
CHOOSE THE RIGHT STREAM PROCESSING ENGINE FOR YOUR DATA NEEDS

Technical and Operational Factors that are
Crucial to the Decision Making Process



DATA IN
MOTION

Process Data Streams at the Speed of Business and at the Scale of IT

Business opportunities that directly impact revenue or boost operational efficiency need to be addressed in near real-time. Digital transformation initiatives and the advancement in mobility, IoT and streaming technologies has led to enterprises being inundated with data. Key business requirements determine how such high volumes of high-speed data should be processed in real-time to provide actionable intelligence. This directly leads to IT having to evaluate which stream processing engine is best fit for purpose for their enterprise needs. Other determining factors include return on investment, its dexterity to be applied across multiple use cases, and its level of maturity for enterprise wide adoption.

This paper is meant to help technology architects and developers choose the right stream processing engine for their needs. We do this by analyzing key technical and operational differentiators between four modern stream processing engines from the Apache open source community:

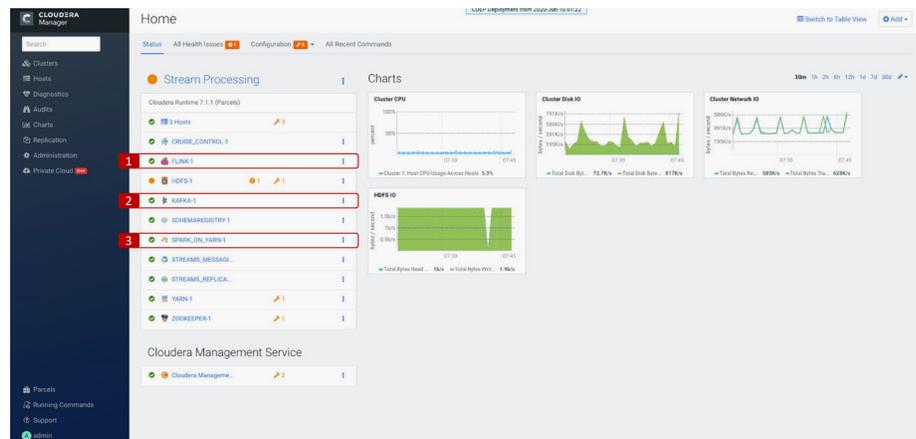
- Kafka Streams
- Spark Structured Streaming
- Storm with Trident
- Flink

This paper also highlights some of the capabilities that are key to any data streaming use case, such as:

- Watermarks to handle late and out of order delivery
- Windowing semantics to structure the streams
- Complex event processing
- Capabilities that enhance operational efficiency

Cloudera offers all of the engines listed above, because we believe that you should use the best tool for the job. Sometimes that tool is a very simple one, but more often than not, you will need the advanced capabilities for your specific use cases.

There are a variety of ways by which to address data stream processing challenges. The solution comes down to the fundamental way in which the engine works and how your organization implements it.



Cloudera Manager provides one view to manage all of your resources including stream processing engines. Here we see Flink (1), Kafka (2), and Spark (3) resources in one comprehensive view. Source: Cloudera.

Table of Contents

Address Challenges Through Informed Decision Making	4
Streaming Challenges	4
Decision Making Process: Technology and Operational Considerations	5
Technology Considerations for Stream Processing Engine Evaluation	5
Functional Aspects	5
Developmental Control	6
Implementation and Beyond	6
Operational Considerations for Stream Processing Engine Evaluation	8
Enterprise Adoption	8
Enterprise Operations	8
Spark, Kafka, or Flink? Which to Use?	9
Flink Use Cases	9
Cybersecurity and Log Processing	9
Outage Classification for Telecom Companies	10
Financial Services: Mainframe Offloading	10
IoT for Manufacturing	10
Technical Features Table	11
Operational Features Table	12
Customer Success	13
Ensure Fit for Purpose and Enterprise Wide Adoption	13

The “streaming first” approach

Stream processing engines have followed different paths in their approach to solving unique time reasoning challenges.

Flink is a “streaming first” distributed system. This means that it has always focused on solving the difficult unbounded stream use cases over bounded stream and batch scenarios.

