COMMISSIONED BY

bytewax

MCKNIGHT CONSULTING GROUP | Data Pipeline Performance Testing

Data Stream Processing Ease of Use and TCO

Bytewax vs.
Apache Flink

Related Research

https://www.exasol.com/resource/mcknight-cloud-analytics-top-database-performance-testing-report/

https://www.synadia.com/lp/nats-tco-comparison-kafka

William McKnight

Jake Dolezal

Prepared by
MCKNIGHT CONSULTING GROUP
McKnight Consulting Group
www.mcknightcg.com
July 2024

Contents

Executive Summary... 3
Landscape.. 4
Ease of Use.. 6
Infrastructure Utilization.. 10
Total Cost of Ownership... 12
Conclusion.. 15
Appendix A: Ease-of-Use Story Breakdown.. 16
Appendix B: Code Snippets.. 18
About Bytewax... 23
About McKnight Consulting Group.. 24
Disclaimer.. 25

Executive Summary

Organizations of various sizes and in a wide range of industries are actively using data in today's business environment to meet their strategic goals. The quest of data-driven decision-making is causing a significant shift, regardless of the objectives—increasing profitability, improving efficiency, raising risk tolerance, fostering readiness, promoting sustainability, or embracing adaptation in a world that is always changing. Businesses are being compelled by this shift to reconsider how they handle data integration.

Stream processing is a transformative solution for data integration, enabling real-time analysis and insights. It eliminates the need for batch processing, allowing organizations to identify trends and anomalies quickly. Stream processing frameworks handle massive data streams efficiently, enabling seamless integration efforts and enabling faster data-driven decisions and operational automation.

A comparison of stream processing pipelines was conducted, comparing Bytewax and Flink. The study found that Flink had 8-25 times more memory utilization than Bytewax, consuming up to 3 GB per workload. Bytewax was found to be 4.6 times less TCO than Flink, based on the development, maintenance, and annual infrastructure costs of running these pipelines just 4 times each. These advantages will add up over time and project by project, making Bytewax a more cost-effective choice for data stream processing.

We were impressed with Bytewax and believe Bytewax should merit strong consideration in any enterprise for a streaming workload.

Landscape

Data integration and management platforms have evolved to meet the evolving needs of businesses, with a focus on automation enabled by AI and cloud-native deployments. Modern methods allow for quick adjustments and efficient data flow between sources and targets, reducing manual labor and improving user experience. Trust is essential in data management, and modern data integration platforms must be equipped with powerful monitoring features to ensure data availability and dependability.

The growing dependence of enterprises on cloud-based data solutions necessitates cost-effectiveness in cloud environments, resource allocation optimization, and cost control. Transparency and understanding are the foundations of trust, and modern data management solutions provide businesses with a comprehensive understanding of company data and provide thorough lineage routes for tracking important information back to a reliable original source.

Modern cloud-based data analytics systems can scale dynamically and automatically to accommodate the increasing demands of many simultaneous and complicated query executions. The core of effective data use is simplified access, integration, control, and skillful handling, as well as the investigation and comprehension of new data sources or existing ones.

Difficulties associated with poor data integration and management can be attributed to personnel or process problems, inadequate tools, or a reliance on outdated technology. It is becoming increasingly difficult to ignore the growing need for sophisticated data integration and management solutions, as it is no longer feasible to continue using antiquated systems. The time has come for streaming data management, and integration that has developed to address the issues associated with modern data.

All of these capabilities drive ease of use and the ultimate metric – TCO. In this benchmark, we assess both, as well as infrastructure utilization, for two streaming data integration players.

Flink

For scenarios demanding immediate or near-real-time insights, traditional data processing solutions may fall short due to their batch-oriented nature. Apache Flink has emerged as a powerful alternative in such cases. This open-source framework excels at handling data streams, allowing for continuous ingestion, analysis, and generation of results. With Flink, data is processed as it arrives, enabling quicker identification of trends, anomalies, and crucial insights to inform real-time decision-making. Flink is a Java-based stream processor that utilizes the Flink scheduling and management layer to parallelize tasks and efficiently use the resources that it has been allocated.

