# How to Choose a Stream Processor for Your App

Leia em portuguÃªs

This item in chinese

## Key Takeaways

- Choosing a stream processor is challenging because there are many options to choose from and the best choice depends on end-user use cases.
- Streaming SQL provides significant benefits with quicker application development times and highly maintainable deployments.
- Query writing environments significantly affect developer productivity, which demands advanced graphical editors and application debuggers for stream processors.
- If a system needs a throughput of less than 50K events/second, you could have major savings with a two node High Availability (HA) deployment.
- If the event rate is beyond a single stream processor node, you should place incoming events into a message broker and process events with snapshots enabled.

Stream Processors are software platforms that enable users to respond to incoming data streams faster (see What is Stream Processing?).

Streaming applications which run on stream processors come in many forms.

The following are some of the examples:

1. Detect a condition and generate an alert (e.g., track the temperature of a kitchen appliance and create an alert if it exceeds a pre-defined threshold)
2. Calculate a running average location of a moving object and update a web page (e.g., detect the location of a person and plot his trajectory on a map)
3. Detect anomalies and act on them (e.g., detect a suspicious user and carry out a detailed analysis of his actions)

If you are curious about other applications, the blog post 13 Stream Processing Patterns for Building Streaming and Real-Time Applications discusses more use cases.

As outlined in the Quora Question "What are the best stream processing solutions out there?", there are many stream processors available to choose from.

The choice depends on the use case, and you should choose a stream processor that best matches your use case.

This article discusses how to make the choice.

Let's approach this problem in three steps. We'll first discuss a reference architecture; anatomy of a streaming application.

Then we will discuss key features required by most streaming applications.

Finally, we will list optional features that can be selected based on the use case.

## Reference Architecture for Streaming Applications

A streaming application needs three things: data streams, a processor to process the data, and code to do something with the decisions (See Figure 1).

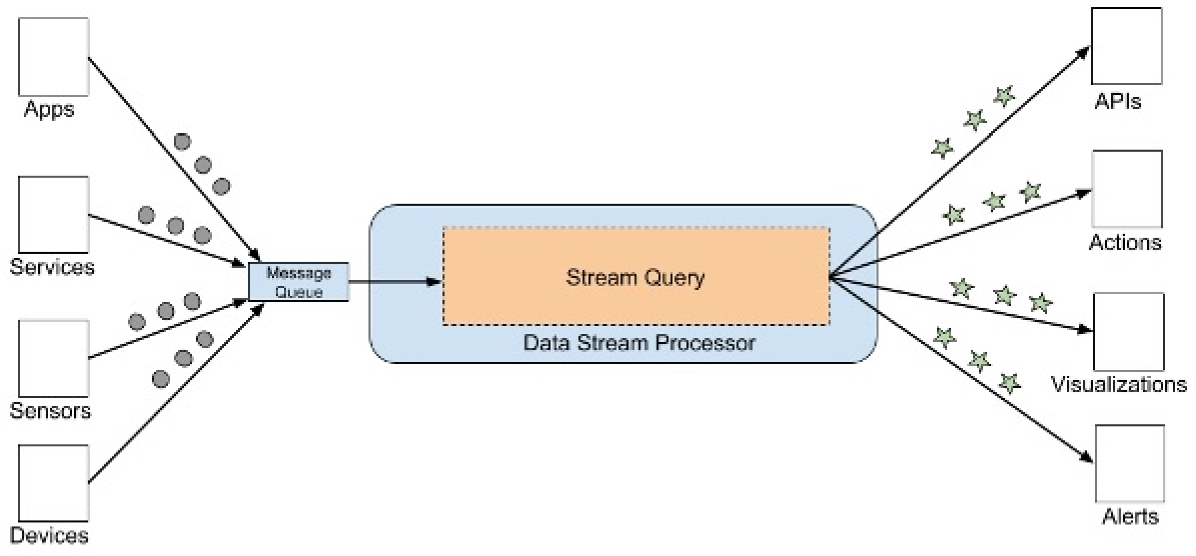


**Figure 1: Reference architecture for streaming applications**

First, collect all data streams received from the sources into message queues of the broker. Unless you have specific requirements that warrant a different design, we recommend that you place messages into a message queue and reading messages from the message queue. This enables you to replay events if needed and simplify High Availability (HA) and fault tolerance.

