Session 8

Data Stream Processing & Complex Event Processing Systems and Performance

Big Data Analytics Technology, MSc in Data Science,
Coventry University UK

Miyuru Dayarathna

Presentation Outline

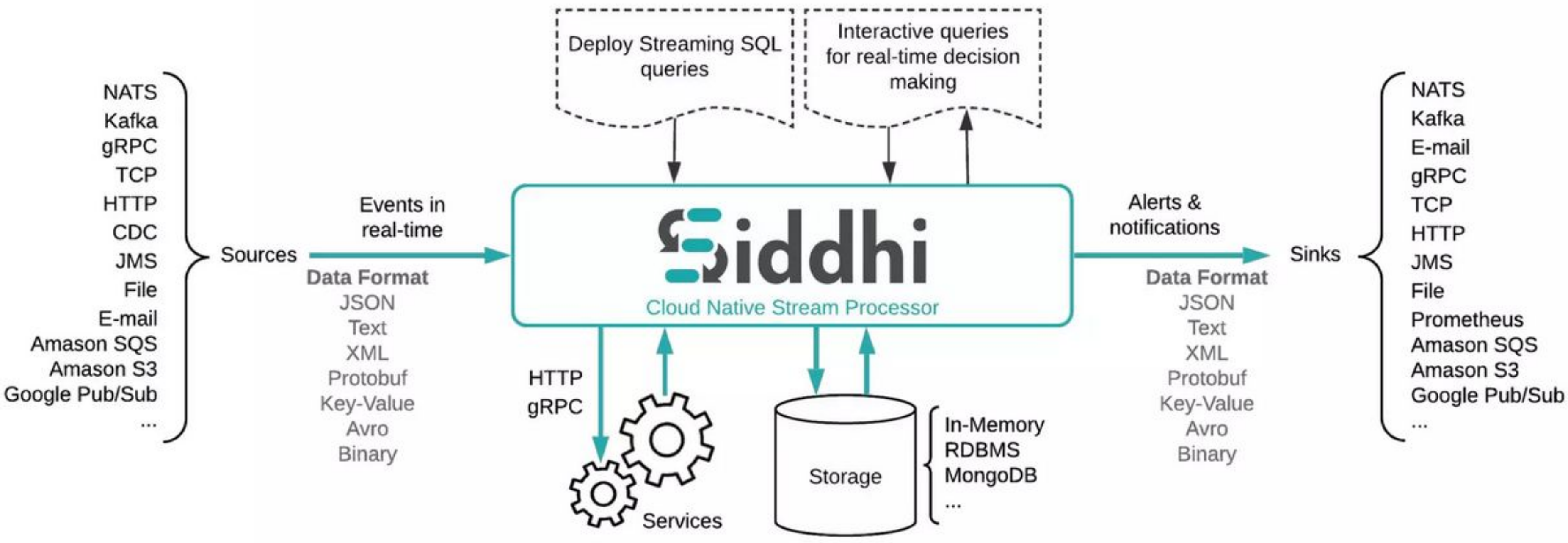
- Siddhi
- Apache Flink
- Performance Benchmarking
- Conclusion

Siddhi Complex Event Processor

- Siddhi is a cloud native stream processor having the following features,
  - Light weight (Low memory footprint and quick startup)
  - 100% open source
  - Native support for Docker and Kubernetes
  - Support Agile Devops workflows and full CI/CD pipeline
  - Allow event processing logics be written in SQL like query language and via graphical tool
  - Single tool for data collection, ingestion, processing, analysis, integration (with services and databases), and to manage notifications

https://www.slideshare.net/Suhothayan/siddhi-cloudnative-stream-processor?from_search=2

Siddhi Complex Event Processor (Contd.)



Siddhi Complex Event Processor - Key features

- Native distributed deployment in Kubernetes
- Native CDC support for Oracle, MySQL, MSSQL, Postgres
- Log running aggregations from seconds to years
- Complex pattern detection
- Online machine learning
- Synchronous decision making
- DB integration with caching
- Service integration with error handling
- Multiple built-in connectors (file, Kafka, NATS, gRPC, ...)