Apache Flink has been recognized for its impact on large-scale data management systems. The project has a large and active community. Leading companies like Airbnb, Uber, Netflix, and Stripe rely on Flink for mission-critical streaming workloads. In 2023, Flink experienced a record 22 million monthly downloads, demonstrating its reliability and scalability in real-time data analytics.

bytewax

Bytewax is a Python-centric framework that simplifies real-time stream processing tasks within the Python environment. It allows users to integrate the entire Python ecosystem into their workflows, including popular environments like Jupyter Notebooks or RaspberryPi, powerful tools like Hugging Face transformers, Scientific Python tools like Numpy and Pandas, and frameworks like Streamlit. Bytewax offers stateful operators for advanced processing and prioritizes data safety with built-in state recovery options. It also provides native connectors for popular data sources like Kafka, Redpanda, DynamoDB, and BigQuery, simplifying data ingestion and management. The Rust engine ensures smooth operation even with demanding workloads, allowing Bytewax to scale across thousands of workers for maximum efficiency.

Bytewax is young, scrappy, motivated, and ready to put its capabilities against all players in the growing data streaming market and contribute to organizational goals very directly through the cultivation of the ultimate resource – enterprise data.

Ease of Use

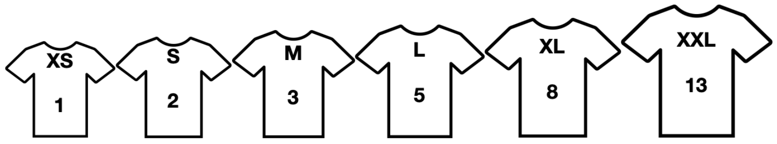
This section analyzes the results of our comparison of ease-of-use.

Methodology

We took inspiration from the agile scrum development project management approach. As experienced consultants in the field, we have extensively used and advocated for the agile project management methodology when developing and operating information management projects. The agile methodology is a much bigger subject than the scope of this paper, but for its purposes, we used a few concepts from an agile methodology in this study that may be familiar to readers.

- Story – A story is a specific task that needs to be completed to move forward in developing a new piece of the project or to complete an operational objective.
- Story Size – A story is sized according to how much time and effort is required to complete it. Sizing a story appropriately is an art and a science that requires some experience.
- Story Points – Regardless of the sizing method used, the story size is typically expressed as a numeric value to quantify story work and completion.

To quantify size and points, we used the Fibonacci sequence, which could be likened to using T-Shirt sizes per the image below. Tasks go from the extra-small (XS) size to the extra-extra-large (XXL) size. The chart below shows each size and the Fibonacci number assigned to it.



and so on…

Use Cases

For our ease-of-use comparison, we developed 4 use cases that are table stakes use cases for data stream processing. We built the pipelines on each platform.

Use Case Pipelines	Source-Target

1. Real-time analytics with stream enrichment	Kafka to Kafka
2. Real-time machine learning	Kafka to Redis
3. Real-time ingest for Retrieval Augmented Generation (RAG)	Kafka to Vector
4. Real-time IoT processing	IoT to Kafka