It turns out that algorithms that work on unbounded streams, also work on bounded streams by treating the latter as a special case of the former. As a result, Flink addresses micro-batch use cases as well.

Address Challenges Through Informed Decision Making

Global digitization has resulted in a vast array of new products and services with such high levels of convenience that it fuels a continuous loop of greater expectations for immediacy. Next day delivery and real-time payments are demands driven by consumers at the point of service that then pressures downstream services to respond faster. Processing and analyzing billions of events per second across geographies is becoming an ordinary affair.

In response, technology teams have been pivoting from large monolithic database architectures to event driven applications and microservices design as a way to reduce the inevitable latency of inputs and outputs across networks by bringing the state of an event closer to the application itself.

Central to this effort are modern data stream processing engines like Storm with Trident, Kafka Streams, Spark Structured Streaming, and Flink. The Storm/Trident framework is the oldest, while Kafka Streams is the newest. The Spark Structured Streaming community is large while Flink’s is growing rapidly. Knowledge of an engine’s development community can help gauge how self-sufficient and productive your team can be.

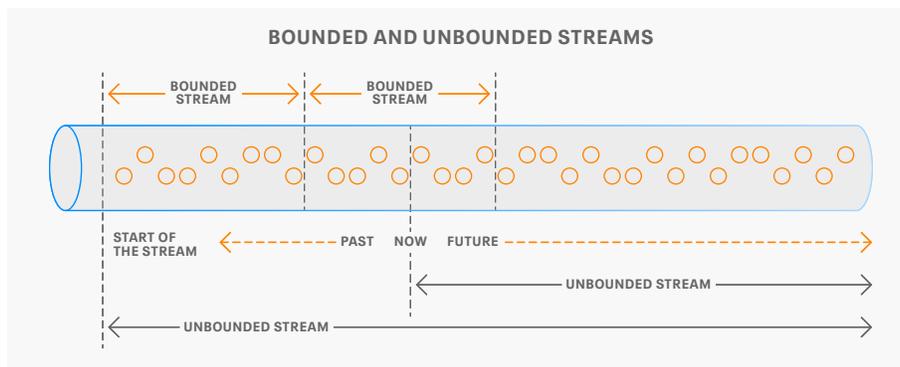
The engine that is best for you depends on your organization’s use cases, team makeup, and various technology and operational factors. This paper is meant to help you in that evaluation process.

Streaming Challenges

Below is a reminder of streaming challenges you’ve undoubtedly had or will come across.

Event time and processing time—The chances that streaming events come in without any delays and with predictable patterns is low, because you can’t control the myriad of input sources that exist across collections of networks that vary in type and quality. Even with the very best networks and the fastest collection mechanisms, there will always be latency between the time an event happens in the real world (event time) and the time your system processes it (processing time).

Bounded and unbounded streams—Bounded streams have a beginning and an end, so it is easy to reason about time and correctly sort events, akin to batch. Unbounded streams are harder to reason about because, without an end, you don’t know if another live event is yet to come. Calculations, aggregations or pattern detection in unbounded streams is very tricky. To handle both scenarios, it is helpful to follow a “streaming first” principle (see [The “streaming first” approach](#)), and to consider capabilities like watermarks to handle late and out of order events (see [Watermarks to handle late delivery](#) on page 5).



Watermarks to handle late delivery

Watermarks are a comprehensive way of handling late or out of order arrivals by providing a set of trigger messages that are injected alongside the data stream.

For unbounded streams, in which you don't have a definitive end, watermarks delineate points at which you would expect all of the events to have occurred. It is from here that you can establish some logic.

A collection of watermarks creates windows, and this is what gives your streams the structure to which reasoning can be applied (see [Windows semantics](#) on page 7).

Flink provides a lot of control as to how watermarks are generated. This provides more options as to how completely you want to capture events that may or may not arrive. This mechanism can also extend to very sophisticated functionality like leveraging upstream and downstream materialized views or using batch engines to reprocess and incorporate late data.

Simple and complex events—Complex events are derived from simple events that have been aggregated, patterned, and evaluated to trigger a response or present a result, often on data that continuously moves under your feet. Decisioning on unbounded streams requires the state of events to be stored and analyzed.

