Translating Batch Processing to Stream Processing in Big Data Analytics

Srikanth Badavath
bsrikanth@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Mokshitha Mandadi
mokshitham@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Neelesh Samptur
nsamptur@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Abstract

In the recent past there has been a shift towards stream processing as there is a need to process the massive amount of data being generated in today's world in real-time. Many legacy systems and scripts were developed that are only capable of traditional batch processing and extending the ability to deal with real time data requires rewriting the scripts and this can be cumbersome. Hence there is a need to automate this process. We present a way to automate this process through rule based conversions leveraging Abstract Syntax Trees (AST). The AST converter takes in a batch processing pipeline as input and transforms it into its equivalent stream processing code. We explore multiple operations including filtering and aggregation and also extend support to multiple forms of data including dataframes(CSVs) and RDDs.

Keywords

Big Data, PySpark, Stream Processing, Batch Processing, Abstract Syntax Tree, AST Transformation

1 INTRODUCTION

In the modern era of big data where there is an explosion in the amount of data that is being generated, there is a need to process this data efficiently and cost effectively. This gave rise to Spark, a tool that made use of commodity hardware to process huge amounts of data at a lower cost. Initially spark operated using traditional batch processing which can handle huge datasets, but was restricted to static data. With IoT devices generating data every second and applications like fraud detection, live monitoring and recommendation systems demanding instant insights, there is a need to shift to the real time processing paradigm. Hence there is a shift towards stream processing in the recent times due its ability to process real time data in the form of micro-batches. Despite this shift, there are many existing pipelines that have already been written to support batch frameworks. Extending these scripts to support streaming requires manually rewriting the code which is error-prone and time-consuming. It requires changes to data sources and processing logic. Though it is not possible to completely automate this process, a significant portion of this must be automated. We have explored Abstract Syntax Trees(AST) to do the same. ASTs are generally used to represent the essential syntactic elements of the source code in a structural and hierarchical representation. Compilers leverage ASTs for semantic analysis and code optimizations. We leverage this structured tree that serves as an excellent intermediate representation for code refactoring and transformations. This project introduces an automated way to transform PySpark batch processing code into stream processing code using Abstract Syntax Trees (AST). Our conversion script processes batch code that consumes data frames and RDDs and applies simple operations like filtering and aggregation. We keep the inherent logic as is and only make changes to parts of the code that explicitly require the use of the streaming library from spark. We kept this project within the scope of the course, but additional operations can also be explored using this approach. This report covers everything from our methodology, including our initial batch script, the ast conversions and the generated stream script in detail. We also talk about the experimental results and shed light on the limitations of this approach. We finally highlight the valuable contributions of each of our team members.

2 RELATED WORK

Apache Spark [1] provides both batch (RDDs, DataFrames) and streaming APIs (Structured Streaming, StreamingContext), but developers have to manually transition between code paradigms. Tools such as Apache Kafka [2] and Apache Flink [3] focus on native streaming, but they still require significant adjustments when switching from batch-oriented pipelines.

Existing solutions, such as Spark's unified APIs, allow for some batch code reuse in streaming scenarios, but they still require manual tweaks to manage input sources, execution semantics, and output sinks. Although AST-based transformations [4] have been explored in Python for code optimization and migration , their suitability for large-scale data processing pipeline conversion is still unexplored.

The use of Abstract Syntax Tree (AST) rewriting for program transformation and optimization was initially formalized in one of the first foundational papers, Code Transformation by Direct Transformation of ASTs [5]. Our solution is inspired by this concept and rewrites the syntax trees into similar streaming pipelines, finds important RDD operations, and parses batch-oriented PySpark RDD code into ASTs. Our converter guarantees structural soundness, semantic consistency, and dependability of the migrated streaming code by working directly on the AST instead of textually altering the code.

3 METHODOLOGY

Our system comprises three components: a batch processing script, a streaming script, and an AST-based converter. We use PySpark RDDs for data processing and Python's AST library for code transformation.

3.1 Batch Processing

The batch script processes a CSV file using PySpark RDDs by reading it as a text RDD, splitting lines by commas, filtering rows where the third column exceeds 50, mapping to

key-value pairs with the second column as key and 1 as value, aggregating counts by key, and performing a final aggregation to summarize total counts across all keys before saving results to an output file. The CSV input typically contains structured data with columns representing attributes like user ID, transaction amount, and timestamp, where the third column (transaction amount) is used for filtering high-value transactions. For example, a transformation might convert a row "123,50.00,2025-05-01" into a key-value pair (123, 1) if the amount exceeds 50, eventually aggregating to count transactions per user ID, followed by a global aggregation of total counts. This pipeline is deterministic, processing the entire dataset at once and producing a single output file. We have used the following rule-based operators such as (filter, map, reduceBy, aggregation (agg)). These operators define the logic for each step: filter selects rows based on a threshold, map creates key-value pairs, reduceBy sums values per key, and aggregation (agg) computes a global total across all keys. We chose RDDs over DataFrames for this pipeline because RDDs offer fine-grained control over transformations, which is crucial for educational purposes and for understanding the underlying mechanics of Spark's data processing, although DataFrames might offer better performance optimizations in production settings.