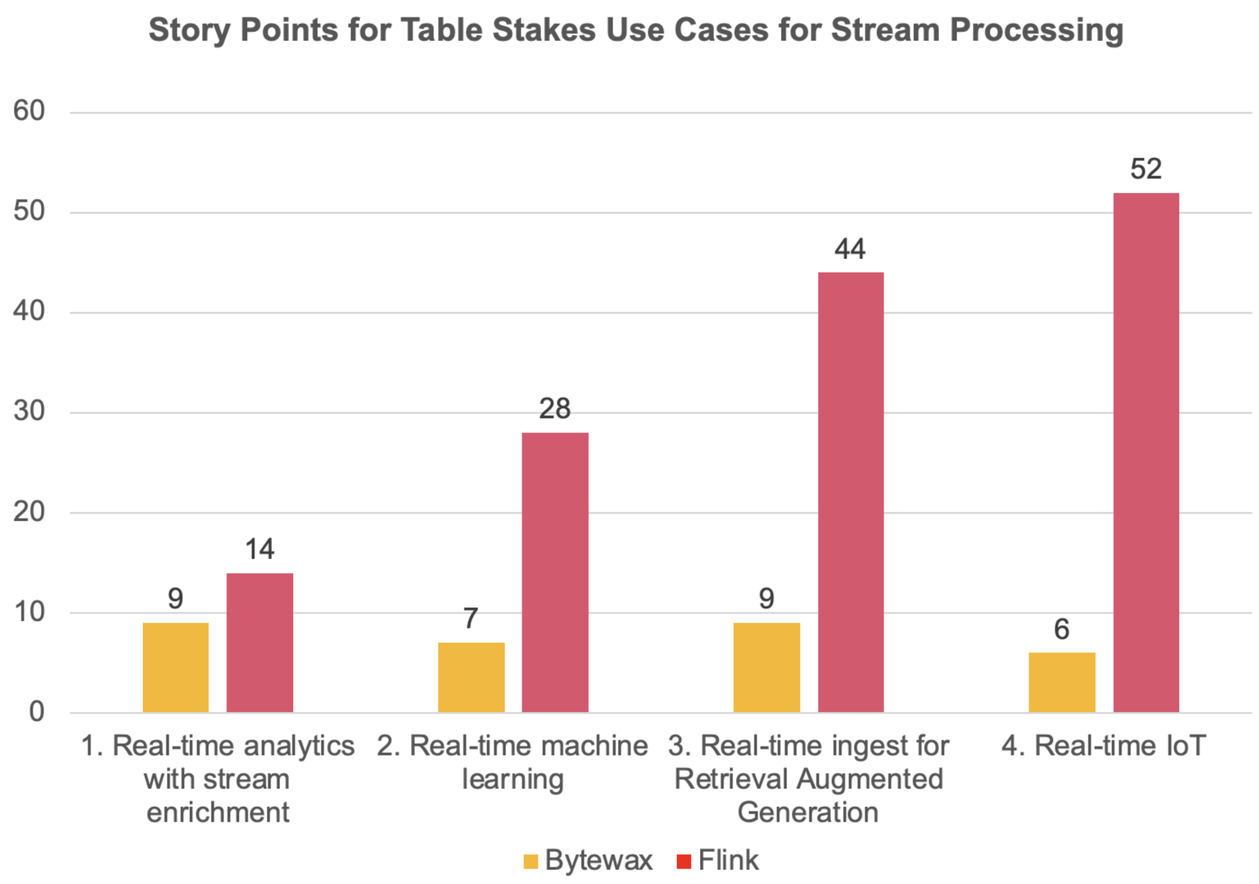
These use cases have a fairly consistent set of development tasks/user stories in all products in order to develop them into production-ready streaming data pipelines. These stories include, but are not limited to:

- Create producer connections
- Create consumer/sink connections
- Write data processing functions
- Write stream deserializers into custom objects
- Create message backup scripts
- Connect to reference databases
- Write join and enrichment functions

We developed these use cases using documented best practices by the vendors and consistently estimated the story points in arrears as we did the tasks in our work breakdowns[1]. Then, we converted our story T-shirt sizes to numbers in the Fibonacci sequence and summed up the story points for each streaming data use case. The following table is a summary of those total points per use case.

Total Story Points by Use Case	bytewax	Flink
1. Real-time analytics with stream enrichment	9	14
2. Real-time machine learning	7	28
3. Real-time ingest for RAG	9	44
4. Real-time IoT processing	6	52

[1] In our approach, we estimated a developer executes an average of 5 story points' worth of development work per week.



A breakdown of the tasks for each use case can be found in the appendix.

In our study, the clear advantage goes to Bytewax because the pure Python approach and their SDK made coding simple and straightforward for each of our use cases. With Flink, we had the challenge of coding in Java—resulting in longer code blocks, complex object orientation, more troubleshooting, and additional error handling for each step.

On a case-by-case basis, this advantage may not seem like much. However, over time and project after project, these advantages from using Bytewax will add up. If you consider the case of a single developer building multiple pipelines, development cycles get longer, and time-to-production gets extended.

Bytewax offers significant advantages within a Python-centric environment. Existing Python developers can leverage Bytewax for stream processing tasks without needing to learn Java, a skill required for many functionalities in Apache Flink, such as advanced windowing and state management. This eliminates the need to onboard team members with new skill sets, which can significantly reduce project overhead and coordination efforts. Additionally, by enabling current developers to handle streaming tasks using familiar Python tools, Bytewax has the potential to lower training costs associated with introducing them to a new paradigm like stream processing.