Stateless and stateful—Stream processing engines excel when analytics require a reassessment of events within the context of time. That is considered stateful, while stateless represents a self-contained fire-and-forget paradigm. There are acceptable trade-offs between stateless high throughput engines and stateful engines that need to address aggregation, enrichment, and other requirements

Decision Making Process: Technology and Operational Considerations

There is often an overreliance on streaming benchmarks when choosing a stream processing engine. Benchmarks focus simply on latency, throughput, and hardware utilization and don't consider functional requirements or the control that developers would have in effectuating a solution. Benchmarks also don't assess important operational, staffing, and other nonfunctional criteria. The rest of the paper summarizes technology and operational considerations that are needed to make an informed decision and to foster adoption of the solution across the enterprise.

Technology Considerations for Stream Processing Engine Evaluation

The next section compares the different stream processing engines with regard to functional, developmental, and implementation considerations.

Functional Aspects

The functional capabilities of stream processing engines as they pertain to approach, streaming model, and time support are used to solve specific business requirements.

Approach—The type of approaches that development communities took at the inception of an engine's development include: "streaming first", "message broker first", and "batch first". The distinction helps in understanding what the engine was originally meant for.

Both Flink and Storm/Trident took a "streaming first" approach. The former is regarded as a modern class leader, while the latter is considered legacy architecture.

Spark Structured Streaming followed a "batch first" approach, while Kafka Streams was initially developed as a "message broker first". Streaming capabilities are popular add-ons for both.

Streaming model—Earlier, we described the concept of "stateless" and "stateful" and that it is critical to distinguish between the two, and the trade-offs between throughput and latency. Natively, Storm utilizes a stateless streaming model and is very useful if you have simple low latency use cases. Its combination with Trident enables some stateful capabilities.

Flink, Kafka Streams, and Spark Structured Streaming are all stateful, but with slight differences. Having taken a "batch first" approach, Spark Structured streaming handles events as micro-batches and is good when high throughput is necessary but low latency is not a big requirement. The two other stateful engines differ in how they store state. Kafka Streams depends on the Kafka ecosystem, while Flink provides more storage options. Both process messages an event at a time and are considered low latency solutions.

Time support—All of the stream processing engines described in this paper are able to distinguish event time from processing time. The nuance is in how much control you have to address some of the trickier use cases. Flink provides a great deal of control with capabilities such as watermarking and session windows (see [Watermarks to handle late delivery](#) and [Window semantics](#) on page 7).

Developmental Control

A common task in every data processing use case is to import data from one or multiple systems, apply transformations, and then export results to another system. Considering the ubiquitousness of streaming data applications, unified integration with machine learning, graph databases, and complex event processing is becoming more common.

Processing abstraction—To help your engineering team be productively focused on business logic instead of advanced streaming concepts, it's important to evaluate the stream processing engine's processing abstraction capabilities.

Spark Structured streaming is strong with machine learning due to its set of libraries. If you are already developing within a Spark ecosystem, the stream processing engine decision is that much easier.

Special attention should be paid to the engine's SQL abstraction. From an analytics democratization point of view, SQL abstraction is a very important basis for comparison. While many senior developers prefer sophisticated languages like Scala for complex analytic work, the expressiveness and simplicity of SQL can get the job done more easily and it is accessible to a wider range of developer resources.

When it comes to the comparison on the basis of SQL, the more standard the better. Flink has the most mature and production tested OpenSource SQL-on-Streams implementation and is fully ANSI compliant. Kafka's KSQL is maturing nicely, but is not fully OpenSource, not ANSI compliant, and not as feature rich. Spark Structured Streaming SQL, though well adopted, has ANSI compliance dating back to 2003. Storm/Trident is in the SQL experimental stages.

Implementation and Beyond

Application development is only as good as its implementation. Below, we cite aspects that need to be considered to move beyond the idea and development stage.

Delivery guarantee—This is a key factor to consider as it relates to your expectations of latency, throughput, correctness, and fault tolerance of message delivery.

