 25% exception rows in the initial pipeline execution, but how would the results change if there were 50% or 100% exceptions?

To answer this question, we generated 11 different synthetic datasets each containing 10 gigabytes of data. Each dataset contained the same number of total rows, but a variable proportion of them were exception rows. The number of exceptions ranged from 0%, 10%, 20%, and continuing on up to 100%. A row was 200 bytes in size totaling 50 million rows in each dataset. We designated normal rows by a 1 in the first column while exception rows contained a 0 in the first column. We filled the rest of the row with single-byte characters until the 200 byte row size was reached.

We executed a simple and cheap pipeline over the dataset, which created a new column by taking the inverse of the value in the first column. This operation would succeed for all rows that contained a 1 but would throw a ZeroDivisionError for the exception rows that contained a 0. Then, in a single resolving step, the pipeline would resolve all of the exceptions to the value 1. The metric we used to measure this experiment was the total execution time that it took to complete both the initial pass over the data and a single resolving step. The one deviation from this setup is for the dataset that contained no exceptions. Since there was no work to be performed in a resolving step, the experiment executed this pipeline only a single time.

**Out-of-order Experiment**

The first condition that we tested compared both incremental and plain Tuplex when exceptions could be merged out-of-order. Figure 17 graphs the results of this experiment.

![Figure 17: Synthetic experiment results for out of order merging.](image)
From the results, plain mode started out with a fast execution time of 27.33 seconds while processing the dataset with no exceptions. From here, there was a large jump for the 10% exception case, which took 53.19 seconds. Finally, this time declined gradually up until 90% exceptions, which took 45.24 seconds.

As the amount of exceptions increases, there are two counteracting forces that affect the execution speed of the pipeline. The first of these is that with more exceptions, there is more work that has to be done in the resolution step to resolve a larger number of rows. In this case, we would expect the execution time to increase as the number of exceptions does. The second factor is that as the number of exceptions increases, the total amount of rows that are written by plain mode should decrease. When 10% of the dataset contains exceptions, plain mode will first process the entire dataset and write out 90% of the rows to disk. During the incremental step, plain mode will again process the entire dataset but this time write out the full 100% of exceptions. In total, given 10% exceptions, plain mode must write out 190% of the dataset. Now when there are 20% exceptions, plain mode must first write out 80% on the first step of the pipeline and then write out the full 100% on the second for a total of 180% of exceptions written. This trend continues as the number of exceptions increases.

Finally, there is a significant drop in execution time between 90% exceptions and 100% exceptions. This decrease is slightly misleading and is possibly an artifact of the experimental design. The large decrease is due to the fact that on the first iteration of the dataset, plain mode writes out a total of 0 rows. On the next iteration, it then writes out the 100% rows. Given the linear decline pattern that persisted between 10% and 90% exceptions, we would initially expect the same slope to apply to the 100% exceptions. However, because the writing only happens a single time, plain Tuplex is able to fully write out its data to the buffer cache. On the other hand, when there are 90% exceptions, plain Tuplex first writes out 10% of the data, and when it writes out the full 100% the subsequent time, this causes the buffer cache to overflow as it pushes it over the edge causing significant slowdowns in write speed. The result could change as the OS buffer cache increases or decreases in size across machines.

The analysis of incremental mode is simpler. With no exceptions present, it does the exact same work as plain mode so is expected to have the same total time. As the number of exceptions increases, incremental mode has to do a linear amount of extra work to process the new exceptions that occur in the incremental resolution step. The reason that incremental Tuplex does not suffer from the same write problems as plain Tuplex is that it only writes out 100% of the data no matter how many exceptions are present in the dataset. When incremental mode executes over the 10% exception dataset, the first iteration is able to write 90% of the rows to disk. However, during the incremental step, because it can dump the new rows to a separate file, it only outputs 10% of the data that it resolved for a total of 100% output over both executions. Because this does not fill up the write buffer, it benefits from fast write speeds.

In this experiment, the cost of outputting data past the capacity of the buffer cache heavily penalizes plain mode in comparison to incremental mode. However, the purpose of
this experiment is to confirm expectations rather than compare a performance benchmark between the two modes. With a cheap pipeline and resolve step, the difference IO costs will dominate the outcome. However, this experiment supports the idea that incremental mode remains effective no matter how many exceptions are present in the output. As the number of exceptions increases, incremental mode only performs a linear amount of extra work proportional to the number of exceptions it must process.

In-order Experiment

The second condition that we tested compared plain, incremental, and commit modes when exceptions had to be merged back into their original order. Figure 18 graphs the results of this experiment.

![In Order 1 Synthetic](image)

Figure 18: Synthetic experiment results for in-order merging.

From the results of this condition, both plain and incremental modes produce almost the exact same results. Both start out with low execution times of around 27 seconds before a large jump when exceptions are added into the dataset and a linear increase over time. The linear increase in this experiment can be explained by the resolve step time now overpowering the file write times. The reason that incremental and plain modes are so similar is because the fast path is so cheap to compute due to the file already being in memory in the incremental step that the costs of resolving and merging exceptions and writing the data out far exceed this fast path savings.

Commit mode follows the same trend as out-of-order incremental mode as expected as well. A linear increase as the amount of work done to resolve and merge exceptions increases. When exceptions are present, commit mode performs faster than incremental or plain mode due to the lack of write times. Moreover, the slope of all three modes is greater than the out-of-order results because the cost is higher to process exceptions due to merging
them back in-order. Overall, this experiment again confirms expectations about how plain and incremental Tuplex work. When the fast path is a negligible cost, it makes sense that incremental merge in-order mode would have negligible gains over plain Tuplex. More so, it makes sense that the gains commit would have are noticeable when writing output times are significant.