The following table demonstrates this if you consider a developer can achieve an average of 5 story points worth of development work per week (adding 25% to the story points for non-development path-to-production activities).

Single Developer Time-to-Production Per Project	**bytewax**	**Flink**	
1. Real-time analytics with enrichment	2.6 weeks	4.0 weeks	
2. Real-time machine learning	2.0 weeks	8.0 weeks	◄ **4x longer!**
3. Real-time ingest for RAG	2.6 weeks	12.6 weeks	◄ **5x longer!**
4. Real-time IoT processing	1.7 weeks	14.9 weeks	◄ **9x longer!**

Moreover, as we will see in the TCO section, this advantage becomes enormous when you consider the need to develop, improve, and maintain multiple enterprise-scale streaming data processing pipelines each year. We will see this when we project the frequency of these use cases into the enterprise in the TCO section below.

Infrastructure Utilization

This section analyzes the results of our benchmark comparison of infrastructure utilization to determine the cloud costs to run and operate these platforms.

In our experience, testing day-in-day-out workloads is more useful for calculating total cost-of-ownership than testing edge and extreme cases, and we find that the great majority of our clients who use streaming data pipelines have volumes of less than 200,000 transactions per second.

Since Flink is based on Java and Bytewax is based on Python, we were very interested in memory utilization because it has a bearing on how much infrastructure we need to deploy to cover all our pipelines in an enterprise production scenario. Java is of interest because of the notorious consumption of memory by the Java heap and the frequent challenge of managing the memory of the Java Virtual Machine (JVM).

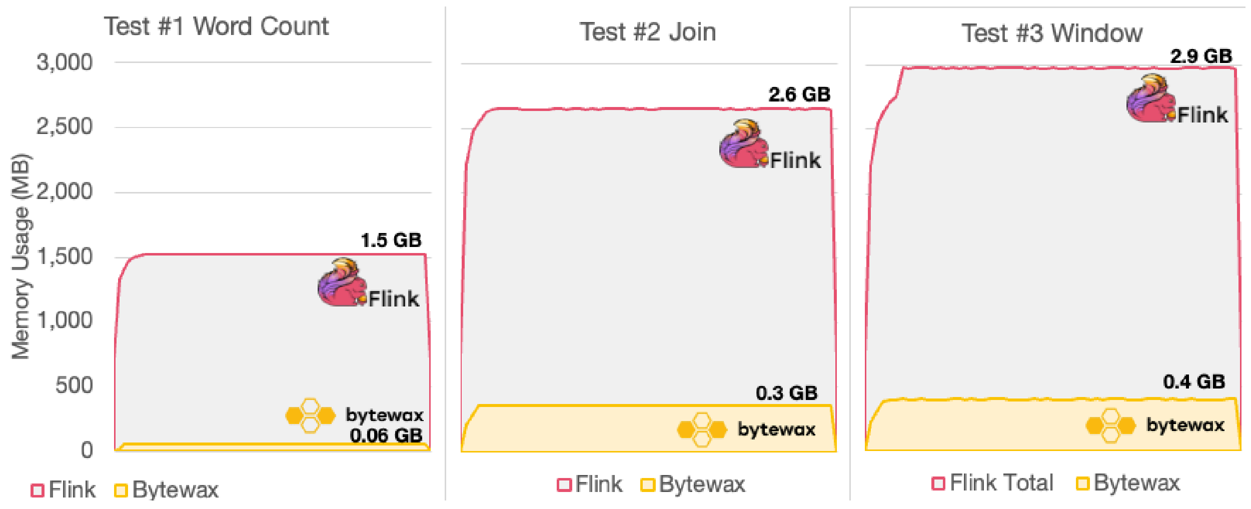
We devised three simple test workloads based on common workloads:

1. Word count – counting the words in a data stream—which can mimic any number of aggregation type activities for streaming data
2. Join – joining two JSON data streams together on a key and producing a combined stream
3. Window – utilizing a windowing function to make calculations across a moving window of streaming data

For the test, we used identical transaction rates (200,000 per second). For Flink, we used its Python SDK (PyFlink) to create an apples-to-apples comparison. We also ran the workload to a specified amount of time (1 minute for each iteration) and completed three cycles of each test. We took measurements of memory utilization every second using SAR[2], and we also monitored the memory by process with TOP[3] to understand the baselines and the memory used by Python and Java while the workload was running.