At-least-once delivery may result in duplicate messages after multiple delivery attempts but you know at least one succeeds. Performance is high with the least overhead, because the state of delivery tracking is not stored. It runs in a fire-and-forget fashion, sufficing low latency, high throughput, guaranteed message requirements but with little regard to data duplication.

For more exacting applications, such as financial transactions, it is important that messages are received and processed exactly-once. This requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. Exactly-once is optimal in terms of correctness and fault tolerance, but comes at the expense of added latency.

All the engines described in this paper provide exactly-once delivery guarantee, though Kafka Streams is limited to the Kafka ecosystem and can't control downstream systems. Flink and Spark Structured Streams guarantee exactly-once delivery from any upstream source but also with downstream platforms in some cases.

State management—The aforementioned trade-off between exactly-once delivery guarantee and the inevitable latency of state storage may drive the selection process based on the state management capabilities that come with the engine.

For example, Kafka Streams provides some stateful capabilities, the difference is that it doesn't provide a scheduler or full framework out-of-the-box. While it provides efficient ways of writing simple applications, you are left to your own devices on how to launch, run, orchestrate and operate those applications.

Kafka Streams is also good for things that are Kafka centric, because it tends to rely heavily on Kafka storage for state. Like Flink, it uses a local RocksDB but checkpoints the state as a Kafka topic and that limits flexibility as to how you store and access the history of that state. Within a Kafka ecosystem, a good linear access mechanism is provided, making everything nice and tame. This works great for simple use cases, but it doesn't provide quite the flexibility and operational capabilities as do some of the other engines.

Fault tolerance/resilience—The demand to mitigate operational disruption is so strong that the concept of “resiliency” attracts regulatory oversight across industries. Streaming architecture capabilities such as checkpointing, savepoints, redistribution, and [state management](#) are crucial to the stream processing engine selection process.

Spark Structured Streaming and Storm/Trident have built-in capabilities, while Kafka Streams requires you to “build your own”, using ZooKeeper to replace a failed broker for example.

Flink's fault tolerance mechanism uses checkpoints to draw consistent snapshots to which the system can fall back in case of a failure. The aforementioned state management capabilities ensure that even in the presence of failures, the program's state will eventually reflect every record from the data stream exactly-once.

The maturity of checkpointing in Flink provides a number of operational benefits, including bootstrapping new versions of jobs, moving them to other clusters, simplifying upgrades of applications and the clusters running them, and moving workloads between cloud and on-premises environments. This approach is one that is starting to be adopted by other stream processing engine communities.

Window semantics

Flink allows you to customize a window structure so you are not limited to pure linear time. You can define it by gaps between events or by the number of events using Session Windows for example. There is a lot of flexibility in how events are assigned to different windows prior to processing.

Windows are logically necessary to the analytics that you are likely to perform because it provides structure on which to base the analytic.

Two other windowing examples are tumbling windows (that slices a stream into even chunks) or sliding windows (that enable your aggregations and analytics to move with time as illustrated below).



Complex Event Processing (CEP)

To process real-time events and extract information from which to identify more meaningful events, like understanding behaviors for example, is probably one of the most interesting things you can do.

Flink’s statefulness and window handling capabilities is the foundation on which advanced CEP is crafted. What makes Flink all the more compelling is that CEP is accessible to a wider range of developer resources through standard SQL abstraction.

For example, the Match_Recognize SQL statement can be very helpful when you are looking for patterns built up through sequences of events that can’t be distinguished by simple counting methods.

The standard SQL abstraction of Flink makes it a compelling choice for use cases that require “simple” complex event processing.

Operational Considerations for Stream Processing Engine Evaluation

All organizations look to control costs by doing more with less. Budget approval for your selected stream processing engine may be contingent on its utility across streaming pipelines, reusability of talent, and synergising tech stacks.

Enterprise Adoption

The best application is no good if it can’t be efficiently and safely deployed across your organization or there is a dependency on hard to find development talent. Effective solutions are those that can be adopted across the entire enterprise.

Deployment model—There is a better chance of adoption if teams don’t have limited deployment options. Flink can be deployed in clustered, Kubernetes YARN, Kafka, Docker, S3, and microservices environments, while Structured Streaming, Kafka Streams, and Storm/Trident are more limited (see [Technical Features Table](#) on page 11). Kafka Streams is the most lightweight for microservices, at the cost of out-of-the-box features, and the Flink library is nearly as good.