Scenarios and use cases supported by Siddhi

- Notification management
- Streaming data integration
- Fraud detection
- Stream Processing at Scale on Kubernetes
- Embedded Decision Making
- Monitoring and Time Series Data Analytics
- Realtime policy enforcement engine
- IoT, Geo, and Edge analytics
- Real time Decision as a Service
- Real time Predictions with Machine Learning

Working with Siddhi

- Develop apps using Siddhi Editor
- CI/CD with build integration and Siddhi Test Framework
- Running Modes
  - Embedded in Java/Python apps
  - Microservice in bare metal/VM
  - Microservice in Docker
  - Microservice in Kubernetes

Practical Session

https://siddhi-io.github.io/PySiddhi/Installation-Guide/

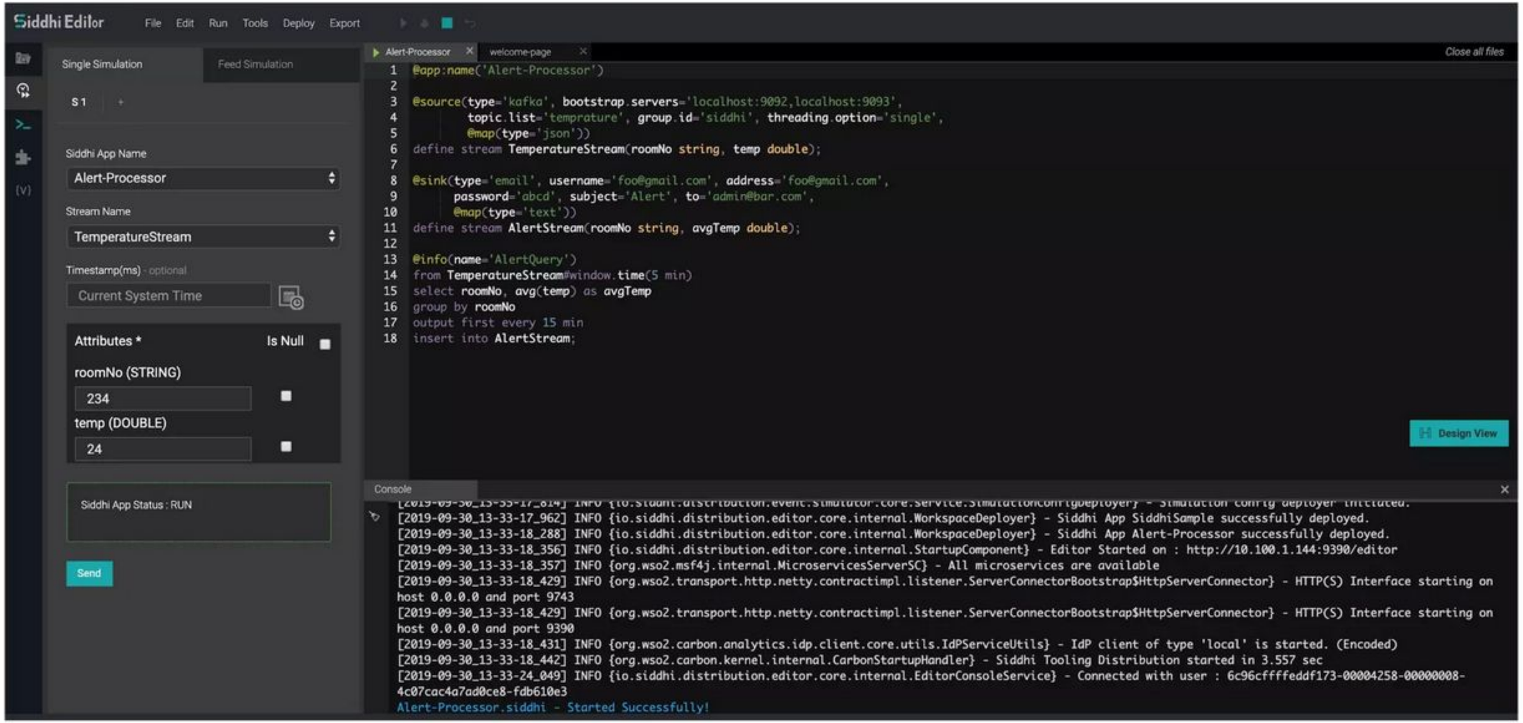
Streaming SQL

```
@app:name('Alert-Processor')

@source(type='kafka', ..., @map(type='json'))
define stream TemperatureStream (roomNo string, temp double);

@info(name='AlertQuery')
from TemperatureStream#window.time(5 min)
select roomNo, avg(temp) as avgTemp
group by roomNo
insert into AvgTemperatureStream;
```