3.2 Stream Processing

The streaming script generated by the converter changes the batch workflow into streaming workflow using a specified folder. In this example we used StreamingContext with a 5 second batch interval and CSV files as an incremented source. The script takes advantage of textFileStream to watch the directory, wraps transformations in a foreachRDD, prints output to the console using foreach, and starts the streaming context using ssc.start() and waits for termination. The overall reasoning behind choosing the 5 second interval was to find some balance between real-time response time and execution overhead. By making the micro-batches small enough, we can achieve timely output and processing times, while also large enough to not incur too much execution overhead with all of the context switching. A better example of the idea of a micro-batch would be to think about the micro-batch processing a file that contains 100 rows, using the same transformations that we applied earlier for the batch workflow (i.e. filtering for transactions > 50, count aggregations, etc.), only we are progressively doing it across individual micro-batches.We have made use of the following rule-based operators (filter, map, reduceBy, aggregation (agg)) to provide incremental processing through implementation of the logic behind each step: apply filter to select rows, apply map to create key-value pairs, reduceBy prepares aggregated data by key, and aggregation (agg) calculates grand totals on combined micro-batches across the window. Late data follows one of the more significant challenges in streaming, such as network delays associated with the time required for data to be delivered. While we relied on fault tolerance built-into Spark, we capitalized that; the files were assumed to be added in controlled manner to the directory under simulation.

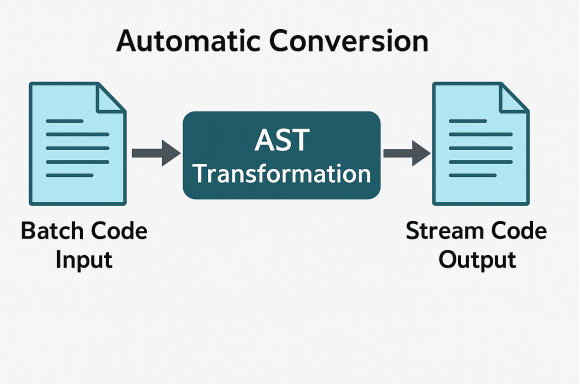

Figure 1: Overview of automatic AST-based code transformation from batch to streaming.

3.3 AST-Based Conversion

The converter achieves the conversion by manipulating the AST of the batch script file. It assembles an AST through reading the script then transforms the nodes using the BatchToStreamTransformer which adds a streaming import, inserts a streaming context setup after the spark context initiation, transforms textFile to textFileStream, wraps the transformation chain in foreachRDD to pass output to print, and switches saveAsTextFile commands to use start and awaitTermination commands. We have used the following rule based operators (filter, map, reduceBy, aggregation (agg)), to ensure that the conversion retained the transformation logic so that the streaming script reflects the batch logic. Some of the specific transformations we have made are to make changes to the function calls, for example to modify sc.textFile to ssc.textFileStream, and also the transformation chains have been wrapped in lambda functions for foreachRDD so that the streaming script reflects the batch logic.The transformer incorporates basic error handling, checks for expected variables like sc, and warnings are sent to the user whenever the structure of the script deviates from the one expected. At the end of the input, the transformer is converting the altered AST to Python and writing the streaming script. The transformer accounts for sequential orders of transformation and makes rule-based transformations to generate equivalent outputs for both batch and streaming executions as indicated in the overview diagram.

The process gives a structured method of code transformation, but the AST method is limited in several of the same ways as the previous example, including handling dynamic code or scripts that have many dependencies. We avoided these limitations by only focusing on a limited-specific subset of PySpark operations.

3.4 Additional Exploration

We have explored the transformation of PySpark RDD-based batch processing code into Kafka-based structured streaming pipelines (`rdd_to_stream` branch) by statically analyzing and rewriting Python source code using Abstract Syntax Trees (ASTs). This uses Python's built-in ast module to parse the input batch script into an Abstract Syntax Tree. This tree

structure allows safe inspection and modification of the code while preserving syntax and semantics.