Community maturity and documentation—To ensure that your developer resources are self-sufficient and productive, the maturity of the developer community and quality of documentation is a very important aspect to consider.

Storm/Trident is the oldest framework but the community has been eclipsed by the newer engines while activity has declined over the years. Kafka Streams is relatively young with very strong community growth and extensive documentation and examples.

While the Spark Structured Streaming community is large and busy with extensive documentation and examples, they could use more reviewers and committers, something that Cludera is helping to drive forward.

Flink is the fastest growing community with strong research and production deployments. Documentation and working examples are good and will broaden considerably as the community matures. As recently as 2019, Flink was the most active community by discussion, and the third most active by code commits. Also, some of the biggest brand name companies have already invested in large deployments of Flink for their real-time stream processing needs.

Enterprise Operations

If you are looking to establish a legacy of successful, fit for purpose data streaming solutions, it is important to know that you’ve selected an engine that completely integrates into your organization’s security framework, provides comprehensive monitoring and metrics, and can scale up and down in line with business demand.

Enterprise management—At Cludera, we’ve invested a good deal of our time to integrate the stream processing engines described in this paper into the [Cludera Data Platform \(CDP\)](#) to make sure that they are all enterprise ready for their respective purposes. Both Flink and Structured Streaming have rich Operations Support Systems (OSS) with enhanced vendor offerings. Kafka Streams has minimal OSS via some vendor offerings, while Storm/Trident has even less.

Over time, Cludera has enhanced Spark Structured Streaming with important metadata, logging, metrics, and operational capabilities that are also starting to find its way to Flink but, for now, the former has the edge on CDP.

As it relates to Kafka, the original security implementation was done by Cludera (via Hortonworks XASecure) and still provides leading security capabilities via scale-tested role-based and attribute-based access control models, and integrated data governance with [Apache Atlas](#) in CDP.

The other advantage of CDP for these stream processing engines is the fine-grained integrated single pane of glass security control.

Operational Efficiency

How would you understand what contributed to an unexpected value in a complex calculation while data continues to stream in?

Use the state processing API in Flink to recreate states as transactional snapshots and then dig in as much as needed to explain the origin and reasoning behind that dynamic calculation.

Checkpoints and savepoints are key to understanding how this works.

A Checkpoint provides a recovery mechanism in case of unexpected job failures. It creates a consistent image of the streaming job's execution state called a savepoint. You can use savepoints to stop-and-resume, fork, or update your jobs.

A fundamental aspect of Flink since its inception has been the separation of the state into local pieces that are linked together through consistent checkpointing levels. This minimizes latency but also provides all sorts of operational efficiency gain. For example, you can:

- Deploy new application versions that are preloaded from the current production state
- Do accurate A/B testing of new algorithms because you can easily bring up new instances against a solid starting point

To deal with supersets of data, the states saved locally in a high performance key value store (RocksDB) are checkpointed down to HDFS, a cloud blob store, or other durable repository.

Scaling up / Scaling down—Another consideration is that streaming workflows tend to be multi-modal and unbalanced throughout the day, so the scaling capabilities are absolutely crucial. Flink and Spark Structured Streaming are developing auto-scaling approaches to automatically maintain steady and predictable performance. They each have a solid orchestration platform underneath, which tends to give them an edge over the "build your own" type of approach that you get with Kafka Streams. Management tools help to scale Storm/Trident but tuning is challenging. If your application is tuned for "millions" then performance could be hurt for the "thousands".

Spark, Kafka, or Flink? Which to Use?

Boiling this down to high level guidance and decision making points, below is the heuristic that Cloudera tends to work with when advising customers.