The stream processor fetches events from the message queue, sends them to the stream query, which processes the data and produces the results. Most stream processors help to act on these results by generating alerts, exposing or invoking APIs, carrying out actions, and exposing visualizations. For example, let's take the first scenario mentioned in the Introduction section, which monitors room temperature to detect anomalies of energy usage. The application detects and notifies abnormalities via email alerts. Figure 2 shows the application data flow graph for this use case.

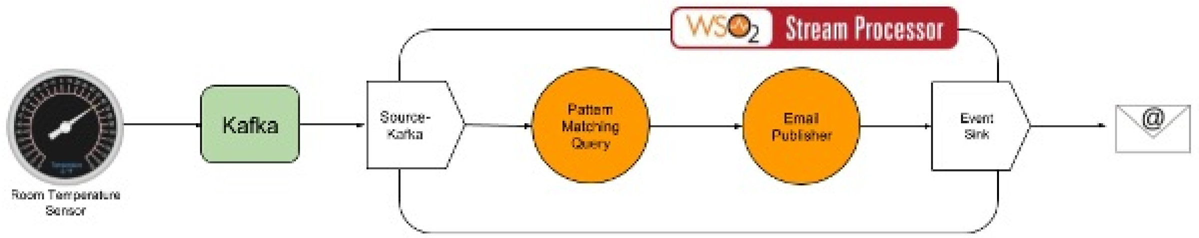


**Figure 2: Architecture of a stream processing application for detecting abnormal increase of room temperature.**

When selecting a stream processor, you need to consider two kinds of features: must-have and good-to-have features. As the name suggests, must-have features are needed. Even if you do not use them now, chances are you will need them soon. You can pick and choose good-to-have features based on your requirements. This article will primarily focus on the must have features.

## Must-Have Features

You should make sure the chosen stream processor supports all of the following features.

### Support for Data Ingestion with a Message Broker

While developing your app, the very first question you face is, “How does my app receive data from external sources?”  The answer is to use a message broker and make sure your stream processor can do this. Most of them do. The following are some of the advantages of using a broker:

1. Messages are stored persistently right away.
2. The broker would become your HA endpoint. The rest of the system doesn’t need HA.
3. If something goes wrong, you can go back and replay the messages from the message broker.

4. Scalable message brokers such as Kafka already handle the complexities of scalability.

To explore the merits of a message broker, please refer to articles Questioning the Lambda Architecture, and The Log: What every software engineer should know about real-time data's unifying abstraction.

## Streaming SQL

The first generation of streaming engines such as Apache Storm and Apache Spark required users to write code. Users would write code and place them inside an agent (sometimes called an actor). The streaming engine lets users wire these agents together and ingest events.

Although this is a great way to get started, it requires users to write code. This leads to duplicated code at multiple places which leads to increased maintenance cost.

Imagine you need to get data out of a database; you'll need to write code describing how to find data. Writing code for stream processing is no better. Batch processing has already moved away from writing code and instead supports querying them via SQL. We can do the same with streaming analytics. Such query language is called Streaming SQL.

The following are some advantages of a streaming SQL language:

- It's easy to follow and easy to hire many developers who already know SQL.
- It's expressive, short, sweet and fast.
- It defines core operations that cover 90% of problems.
- Streaming SQL language experts can implement application specific custom analytics when they like by writing extensions.
- A query engine can better optimize the executions with a streaming SQL model.

With Streaming SQL, users can query data without having to write code. The platform handles data transfers, data parsing, and also provides operators such as joins, windows, and patterns directly in the language. The Listing 1 shows the Streaming SQL code for the aforementioned anomaly detection application.