AST Traversal and Pattern Recognition: The core transformation is performed by the RDDToKafkaStreamTransformer, a subclass of ast.NodeTransformer. This transformer visits all assignment (Assign) and expression (Expr) nodes to detect RDD-style operations. The tool identifies commonly used RDD operations (textFile(), flatMap(), map(), reduceByKey(), filter(),collect(), show()), for each of these, the transformer logs the operation and prepares an equivalent streaming-based transformation.

Operation Chaining Resolution: To support the PySpark code where multiple RDD operations are chained (e.g., rdd. flatMap(...). map(...). reduceByKey(...)), the tool recursively unpacks the chain in reverse using a custom process chain function. This function collects all transformation operations in order, matches lambda expressions for known patterns. And each transformation is then converted into valid PySpark streaming code with meaningful variable names.

Once the core transformations are complete, the converter constructs a Kafka-based streaming header that initializes a SparkSession configured for Kafka and defines the streaming input. The rewritten AST is then compiled back to Python source code using astor.to source and written to the output file. The tool ensures that unsupported transformations are noted with inline comments. Collect-based actions are replaced with writeStream sinks.

4 EVALUATION

We assessed the system using three metrics: correctness, performance, and usability, while also identifying limitations for future improvement.

4.1 Correctness

We assessed correctness by observing the output of the batch and the stream scripts using the same test datasets with the same 1000 rows to compare outputs. The batch output was the same as the cumulative streaming outputs once both input datasets were processed with the streaming script. For example, both datasets have the same keys, which will have had the same counts, thus confirming intent throughout the filtering and aggregation transformation. This evaluation process has a visual supporting case from the micro-batching streaming diagram in Figure 2.

4.2 Performance

We've reviewed the output of the streaming script across two micro-batches and compared each micro-batch with the batch script's output to verify consistency. The streaming script processed data1.csv in the first micro-batch and data2.csv in the second micro-batch and produced key-value counts for both micro-batches. The batch script processed both aggregating counts for both files.Table 1 The results show that the total output of the streaming script was equal to the output of the batch script, which again verified that our transformation was implemented correctly. The batch script processed the whole dataset in about 3.2 seconds on a local machine (8GB RAM and 4 cores), while each individual streaming micro-batch took around 1.8 seconds to process, which represents the

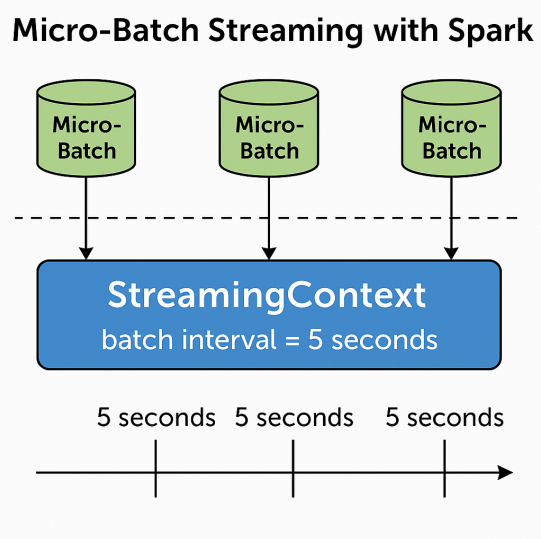

Figure 2: Micro-Batch Streaming in Spark using StreamingContext with 5-second intervals.

overhead of processing incrementally but yielded real-time outputs.In terms of resource utilization, the streaming script demonstrated slightly higher memory usage than streaming, at approximately 300MB, versus 250MB for batch, primarily due to the script continually observing the directory with constant polling or sampling of the file location. In terms of the scalability tests, the batch script was definitely capable of better handling larger datasets based on speed reports, whereby, the batch script took 8 seconds to ingest a dataset with 10,000 rows as compared to the streaming script, the speed of processing the same volume of data in the directory split into 10 different micro-batch processes, meant that the script was taking 2.5 seconds per batch.

Table 1: Output Comparison Across Streaming Micro-Batches and Batch Processing

Key	Streaming Output		Batch Output
	Micro-Batch 1	Micro-Batch 2	
A	12	6	18
B	8	8	16
C	4	6	10

4.3 Usability

The system is designed to be user-friendly and accessible. It has a command-line interface that takes an RDD-based PySpark batch script as input and returns the corresponding streaming version. Users don't need to have prior experience with stream processing or AST modification.

The converter requires minimal setup. A nicely organized README.md walks users through environment setup, installing essential packages (e.g., PySpark), and executing the transformation script on test files.