REVIEW OF THREE MODERN STREAM PROCESSING ENGINES

		
<p>Guidance</p> <p>Spark Structured Streaming is best for developer accessibility and whole platform solutions where low latency and advanced streaming are not required, e.g. combining batch and stream where response time is measured in seconds to minutes.</p>	<p>Guidance</p> <p>Kafka Streams is best for Kafka Streams-only architectures without advanced streaming features</p>	<p>Guidance</p> <p>Flink is best for covering the full range of streaming pipeline requirements</p>
<p>Decision Making Points</p> <ul style="list-style-type: none"> • Spark Structured Streaming is already standard at your organization • You need a unified batch/stream solution • The highest levels of throughput is crucial • Low latency is not necessary • Advanced time/state features would be overkill 	<p>Decision Making Points</p> <ul style="list-style-type: none"> • You only need microservices • Throughput is essential • Low latency is crucial • Time/state features are not needed out-of-the-box • Application operational and resilience requirements are simple or handled elsewhere 	<p>Decision Making Points</p> <ul style="list-style-type: none"> • You need flexibility across microservices, batch and streaming • High throughput is necessary • Low latency is crucial • Use cases call for advanced windowing and state capabilities • You are not scared of new solutions, especially those that are best-in-class
<p>Both Flink and Kafka can be used as libraries in microservice architectures</p>		

Flink Use Cases

To get a practical understanding of how Flink is used in the real world, we have described a variety of use cases below.

Cybersecurity and Log Processing

A classic streaming data challenge is to identify and act upon intrusions and fraudulent events that are hidden within terabytes of dynamic machine logs. Throughput and latency are obviously important factors to consider because you want to identify an issue as quickly as possible. But effective action against criminals that doesn't alienate good customers requires an understanding of the behaviors of each.

The state management capabilities of Flink is a foundational component to cybersecurity solutions. From that we are able to leverage the performance benefits that comes from localized state when fulfilling enrichment functions and complex event processing. Session windows and watermarking supports deep dive investigations into data that is constantly moving.

Outage Classification for Telecom Companies

For a long time, telecom companies have focused on their network infrastructure and preventative maintenance but often as a reaction to past events. Customer and regulatory demands require a dynamic approach to predict and mitigate spotty performance and outages.

Adapting network strength and mass availability is crucial and that requires aggregated analysis on vast amounts of metrics over a wide array of networks to find anomalies, predict where failures are likely to occur, or even to just record the state of their current network at any point of time.

5G will only increase the volume and variety of metrics available, so having the ability to scale and perform analytics on those incoming events quickly is absolutely critical to identifying problems before customers do.

Financial Services: Mainframe Offloading

Consumers are demanding when it comes to speed of service. Overextended number crunching mainframes that are not meant for low latency end user interactions are often the bank's bottleneck, particularly with the advent of Open Banking directives. The solution has been to offload customer relationship functions from mainframes to stateful stream processing engines like Flink. Personalized product offerings based on real-time spending patterns is an example.

A key driver of success is data consistency. Flink's exactly-once delivery guarantee ensures correct accounting of spending behavior and balances. Combining that with complex event processing on the latest product and marketing data ensures that the right offer is being made at the right time. The fact that data access, enrichment, and decisioning is local, not external, enables millisecond level response time.

IoT for Manufacturing

IoT devices streamline supply chain operations within a manufacturing facility. Today, manufacturers are leveraging advanced monitoring sensors and real-time technologies to track quality of goods, automate the visual inspection of goods, and customized manufacturing for individual partners.

Flink's advanced windowing and state capabilities help to make the best use of sensors that collect data on machine health and productivity because, in order to diagnose and address problems before they occur, the sensor data needs to be aggregated and compared to previous data. Additionally, since IoT in manufacturing often has a diverse pipeline of use cases, the flexibility of Flink to be applied across microservices, batch, and streaming solutions is important.

Technical Features Table

The table below gives technical comparison across four modern stream processing engines. Refer to it when evaluating the functional and developmental aspects of your project.