**Listing 1: Application for high room temperature anomaly detection**

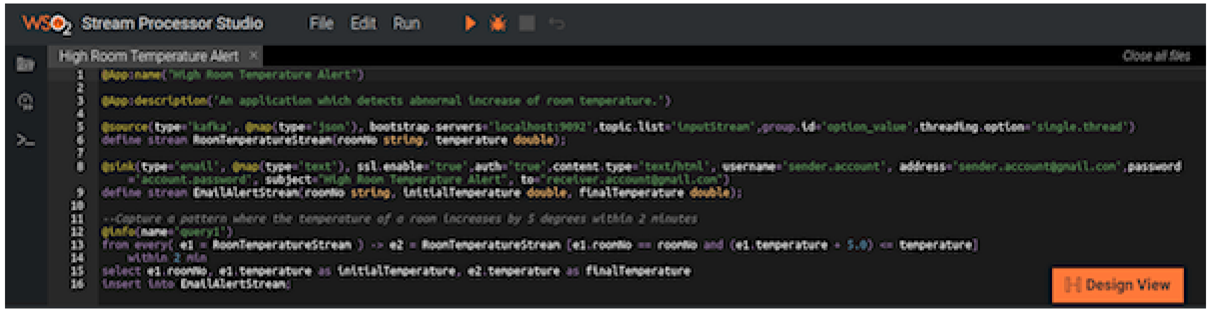
```
@App:name("High Room Temperature Alert")

@App:description('An application which detects abnormal increase of room temperature.')

@source(type='kafka', @map(type='json'), bootstrap.servers='localhost:9092',topic.list='inputStream',gr
define stream RoomTemperatureStream(roomNo string, temperature double);

@sink(type='email', @map(type='text'), ssl.enable='true',auth='true',content.type='text/html', username
define stream EmailAlertStream(roomNo string, initialTemperature double, finalTemperature double);

--Capture a pattern where the temperature of a room increases by 5 degrees within 2 minutes
@info(name='query1')
from every( e1 = RoomTemperatureStream ) -> e2 = RoomTemperatureStream [e1.roomNo == roomNo and (e1.tem
    within 2 min
select e1.roomNo, e1.temperature as initialTemperature, e2.temperature as finalTemperature
insert into EmailAlertStream;
```

If a stream processor does not support Streaming SQL, developing your streaming app takes more time. For example, if you were to develop the same app shown in Listing 1 with Java, you would have ended up spending a significant amount of time writing code to detect patterns. Furthermore, once deployed in production, maintaining such application is very expensive. A streaming app would need several operators such as transformation, aggregation/correlation, window, and pattern matching. You have to write those algorithms from scratch. To learn more about streaming SQL, please check out the article on Stream Processing 101: from SQL to Streaming SQL.

## Stream Processing APIs & Query Writing Environment

What tools does your stream processor provide you with to develop your app? Most well-known stream processors provide an editor to author queries, either visually or textually. The following picture (see Figure 3) shows a query authored with WSO2 Stream Processor, an open source stream processor released with Apache 2.0 license. Such an editor supports visual error messages and auto-completion (see Figure 3). Streaming SQL is a powerful but delicate language. Being able to see the output while writing queries is useful. The first level is to be able to attach an event archive to the editor, replay them and see the outputs having written a query. The second level is to have outputs change while changing the query.

**(Click on the image to enlarge it)**



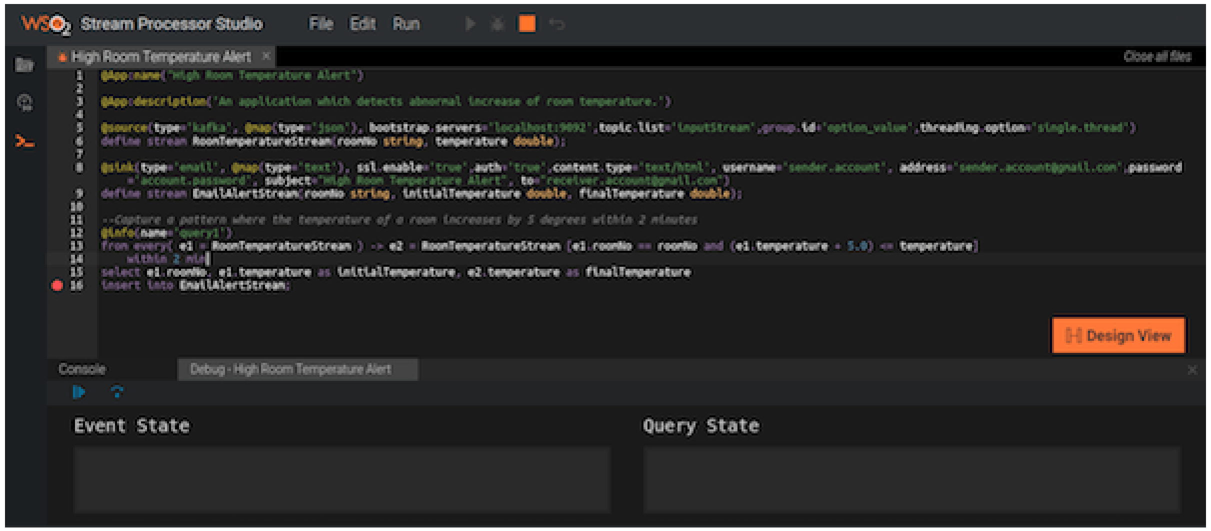
```
@App:name('High Room Temperature Alert')

@App:description('An application which detects abnormal increase of room temperature.')

@source(type='kafka', @map(type='json'), bootstrap.servers='localhost:9092',topic.list='inputStream',group.id='option_value',threading.option='single.thread')
define stream RoomTemperatureStream(roomNo string, temperature double);

@sink(type='email', @map(type='text'), ssl.enable='true',auth='true',content.type='text/html', username='sender.account', address='sender.account@gmail.com',password
    ='account.password', subject='High Room Temperature Alert', to='receiver.account@gmail.com')
define stream EmailAlertStream(roomNo string, initialTemperature double, finalTemperature double);

--Capture a pattern where the temperature of a room increases by 5 degrees within 2 minutes
@info(name='query1')
from every( e1 = RoomTemperatureStream ) -> e2 = RoomTemperatureStream [e1.roomNo == roomNo and (e1.temperature + 5.0) <= temperature]
    within 2 min
select e1.roomNo, e1.temperature as initialTemperature, e2.temperature as finalTemperature
insert into EmailAlertStream;
```