The following graphs detail the memory utilization measurements we took during our benchmark.

[2] SAR stands for System Activity Report, a command-line tool in Linux and Unix-based systems. The sar command allows users to monitor and record system resources like CPU usage, network activity, and yes, memory utilization.
[3] TOP is a widely used command-line utility in Linux and Unix-based systems. The top command provides a real-time view of various system resources, including memory usage for each running process.



As you can see, the memory utilization by Flink (PyFlink plus Java) was between 8 and 25 times more than Bytewax, depending on the test, consuming up to 3 GB per workload.

You can fit 8 to 25 more Bytewax pipelines on a single virtual machine than you can with Flink. As we will show in the next section, this can have significant bottom-line cost impacts.

Total Cost of Ownership

This section analyzes a total cost of ownership scenario for the platforms we tested.

Platform Costs

We tested the open source, community, free versions of these platforms. Please note that Bytewax offers a platform version with Business and Enterprise tiers which gives access to:

- Premium connectors
- Management, observability, and data flow dashboards and APIs
- Kubernetes multi-node deployment
- OpenID connectivity and role-based access controls
- Cloud backup
- Private and enterprise-level support

Infrastructure Costs

From our benchmarking exercise described in the previous section, we found memory consumption to be the largest driver of infrastructure requirements between these two platforms.

To illustrate this cost scenario, we sampled the pricing of several memory-optimized EC2 instances available on AWS. Memory-optimized instances are designed to deliver good performance for workloads that process large data sets in memory. We found that the average hourly cost of the EC2 instances is a function of vCPU cores and the GB of RAM. It works out to approximately $0.0124 per GB of RAM per hour.

For the purposes of this cost scenario, we also assumed using AWS Elastic Kubernetes Service ($0.10 per cluster per hour) across 3 availability zones and 24 data stream pipeline processes in production.

We utilized RocksDB with Flink to store state, which adds a recommended 40% more memory consumption[4]. Bytewax uses SQLite, which adds a negligible amount of memory to retain state.

Thus, by our calculations, each platform would require the following memory:

[4] Depending on your configuration of the default managed memory fraction. Flink recommends 0.4 in their documentation.

Memory Consumption	bytewax	Flink	
Per Pipeline	0.4 GB	9.6 GB	
Total Memory Consumption	**4.1 GB**	**97.9 GB**	◄ **10x more memory!**

And from this memory consumption, we can derive infrastructure costs for this configuration and deployment.

Cloud Infrastructure Costs	bytewax	Flink	
Per Hour	$1.05	$4.35	
Per Month	$758	$3,130	
Per Year	**$9,096**	**$37,562**	◄ **4x more cost!**

People Time, Effort, and Cost Assumptions

Employees contribute to the initial implementation and largely to the maintenance of these projects. Their time and effort cannot be discounted. Choosing a platform with a lower time-effort burden to build, operate, and maintain will free developer time and energy to work on more strategic projects within the organization. Based on the pipeline development we did on both of these platforms, detailed above, we provide an estimate of what an organization might need to have a developer do this for a modest 4 iterations of each pipeline development per year.

This is our assumption regarding the annual cost of that developer(s).

People Time, Effort, and Costs	
Standard Work Time per Year	50 weeks
Story Points Per Developer Per Week	5 points
Average Annual Cash Compensation	$150,000
Burden rate	22%
Fully Burdened Annual Labor Cost	**$183,000**

Total Cost of Ownership

To project total of ownership, we made the following calculations.

The following tables break down the development and maintenance costs of running these platforms.