5 Discussion

From the experimental results, incremental exception resolution has demonstrated significant speedups over the previous version of Tuplex. While in practice, the amount of time saved will depend on many factors including IO costs or the proportion of exceptions in the dataset, it is clear that by not having to reprocess unnecessary work, data scientists can speed up their development time. Furthermore, incremental exception resolution does not add significant development or workflow overhead. In most cases, it is able to work seamlessly in the background without the developer’s knowledge to speed up their development process. The one exception to this is the use of commit mode, which depending on the implementation, may force developers to consciously commit their results at the end of incremental exception resolution.

Time saved working on data science pipelines has broad implications for both data scientists, their employers, and the environment. For data scientists, time spent executing queries and developing robust pipelines cost money. Whether that be because Tuplex is running on an EC2 instance that is charged as a function of up-time or future versions of Tuplex that are running on AWS Lambda where each query has an associated cost that depends on the amount of work that is performed. These costs add up significantly over time and developers must be conscious of it when doing their job. While some data scientists may not be fronting the bill for cloud computing, their employers are instead. By reducing everyday expenses for data science teams, employers will have more funds to distribute to other aspects of the data science development process whether that be data collection, hiring more engineers, or conducting novel research. All of these present useful tasks to spend time and money on, whereas unnecessary recomputation does not.

In addition to monetary costs, time spent processing data has a real-world environmental impact due to the power consumption necessary to run the CPUs. Despite sustainability initiatives across the data science industry, businesses have yet to reckon with the immense environmental footprint that cloud computing has due to their use of non-renewable energy [14] [1]. While this problem is systemic in nature, it is one that must be addressed in data science research in order to effect meaningful change. By reducing the time spent during the development process, Tuplex helps reduce the time spent computing large amounts of data to when it is actually necessary to perform those computations. By eliminating this work behind the scenes, Tuplex can passively work to reduce energy consumption across the board.
5.1 Further Research

While this paper seeks to introduce incremental exception resolution into Tuplex and as a model for other systems, there is further research that can be applied to the topic in order to better understand the impact of incremental exception resolution, test its feasibility with data scientists, and create a more robust system. The first area of further research that can be explored is research into the user interface and user experience of incremental Tuplex. In order to be adopted by real-world data scientists, it is important to create a system that is intuitive, easy to use, and seamlessly integrates into existing use cases and frameworks. In order to expand on this research, experiments can be run by collecting input from real data scientists on various APIs that can be used to represent incremental Tuplex. One of the largest areas of ambiguity is how to implement commit mode into the system. By enabling commit mode, developers must know exactly when they are ready to walk away from their computer and must save the progress they have made so far. But this system is imperfect. How can Tuplex handle system failures, power outages, or file system changes that occur in the middle of incremental resolution with commit mode turned on? Further implementation would need to be applied in order to create a fault-tolerant system. Perhaps the user wants to save their progress for the day without actually having to write out the full data to disk. Tuplex should have a way of dumping the serialized bytes to memory instead of writing out the entire dataset in the middle of incremental exception resolution.

The next area of further research that can be explored is adding additional features and support for incremental mode with the full capabilities of Tuplex. Currently, incremental resolution is not supported for join and aggregate operators in Tuplex. While this can be accomplished, there are many nuances that arise with implementing this in the most efficient possible way. For example, with join operators, Tuplex no longer operates as a single execution stage. Instead, there are now two stages that contain multiple data input sources. It is no longer sufficient to only reprocess the exceptions that occur in both of the datasets because those exceptions need to be joined with the normal rows as well. Thus, an efficient system may want to reprocess the smaller of the two normal case rows. This along with other considerations should be explored in order to fully implement an efficient join operator in Tuplex.

Finally, there is ongoing research to implement Tuplex on a distributed system of AWS Lambda. An implementation of incremental exception resolution on a distributed system should be researched in order to observe the effects of extreme parallelism as well as cost savings in comparison to cost savings on a server like an EC2 instance.

6 Conclusion

In the race to claim the title of fastest data analytics framework, related projects are commonly focused on a single, simple narrative: how fast their framework can execute a single workload or query in comparison to an equivalent workload in another framework. While certainly an
important metric to consider, this benchmark does not tell the full story. Data scientists do not spend their time waiting for polished, optimized queries to finish executing. After all, if they have already been deployed, it will execute automatically or it is only ever used once. Rather, the time sink for data scientists is the journey to developing that finished pipeline. In order to overcome failures arising from dirty data and other scaling issues, data scientists work incrementally and slowly scale up their workloads until they are at the required size. The time spent during development adds up and is not necessarily apparent when considering a single pipeline’s execution speed- even if that contributes to part of the story.

Many factors contribute to the time it takes data scientists to develop their pipelines. The first of these factors is usability and user experience. How easily does a framework allow a data scientist to debug failures? How are exceptions reported? In this paper, we demonstrate that Tuplex provides a uniquely easy to debug experience when it comes to pipeline exceptions in comparison to other exception strategies from related frameworks. While not measured quantitatively, this inevitably will impact the development time for data scientists.

While many frameworks work to optimize the efficiency of a single query, in this paper we present an optimization over the pipeline development process itself: incremental exception resolution. In a real-world experiment, we demonstrate the capabilities of this to reduce pipeline development time no matter how many exceptions exist in a dataset. By speeding up the development process, Tuplex helps data scientists save time, money, and reduce their environmental impact at scale.
References