**Figure 3: Stream Processor Studio showing the Code View of the high room temperature alert application**

Almost all the stream processors have some application debugging support. However, the depth of debugging support varies. Some debuggers allow you to put breakpoints and inspect the intermediate variable values and trace (See Figure 4). Others provide event logs. Some debuggers provide visibility into metrics such as event flow count between operators.

Not having any debugging support at all makes it very difficult to investigate the stream application's behavior. You should choose a stream processor that has extensive debugging support. That saves many hours during query authoring and maintenance.

**(Click on the image to enlarge it)**



WSO2 Stream Processor Studio
File
Edit
Run
High Room Temperature Alert
Close all files
Design View
Console
Debug - High Room Temperature Alert
Event State
Query State

```
@App:name('High Room Temperature Alert')

@App:description('An application which detects abnormal increase of room temperature.')

@source(type='kafka', @map(type='json'), bootstrap.servers='localhost:9092',topic.list='inputStream',group.id='option_value',threading.option='single.thread')
define stream RoomTemperatureStream(roomNo string, temperature double);

@sink(type='email', @map(type='text'), ssl.enable='true',auth='true',content.type='text/html', username='sender.account', address='sender.account@gmail.com',password
    ='account.password', subject='High Room Temperature Alert', to='receiver.account@gmail.com')
define stream EmailAlertStream(roomNo string, initialTemperature double, finalTemperature double);

--Capture a pattern where the temperature of a room increases by 5 degrees within 2 minutes
@info(name='query1')
from every( e1 = RoomTemperatureStream ) -> e2 = RoomTemperatureStream [e1.roomNo == roomNo and (e1.temperature + 5.0) <= temperature]
    within 2 min
select e1.roomNo, e1.temperature as initialTemperature, e2.temperature as finalTemperature
insert into EmailAlertStream;
```

**Figure 4: Debugging of the high room temperature alert application with Stream Processor Studio**

Some stream processors also include drag and drop type Graphical User Interfaces (GUIs) (see Figure 5). The drag and drop GUI may provide a toolbox containing the elements that can be dropped and a two-way scrollable canvas to drop the elements. The toolbox may contain icons for defining streams, operators such as windows, join, and filter. Once the user adds the streaming operators on the screen using drag-n-drop, they can modify the properties of each stream/operator by using the operators' settings wizard.