Development & Maintenance	bytewax	Flink	
Total Story Points Per Year	124	552	
Maintenance Points Per Year	+25%	+30%	
Total CICD Points Per Year	**155**	**718**	**◄ 4.6x more effort with Flink!**
Developer FTE Needed	0.6	2.9	
People Costs	$113,460	$525,283	

Annual Infrastructure Costs	bytewax	Flink	
EC2 Instances	**$9,096**	**$37,562**	**◄ 4.1x more infrastructure for Flink!**

Total Cost of Ownership	bytewax	Flink	
One-year Period	**$122,556**	**$562,846**	**◄ 4.6x greater TCO for Flink!**

Bytewax costs over four times less each year with developer costs figured in. Keep in mind that for customers who require building many more pipelines than this, which will be most, the developer count would escalate with Flink to support that development.

Conclusion

A stream processing pipeline comparison between Flink and Bytewax was done. According to the study, Flink used up to 3 GB of memory per task, which is 8–25 times more than Bytewax. Based on development, maintenance, and the annual infrastructure costs of operating these pipelines just four times apiece in a year, Bytewax was shown to have a TCO that was 4.6 times lower than Flink's, to accomplish the same functions.

Bytewax will eventually become a more affordable option for data stream processing. These benefits will accumulate over time and project by project.

Bytewax is a robust stream processing framework that is particularly suitable for enterprise-grade streaming workloads. It operates entirely within the Python ecosystem, allowing developers to use familiar libraries and environments like Jupyter notebooks for data exploration and Hugging Face transformers for advanced natural language processing tasks. This intuitive environment lowers the barrier to entry for developers and data scientists, accelerating the development and deployment of streaming applications.

Bytewax excels in handling stateful operations, enabling tasks like joins, windowing, and aggregations, enabling developers to construct complex workflows that analyze streaming data in relation to past events. It also prioritizes robustness through built-in state recovery mechanisms, ensuring critical workloads can withstand potential failures without data loss.

Bytewax's architecture allows it to distribute processing tasks across thousands of workers, making it adept at handling even the most demanding streaming data pipelines. This scalability allows Bytewax to grow alongside business needs, adapting to ever-increasing data volumes and processing requirements. In conclusion, Bytewax's user-friendly Python environment, robust stateful processing, built-in state recovery, and exceptional scalability make it a strong contender for any enterprise considering a streaming workload solution.

In our evaluation of various stream processing frameworks, Bytewax emerged as a particularly compelling solution for enterprise-grade streaming workloads.

Appendix A: Ease-of-Use Story Breakdown

Real-time analytics with stream enrichment	bytewax	Flink
a. Create Kafka producer	1	1
b. Create Kafka consumer	1	1
c. Write data processing function	1	2
d. Write data deserializer to custom object	2	2
e. Create message backup script	2	3
f. Connect to reference database	1	2
g. Write join and enrichment function	1	3
TOTAL	**9**	**14**

Real-time machine learning	bytewax	Flink
a. Create Kafka producer	1	1
c. Write data processing function	1	2
d. Write data deserializer to custom object	2	2
d. Create Redis connector	1	5
e. Create PubSub pipeline	1	8
f. Redis set key async I/O operator	1	5
g. Create async micro-batch sync	Unnecessary	5
TOTAL	**7**	**28**

Real-time ingest for Retrieval Augmented Generation	bytewax	Flink
a. Create Kafka producer	1	1
b. Write data processing function	1	2
c. Write data deserializer to custom object	2	2

Real-time ingest for Retrieval Augmented Generation	bytewax	Flink
d. Write enrichment function	2	5
e. Write custom Pinecone sink	2	21
f. Write RAG query to latest data	1	13
TOTAL	**9**	**44**

Real-time IoT	bytewax	Flink
a. Write data processing function	1	2
b. Write data deserializer to custom object	2	2
c. Create Kafka consumer	1	1
d. Create MQTT connectors	1	34
e. Configure MQTT connectors	1	13
TOTAL	**6**	**52**

Appendix B: Code Snippets

We have included snippets of code we used for the test cases. This highlights the ease of coding with Bytewax compared to PyFlink–even in these simple cases.

NOTE: Input and output connectors and trivial details were removed for brevity and replaced with ellipses (...). Other standard Python library imports were also removed.