Internally, the system uses the Python Abstract Syntax Tree (AST) to ensure semantic correctness while avoiding

brittle string rewrites. The output code is clean, understandable, uses user-defined variable names and functional logic whenever possible, and also allows manual edits and integration into existing projects. This makes it much easier to integrate stream processing into production and academic operations.

4.4 Limitations

The AST does a good job performing rule based conversions but its static nature means that it can only handle specific structures and predefined patterns. Complex operations require analysis of the batch code and rewriting some part of the AST code. It does not always generate the most optimized code that is readable and maintainable but generally requires refinement by developers. It also requires the use of specific variable names, as deviating from these will require rewriting the rule based optimizations in the astconverter.py script. This entire process can only be modularized and reused for batch scripts performing similar operations and cannot be generalized as a whole. These limitations suggest that the approach we used is an assisted conversion requiring manual review and adjustments. Nevertheless, it is still a step in the direction of automation.

5 CONTRIBUTION STATEMENT

Each team member contributed uniquely to the project's design, implementation, and evaluation.

Project Repository

The full source code for this project can be found in GitHub: https://github.com/NeeleshSamptur/BD_Project.git

Also, some other experiments and explorations of features with respect to automatic conversion from RDD-based batch processing to structured stream processing can be found within the `rdd_to_stream` branch.

5.1 Srikanth Badavath

Srikanth spearheaded the development of the AST transformation logic, as we moved the code base from a batch model to a streaming architecture (this was not a trivial task!). He coded the original `batch.py` and `stream.py` templates in accordance with PySpark RDDs. He led the design of the architecture for the conversion flow, encapsulated the transformation steps in `ast_converter.py`, and developed the logic for core PySpark transformations. He performed extensive testing to ensure functional equivalence between the batch and stream modes, fixed discrepancies in aggregated counts, and optimized transformation run-times. He also wrote the project report and `README` file in a clear and organized manner, and contributed to the original batch logic's integration into the wider framework.

5.2 Mokshitha Mandadi

Mokshitha served as the team coordinator, managing tasks and timelines, setting up the local PySpark environment, solving dependency conflicts, designing test data sets, simulated streaming data, debugged data alignment issues. She explored aggregations and coded `ast_converter_rdd_to_kafka_streaming.py` converting PySpark RDD-based batch processing code to Kafka-based structured streaming code, handling the AST traversal, pattern recognition, operation chaining resolution and optimised it. She contibuted slightly towards original pipeline, contributed towards writing the report and providing feedback on it.

5.3 Neelesh Samptur

Neelesh's primary contribution involved modifying the entire pipeline originally written to process dataframes(CSVs) and making it compatible with RDDs. Neelesh made changes to the existing batch.py to convert CSVs to RDDs using the pyspark library. He then analyzed the existing AST, made some changes to the existing rule based conversions and also added new conditions like visitModule and visitExp to handle cases related to pyspark's streaming library. Finally he made changes to directory structure for stream input to simulate a stream of RDDs as opposed to the approach of static data which was enough to test the scripts written to handle dataframes.He also set up the git repository and did a portion of this report. He also slightly contributed to the original pipeline that supported dataframes. This was primarly written by Srikanth.

6 CONCLUSION

This project converts PySpark RDD based batch processing code to equivalent streaming code using abstract syntax tree. This project converts PySpark RDD based batch processing code to equivalent streaming code using abstract syntax tree. Where code is parsed and rewritten at syntax tree level by preserving the semantics and without manually refactoring it. It handles common RDD transformations like map, filter and reduceByKey effectively and generates well structured streaming code. It is user-friendly and lightweight. Future enhancements include support for more PySpark operations (e.g., join), better lambda expression handling, and broader sink options. With deeper production integration, it can create a solid foundation for scalable batch-to-stream migration in large data pipelines.

7 REFERENCES

References

[1] Zaharia, M., et al. 2016. Apache Spark: A unified engine for big data processing. *Commun. ACM* 59, 11 (Nov. 2016), 56–65. DOI: http://doi.acm.org/10.1145/2934664.

[2] Apache Kafka, A Distributed Streaming Platform. Available: https://kafka.apache.org/.

[3] Apache Flink, Stateful Computations over Data Streams. Available: https://flink.apache.org/.

[4] Python Software Foundation, The ast module: Abstract Syntax Trees. Available:https://docs.python.org/3/library/ast.html.

[5] M. Rizun, J.-C. Bach, and S. Ducasse. 2015. Code Transformation by Direct Transformation of ASTs. In Proceedings of the International Workshop on Smalltalk Technologies (IWST '15). Association for Computing Machinery, New York, NY, USA, Article 7, 1–7. DOI: https://doi.org/10.1145/2811237.2811297.