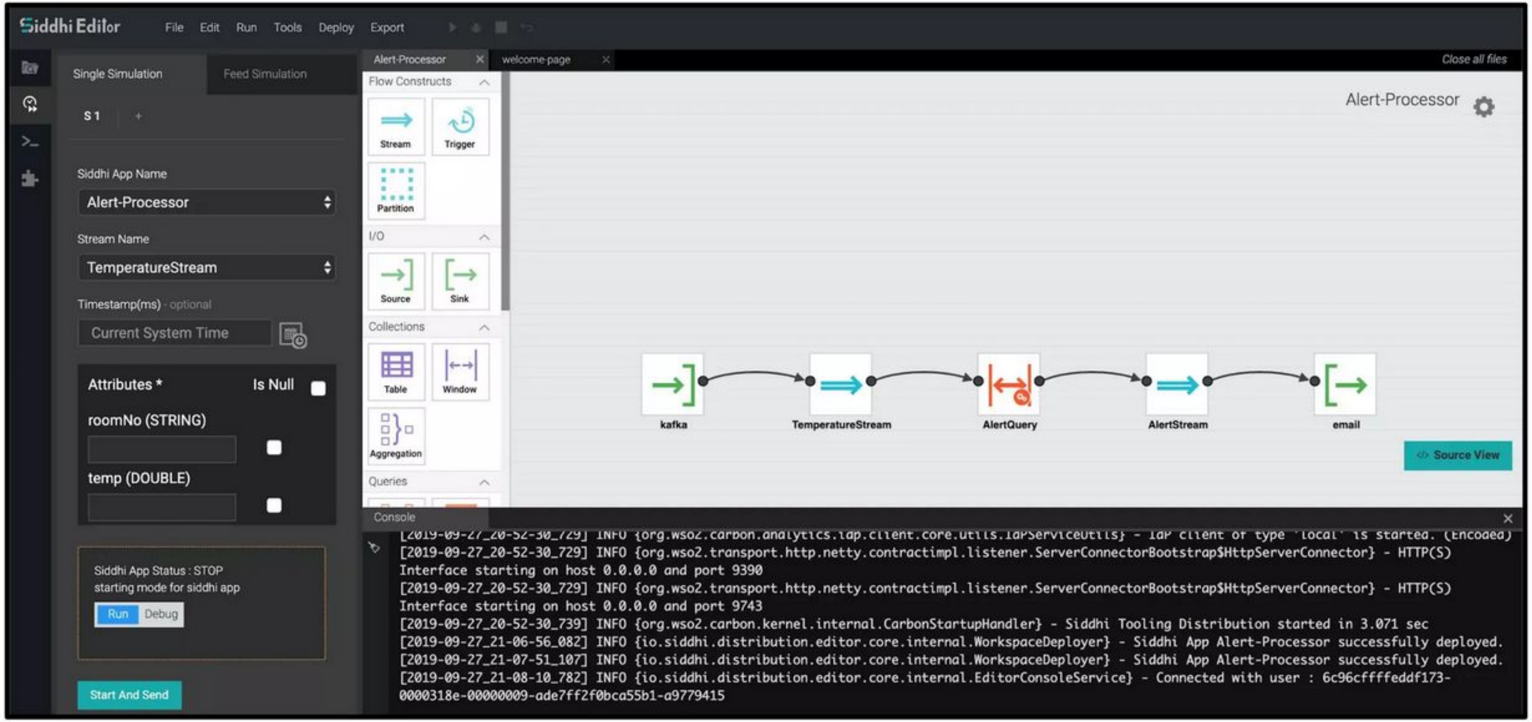
Source/Sink & Streams

Window Query
with Rate Limiting

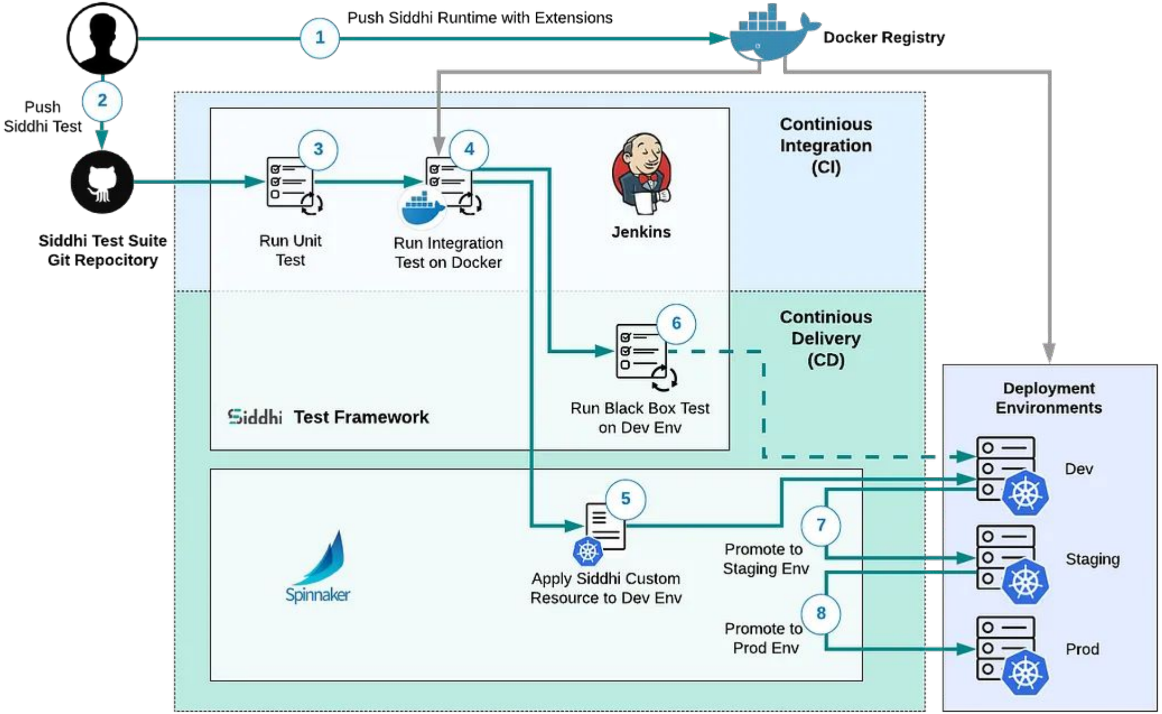
Web based source editor



Web based graphical editor



Reference CI/CD Pipeline of Siddhi



https://medium.com/siddhi-io/building-an-efficient-ci-cd-pipeline-for-siddhi-c33150721b5d

Supported Data Processing Patterns

- Consume and publish events with various data formats
- Data filtering and preprocessing
- Date transformation
- Database integration and caching
- Service integration and error handling
- Data Summarization
- Rule processing
- Serving online and predefined ML models
- Scatter-gather and data pipelining
- Real time decisions as a service (On-demand processing)

Scenario : Order Processing

- Customers place orders
- Shipments are made
- Customers pay for the order

Tasks :

- Process order fulfillment
- Alerts sent on abnormal conditions
- Send recommendations
- Throttle order requests when limit exceeds
- Provide order analytics over time

Consume and Publish Events with various Data formats

- Supported transports
  - NATS, Kafka, RabbitMQ, JMS, IBMMQ, MQTT
  - Amazon SQS, Google Pub/Sub
  - HTTP, gRPC, TCP, Email, WebSocket
  - Change Data Capture (CDC)
  - File, S3, Google Cloud Storage
- Supported data formats
  - JSON, XML, Avro, Protobuf, Text, Binary, Key-value, CSV