Test 1: Word Count

Bytewax

```
from bytewax.dataflow import Dataflow
import bytewax.operators as op

...

flow = Dataflow("wordcount")

def tokenize(line):
  return re.findall(r'[^\s!,.?":;0-9]+', line)

tokens = op.flat_map("tokenize_input", ..., tokenize)
counts = op.count_final("count", tokens, lambda word: word)
op.output("out", counts, ...)

...
```

PyFlink

```
from pyflink.common import WatermarkStrategy, Encoder, Types
from pyflink.datastream import StreamExecutionEnvironment,
RuntimeExecutionMode
from pyflink.datastream.connectors.file_system import (FileSource,
StreamFormat, FileSink, OutputFileConfig, RollingPolicy)

...

env = StreamExecutionEnvironment.get_execution_environment()
env.set_runtime_mode(RuntimeExecutionMode.BATCH)
env.set_parallelism(1)

def split(line):
  yield from line.lower().split()
```

```
ds = env.from_collection(...)

ds = ds.flat_map(split) \
  .map(lambda i: (i, 1), output_type=Types... \
  .key_by(lambda i: i[0]) \
  .reduce(lambda i, j: (i[0], i[1] + j[1]))

env.execute()

...
```

Test 2: Join

Bytewax

```
from bytewax.dataflow import Dataflow
import bytewax.operators as op

...

flow = Dataflow("join")

inp1 = op.input("inp1", flow, ...)
inp2 = op.input("inp2", flow, ...)

merged_stream = op.merge("merge", inp1, inp2)

...
```

PyFlink

```
from pyflink.common import WatermarkStrategy, Row
from pyflink.common.serialization import Encoder
from pyflink.common.typeinfo import Types
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream.connectors import FileSink, OutputFileConfig,
NumberSequenceSource
from pyflink.datastream.execution_mode import RuntimeExecutionMode
from pyflink.datastream.functions import KeyedProcessFunction, RuntimeContext,
MapFunction, CoMapFunction, CoFlatMapFunction
from pyflink.datastream.state import MapStateDescriptor, ValueStateDescriptor
from pyflink.common import JsonRowDeserializationSchema,
JsonRowSerializationSchema

...
```

```
class JoinStream(CoFlatMapFunction):

  def open(self, runtime_context: RuntimeContext):
    state_desc = MapStateDescriptor('map', Types.INT(), Types.STRING())
    self.state = runtime_context.get_map_state(state_desc)

  def flat_map1(self, value):
    self.state.put(value[0], value[1])

  def flat_map2(self, value):
    self.state.put(value[0], value[1])

env = StreamExecutionEnvironment.get_execution_environment()
env.set_parallelism(1)

ds = env.from_collection(...,
  type_info=Types...)

ds2 = env.from_collection(...,
  type_info=Types...)

connect_ds = ds.connect(ds2)
connect_ds.key_by(lambda a: a[0], lambda a: a[0]).flat_map(JoinStream(),
Types...)

env.execute()

...
```

Test 3: Windowing

Bytewax

```
import bytewax.operators as op
import bytewax.operators.windowing as win
from bytewax.dataflow import Dataflow
from bytewax.operators.windowing import EventClock, TumblingWindower,
WindowMetadata

...

flow = Dataflow("windowing")

align_to = datetime(..., tzinfo=timezone.utc)
stream = op.input("input", flow, ...)
keyed_stream = op.key_on("key_on_user", stream, lambda e: e["user"])
```

```
clock = EventClock(lambda e: e["time"],
wait_for_system_duration=timedelta(seconds=0))
windower = TumblingWindower(length=timedelta(seconds=10), align_to=align_to)

win_out = win.collect_window("add", keyed_stream, clock, windower)

op.output("out", win_out.down, ...)

...
```