TECHNICAL FEATURES				
	Flink 1.10	Kafka Streams 2.4	Spark Structured Streaming 2.4	Storm 2.0 and Storm Trident
Approach, position	<ul style="list-style-type: none"> Streaming first Modern class-leader 	<ul style="list-style-type: none"> Message-broker first Popular streaming add-on 	<ul style="list-style-type: none"> Batch first Popular streaming add-on 	<ul style="list-style-type: none"> Streaming first Legacy architecture
Streaming model, throughput, type	<ul style="list-style-type: none"> Stateful (First class requirement) <500 milliseconds Event-at-a-time 	<ul style="list-style-type: none"> Stateful <500 milliseconds Event-at-a-time 	<ul style="list-style-type: none"> Stateful Greater than one second Microbatch 	<ul style="list-style-type: none"> Natively stateless < 500 milliseconds Event-at-a-time with stateful plugins
Time support	<ul style="list-style-type: none"> Event time Processing time Customizable for greater control 	<ul style="list-style-type: none"> Event time Processing time 	<ul style="list-style-type: none"> Event time Processing time 	<ul style="list-style-type: none"> Event time Processing time
Processing abstractions	<ul style="list-style-type: none"> Table SQL (ANSI Standard) Complex event processing Graph Machine learning Batch (experimental) 	<ul style="list-style-type: none"> Table SQL-like DSL (KSQL) (not ANSI compliant) No batch 	<ul style="list-style-type: none"> Table ANSI SQL:2003 in Spark Structured Streaming 2.3 Graph Machine learning Unified APIs for batch and stream 	<ul style="list-style-type: none"> Streaming only out-of-the-box SQL- Experimental since 1.2.3
Delivery guarantee	<ul style="list-style-type: none"> Upstream: Exactly-once Downstream: Some capabilities depending on the downstream system 	<ul style="list-style-type: none"> Upstream: Exactly-once Kafka Streams only Downstream: No 	<ul style="list-style-type: none"> Upstream: Exactly-once Downstream: Some capabilities depending on the downstream system 	<ul style="list-style-type: none"> Upstream: <ul style="list-style-type: none"> At-least-once (out-of-the-box) Exactly-once (with Trident) Downstream: No
State management	<ul style="list-style-type: none"> RocksDB Configurable snapshots Queryable 	<ul style="list-style-type: none"> BYO RocksDB Snapshot to Kafka Streams topic Queryable 	<ul style="list-style-type: none"> RocksDB Databricks only OSS sync on HDFS 	<ul style="list-style-type: none"> External, not out-of-the-box, at-least-once processing in stateful operations
Fault tolerance and resilience	<ul style="list-style-type: none"> Built-in Checkpoints Savepoints Redistributable 	<ul style="list-style-type: none"> BYO Microservice 	<ul style="list-style-type: none"> Built-in 	<ul style="list-style-type: none"> Built-in

■ Great fit for purpose
 ■ Fits with some work
 ■ Fits with a lot of work
 ■ Not fit for purpose

Operational Features Table

The table below gives an operational comparison across four modern stream processing engines. Refer to it when evaluating the nonfunctional aspects of your project.