**(Click on the image to enlage it)**

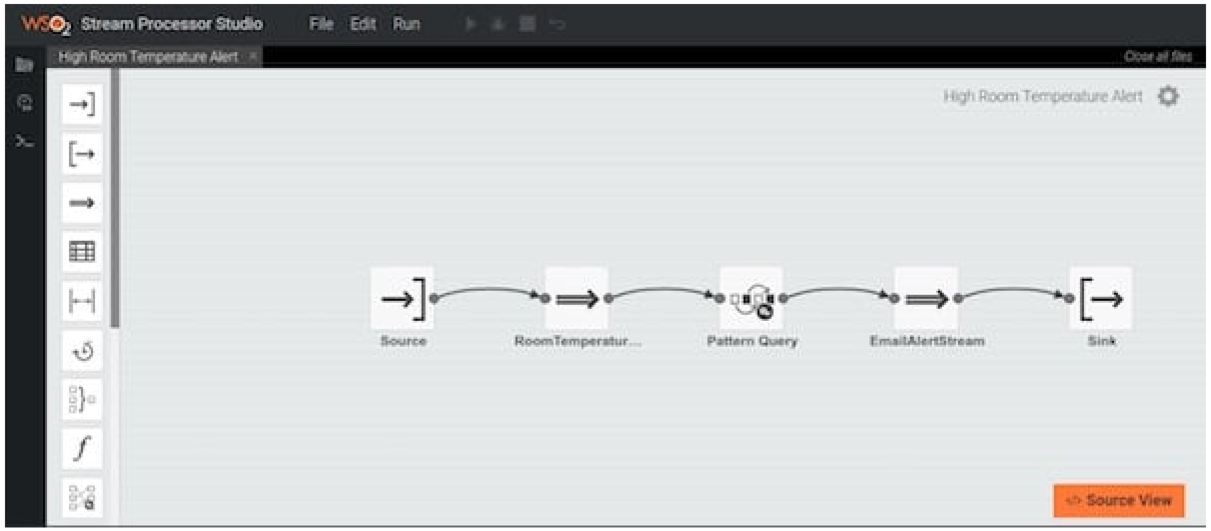


**Figure 5: Stream Processor Studio's Graphical Editor showing the application data flow graph of high room temperature alert application**

While they make great demos, it is not clear whether drag and drop interface is better for building apps. For example, with SQL, such an interface has never been used widely. SQL query authoring is done by writing SQL directly. It is a myth that business users who do not understand programming can write queries with drag and drop toolbox. Even such users need to understand programming to go beyond the basics.

## Reliability, High Availability (HA), and Minimal HA

What happens if your system suddenly crashes? Stream processing applications run forever and never stop. Hence, if your application is a stateful application, it will eventually lose valuable information (e.g., state) due to system failures. We call the ability to recover from failure as “reliability,” and we call the ability to continue operations with minimal interruption as High Availability (HA).

### State Management

Most streaming queries are stateful. Where to keep and how to access the state? The state is the information the stream processor remembers between processing two events.

The state has three types: Application state, User state, and System state.

Application state refers to the values being created and maintained while running the application. An example is a state required to detect conditions, such as patterns or content of a length window. Application state resides in the ephemeral store (see Figure 6), such as the main memory and periodically gets flushed to the permanent store.

User state is the user data which is accessed by the application to make runtime decisions. For example, a permanent store such as an RDBMS can contain information of user’s credit history.

System state refers to everything else the framework provides to make sure that if the stream processor crashes, it is recoverable back to its normal operations.