PyFlink

```
from pyflink.datastream.connectors.file_system import FileSink,
OutputFileConfig, RollingPolicy
from pyflink.common import Types, WatermarkStrategy, Time, Encoder
from pyflink.common.watermark_strategy import TimestampAssigner
from pyflink.datastream import StreamExecutionEnvironment,
ProcessWindowFunction
from pyflink.datastream.window import TumblingEventTimeWindows, TimeWindow

...

class MyTimestampAssigner(TimestampAssigner):
    def extract_timestamp(self, value, record_timestamp) -> int:
        return int(value[1])

class CountWindowProcessFunction(ProcessWindowFunction[tuple, tuple, str,
TimeWindow]):
    def process(self,
                key: str,
                context: ProcessWindowFunction.Context[TimeWindow],
                elements: Iterable[tuple]) -> Iterable[tuple]:
        return [(key, context.window().start, context.window().end, len([e for
e in elements]))]

env = StreamExecutionEnvironment.get_execution_environment()
env.set_parallelism(1)

win_stream = env.from_collection(...,
  type_info=Types...)

watermark_strategy = WatermarkStrategy.for_monotonous_timestamps() \
    .with_timestamp_assigner(MyTimestampAssigner())

ds = win_stream.assign_timestamps_and_watermarks(watermark_strategy) \
    .key_by(lambda x: x[0], key_type=Types...) \
    .window(TumblingEventTimeWindows.of(Time.milliseconds(5))) \
    .process(CountWindowProcessFunction(),
            Types...)

env.execute()

...
```

About Bytewax

Bytewax was started to support organizations develop machine learning and AI capabilities particularly managing the data to support machine learning and AI inference. To fulfill this mission, Bytewax developed the open source project Python stream processing project of the same name. Bytewax Python stream processor supports a wide variety of use cases that leverage advanced operations like joining data streams, computing windowed calculations, maintaining representations of state, tracking progress, managing out-of-order data, and more. Some of these applications include

1. processing data for recommendation systems,
2. anomaly detection in physical and virtual systems,
3. fraud detection for financial institutions,
4. processing unstructured data for search and AI systems,
5. analyzing virtual systems in real time.

Bytewax also provides a commercial offering (Bytewax Modules and the Bytewax Platform) to extend the capabilities of the open source library to cover the complexities of writing, scaling, and managing many stream processing workflows. Together, the open source library, the Modules, and the Platform provide a complete data management solution for real-time and streaming data processing.

Bytewax's Python interface and cloud-native architecture make it the easiest stream processing system for developers resulting in organizations ability to bring products to market faster and with a lower burden of development and maintenance. All resulting in a lower TCO and higher ROI on development projects, as outlined in this comparison with the leading stream processing tool, Apache Flink.

About McKnight Consulting Group

Information Management is all about enabling an organization to have data in the best place to succeed to meet company goals. Mature data practices can integrate an entire organization across all core functions. Proper integration of that data facilitates the flow of information throughout the organization which allows for better decisions – made faster and with fewer errors. In short, well-done data can yield a better run company flush with real-time information... and with less costs.

However, before those benefits can be realized, a company must go through the business transformation of an implementation and systems integration. For many that have been involved in those types of projects in the past – data warehousing, master data, big data, analytics - the path toward a successful implementation and integration can seem never-ending at times and almost unachievable. Not so with McKnight Consulting Group (MCG) as your integration partner, because MCG has successfully implemented data solutions for our clients for over a decade. We understand the critical importance of setting clear, realistic expectations up front and ensuring that time-to-value is achieved quickly.

MCG has helped over 100 clients with analytics, big data, master data management and “all data” strategies and implementations across a variety of industries and worldwide locations. MCG offers flexible implementation methodologies that will fit the deployment model of your choice. The best methodologies, the best talent in the industry and a leadership team committed to client success makes MCG the right choice to help lead your project.

MCG, led by industry leader William McKnight, has deep data experience in a variety of industries that will enable your business to incorporate best practices while implementing leading technology. See www.mcknightcg.com.

Disclaimer

McKnight Consulting Group (MCG) runs all its tests to strict ethical standards. The results of the report are the objective and unbiased results of the application of queries to the simulations described in the report. The report clearly defines the selected criteria and process used to establish the field test. The report also clearly states the data set sizes, the platforms, the methods, etc. that were used. The reader is left to determine for themselves how to qualify the information for their individual needs. The report does not make any claims regarding third-party certification and presents the objective results received from the application of the process to the criteria as described in the report. The report strictly measures performance and does not purport to evaluate other factors that potential customers may find relevant when making a purchase decision. This is a sponsored report. The client chose its configuration, while MCG chose the test, configured the database and testing application, and ran the tests. MCG also chose the most compatible configurations for the other tested platforms. Choosing compatible configurations is subject to judgment. The information necessary to replicate this test is included. Readers are encouraged to compile their own representative configuration and test it for themselves.