OPERATIONAL FEATURES				
	Flink 1.10	Kafka Streams 2.4	Spark Structured Streaming 2.4	Storm 2.0 and Storm Trident
Deployment model	<ul style="list-style-type: none"> • Clustered • Kubernetes • YARN • Kafka • Docker • S3 • Microservices 	<ul style="list-style-type: none"> • Not clustered • Kubernetes • Microservices 	<ul style="list-style-type: none"> • Clustered • Kubernetes 	<ul style="list-style-type: none"> • Clustered
Documentation	<ul style="list-style-type: none"> • Good technical documentation • Growing examples • Stack Overflow coverage 	<ul style="list-style-type: none"> • Extensive documentation • Extensive examples • Stack Overflow coverage 	<ul style="list-style-type: none"> • Extensive documentation • Extensive examples • Stack Overflow coverage 	<ul style="list-style-type: none"> • Good documentation for 1.x
Maturity/community	<ul style="list-style-type: none"> • Smaller but fastest growing community with strong research and production deployments 	<ul style="list-style-type: none"> • Newest, strong community with strong growth 	<ul style="list-style-type: none"> • Spark Structured Streaming community is strong, but Streaming is a small, quiet corner 	<ul style="list-style-type: none"> • Oldest framework, community eclipsed by newer engines
Use cases	<ul style="list-style-type: none"> • Unbounded and bounded streams • Batch • Complex event processing • IoT • Microservices • Others 	<ul style="list-style-type: none"> • Microservice/event driven, embedded in another application 	<ul style="list-style-type: none"> • Unified ETL, semi-RT processing 	<ul style="list-style-type: none"> • IoT, complex event processing
Enterprise management	<ul style="list-style-type: none"> • Rich OSS • Enhanced vendor offerings 	<ul style="list-style-type: none"> • Minimal OSS • Some via vendor offerings 	<ul style="list-style-type: none"> • Rich OSS • Enhanced vendor offerings 	<ul style="list-style-type: none"> • Some integrations
Push button security	<ul style="list-style-type: none"> • Complex • Some OSS support • Limited vendor offerings 	<ul style="list-style-type: none"> • Simple, some OSS support, good vendor offerings 	<ul style="list-style-type: none"> • Complex • Good OSS support • Good vendor offerings 	<ul style="list-style-type: none"> • Complex • Good OSS support • Good vendor offerings
Logging/metrics	<ul style="list-style-type: none"> • Usual OSS integrations, some vendor offerings 	<ul style="list-style-type: none"> • BYO microservices 	<ul style="list-style-type: none"> • Good logging integration 	<ul style="list-style-type: none"> • Good logging integration
Scaling up/down	<ul style="list-style-type: none"> • Not yet autoscaling, but all requirements available 	<ul style="list-style-type: none"> • BYO microservice, scaling limits (e.g. shuffle sort) 	<ul style="list-style-type: none"> • Not yet autoscaling, but all requirements available 	<ul style="list-style-type: none"> • Management tools help but tuning is challenging

■ Great fit for purpose
 ■ Fits with some work
 ■ Fits with a lot of work
 ■ Not fit for purpose

About Cloudera

At Cloudera, we believe that data can make what is impossible today, possible tomorrow. We empower people to transform complex data into clear and actionable insights. Cloudera delivers an enterprise data cloud for any data, anywhere, from the Edge to AI. Powered by the relentless innovation of the open source community, Cloudera advances digital transformation for the world's largest enterprises.

Learn more at cloudera.com

Connect with Cloudera

About Cloudera DataFlow:
cloudera.com/cdf

Join the Cloudera Community:
community.cloudera.com

Read about our customers' successes:
cloudera.com/customers

Customer Success

To provide insights into the business impact that can be drawn through a comprehensive data-in-motion solution, we provide two customer success examples.

1. **An international communications company** serving consumers and businesses in ten countries, deployed the Cloudera streaming data platform to tackle a variety of critical use cases including, stream processing, log aggregation, and large-scale messaging and customer insights.

Results included improved overall customer experience through strategic use of data analysis, reduced infrastructure management costs and TCO, and enabled real-time actions to improve business outcomes.

2. **A large European bank** specialising in agriculture financing and sustainability oriented banking across global markets leveraged Cloudera's streaming data platform to run sophisticated real-time algorithms and financial models to help customers manage their financial obligations, including loan repayments.

By implementing the platform and gaining the ability to stream real-time data, the bank can now detect warning signals in extremely early stages of where clients may go into default. Through their new, governed data lake, the bank's account managers are also able to access an in-depth overview of customer data, enabling them to generate liquidity overviews and advise customers on how to avoid defaulting. Through rapid data processing, better models are created that more accurately predict warning signals.

Ensure Fit for Purpose and Enterprise Wide Adoption

Streaming and time based reasoning applications are confronted with both simple and complex sets of challenges. Functional business requirements determine how data should be processed and that, in turn, helps to evaluate which of a number of stream processing engines suffice your requirements.

This paper described a number of capabilities that would address the most complex of challenges and handle simple scenarios, while keeping in mind acceptable trade-offs. You don't want to over-engineer a solution but you want to know that it can grow to support an evolving business. To support that growth, there are a number of technical and operational factors that are crucial to the decision making process.

We also suggested that you take a broad perspective that also considers nonfunctional aspects such as how your team can deliver on the solution's promise, how it integrates into your organization's security framework, operational processes, support structure, and how it can scale up and down in line with business demand.

In summary, the view expressed here will help ensure that you choose the right stream processing engine that is both fit for purpose to the business challenge at hand, and will also enjoy enterprise wide adoption.