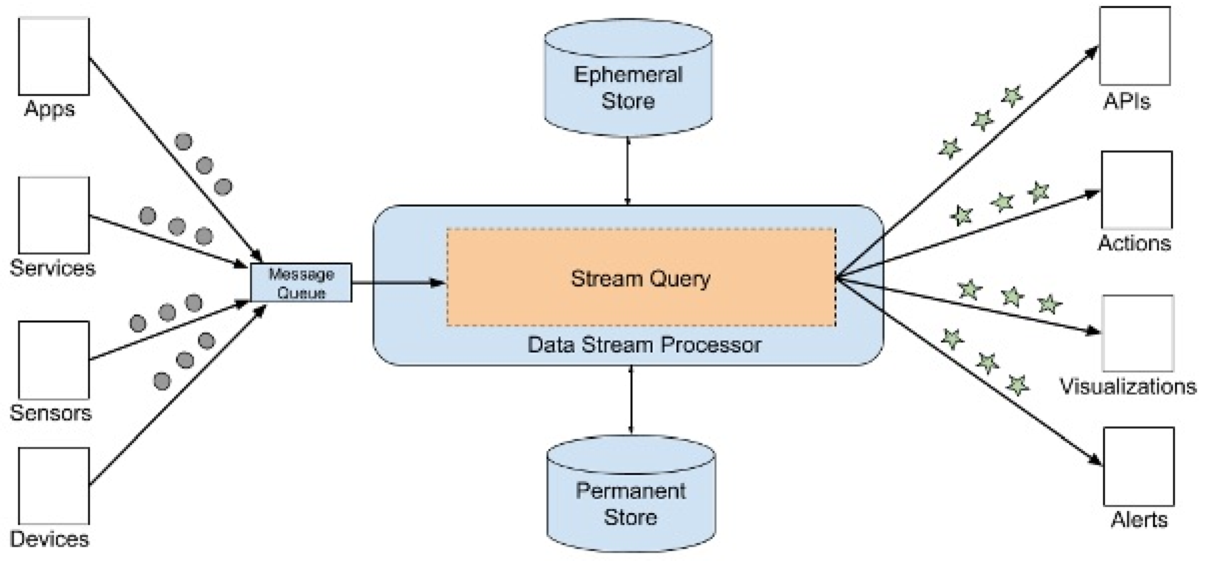


**Figure 6: Data stream processor with data stores**

Highly available stream processors need reliable and fault tolerant state management to avoid losing its state. User state is stored directly in a persistent store. Stream processors recover application and system state through active-warm deployments, snapshotting, or recomputing.

After a failure, recomputing replays the events from the last known good state to continue the executions. However, if the application is stateful, the last known good state would be beginning. This often leads to replaying and reprocessing a large number of events.

To avoid have to replay large number of events, the stream processor can take periodic snapshots. Then, it can restore the state to the snapshot and replay events from that snapshot.

## Minimal HA deployment

With many stream processors, a single node can handle an event rate of over 50,000 events/sec. The highest throughput required by most of the stream processing use cases stays well below 50,000 events/sec. Often those use cases do not need to scale beyond two nodes, in which case, you could have significant savings by the deploying active-warm deployment described above. For example, see the article “Is Your Stream Processor Obese?”.

## Back Pressure

What happens if your application receives more events than it can handle? Within stream processors, the back-pressure maintains the system’s stability, refusing to accept excess events.

If correctly done, the back pressure needs to be built into each level of the stream processor. Back-pressure transfers the burden of a bottleneck back to event sources, avoiding queue overflows and out of memory errors. Since the stream processor stops accepting new events from the external system, external systems may have to buffer the data or even discard them when the buffers become full.

If dynamic scaling is available, the system may auto-scale instead of backpressure. However, no system scales indefinitely, and when it hits its limits, it needs to employ backpressure.

This is a critical feature to look for when you are choosing a stream processor.

## Reliable HA Recommendations

To choose a reliable and HA model, you need to deliberate carefully.

If the event rate you need to handle is within a capacity of a single stream processor node, which is typically higher than 50,000 events/sec, we recommend the following deployment. Place incoming events in a message broker and then deploy two stream processor nodes in active-warm formation to consume events from a message broker using a topic. The stream processor needs to be able to detect failure of the active node and switch.

If your event rate is beyond a single stream processor node, you should place incoming events in a message broker and process events with snapshots enabled. If a failure happened, the stream processor can restore the state using a snapshot and replay events from the point of the snapshot.

## Optional Features

The former section discussed the essential features most streaming applications would need. In contrast, the following are a list of some optional features that are needed only by some applications. You should look for them only if your application needs them:

- Business user friendliness via drag and drop type Graphical User Interfaces (GUIs)
- Streaming Machine Learning
- Optional Reliability Features
  - Message processing guarantee
  - Out-of-order events
- Large Scale System Performance
  - Scalability
  - Handle large windows

Due to limited space,  this article does not discuss the optional features, and we plan to cover them in future articles.

## Conclusion

By nature, different stream processors matches with different use cases. While trying to select the stream processor best for you, you must consider many aspects in order to make the right choice.

This article discusses a reference architecture for stream processing and presents a systematic approach for choosing a stream processor. The approach answers two main questions. First, to what extent does the stream processor support the core stream processor architecture features? Second, what are the special requirements of the application, and to what extent are those are being satisfied by the candidate stream processors? The former was the focus of this article and latter will be discussed in detail in future articles. Figure 7 shows a categorization of the features for formulating the answers.

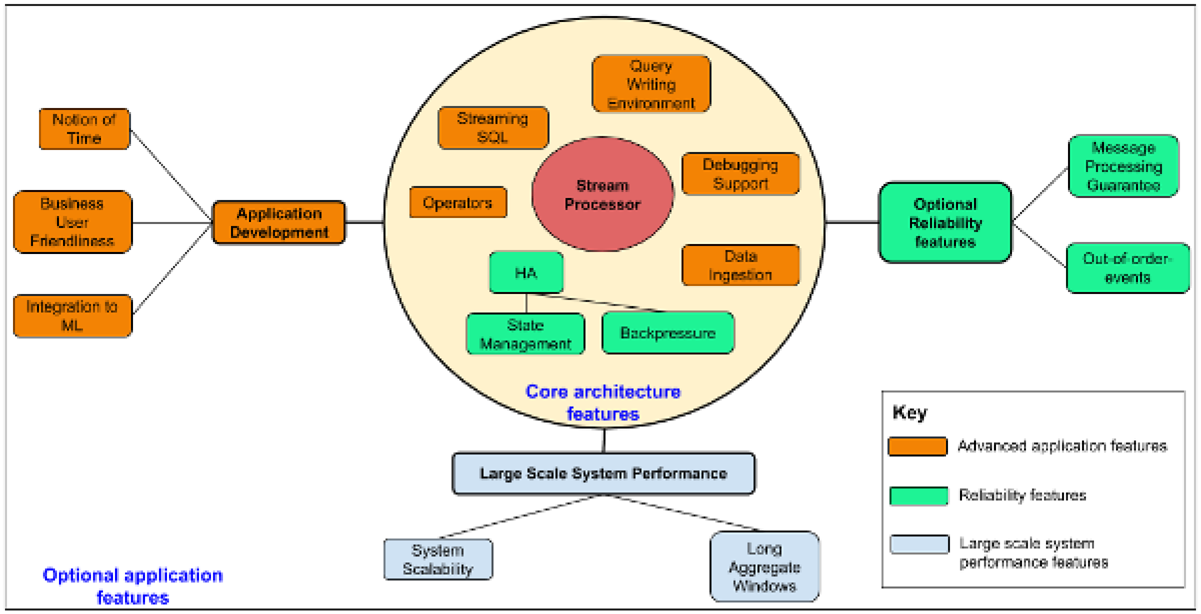

Notion of Time
Business User Friendliness
Integration to ML
Application Development
Streaming SQL
Query Writing Environment
Stream Processor
Debugging Support
Operators
HA
Data Ingestion
State Management
Backpressure
Core architecture features
Optional Reliability features
Message Processing Guarantee
Out-of-order events
Large Scale System Performance
System Scalability
Long Aggregate Windows
Optional application features
Key
Advanced application features
Reliability features
Large scale system performance features

**Figure 7: Categorization of the features used for answering the two key questions**

The article discussed each core feature in detail and why they are important, while providing guidelines on how to choose a stream processor that best matches the nature of the application.

## About the Authors

**Miyuru Dayarathna** is a senior technical lead at WSO2. He is a computer scientist with multiple research interests and contributions in stream processing, graph data management and mining, cloud computing, performance engineering, IoT, etc. He is also a consultant at the Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka. He has published technical papers in reputed international journals and conferences as well as organized several international workshops on high performance graph data management and processing.

**Srinath Perera** is a scientist, software architect, and a programmer who works on distributed systems. He is a member of the Apache Software Foundation. He is a key-architect behind several widely used projects such as Apache Axis2, WSO2 Stream Processor. Srinath has authored two books about MapReduce and frequent author of technical articles. He received his Ph.D. from Indiana University, USA, in 2009. He has been involved with the Apache Web Services project since 2002, and he is a committer on several Apache open-source projects, including Apache Axis, Axis2, and Geronimo.

### Related Editorial

- **Democratizing Stream Processing with Apache Kafka® and KSQL - Part 2**
- **Distributed Messaging Framework Apache Pulsar 2.0 Supports Schema Registry and Topic Compaction**
- **Event Sourcing to the Cloud at HomeAway**

- **Apache Kafka: Ten Best Practices to Optimize Your Deployment**

- **William McKnight on Data Platforms and Creating a Modern Data Architecture**

2