Consume and Publish Events with various Data Formats

Default JSON mapping

```
{"event":{"custId":"15","item":"shirt","amount":2}}
```

```
@source(type = mqtt, …, @map(type = json))
define stream OrderStream(custId string, item string, amount int);
```

Custom JSON mapping

```
{"id":"15","itm":"shirt","count":2}
```

```
@source(type = mqtt, …,
        @map(type = json, @attribute("$.id","$.itm" "$.count")))
define stream OrderStream(custId string, item string, amount int);
```

Data Filtering and Preprocessing

- Filtering
  - Value ranges
  - String matching
  - Regex
- Setting Defaults
  - Null checks
  - Default function
  - If-then-else function

```
define stream OrderStream
              (custId string, item string, amount int);

from OrderStream[item!=“unknown”]
select default(custId, “internal”) as custId,
       item,
       ifThenElse(amount<0, 0, amount) as amount,
insert into CleansedOrderStream;
```

Data Transformation

- Data extraction
  - JSON, Text
- Reconstruct messages
  - JSON, Text
- Inline operations
  - Math, Logical operations
- Inbuilt functions
  - 60+ extensions
- Custom functions
  - Java, JS

```
json:getDouble(json,"$.amount") as amount
```

```
str:concat('Hello ',name) as greeting
```

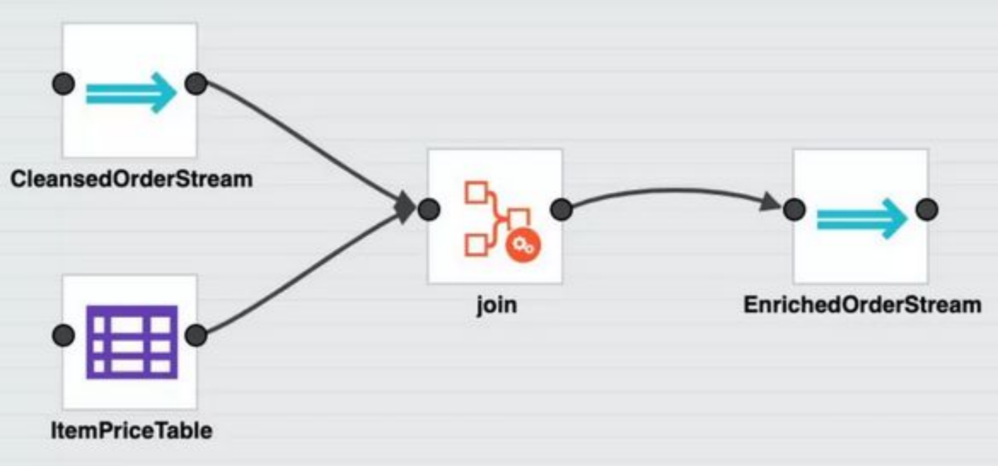
```
amount * price as cost
```

```
time:extract('DAY', datetime) as day
```

```
myFunction(item, price) as discount
```

Database Integration and Caching

- Supported Databases and frameworks
  - RDBMS (MySQL, Oracle, DB2, Postgre, H2)
  - Redis
  - Hazelcast
  - MongoDB
  - HBase
  - Cassandra
  - Solr
  - Elastic Search



In-memory Table

Joining stream with a table

```
define stream CleansedOrderStream
           (custId string, item string, amount int);


@primaryKey('name')
@index('unitPrice')
define table ItemPriceTable (name string, unitPrice double);

from CleansedOrderStream as O join ItemPriceTable as T
    on O.item == T.name
select O.custId, O.item, O.amount * T.unitPrice as price
insert into EnrichedOrderStream;
```

In-memory Table

Join Query

Database Integration

Joining stream with a table

```
define stream CleansedOrderStream
          (custId string, item string, amount int);

@store(type='rdbms', …,)
@primaryKey('name')
@index('unitPrice')
define table ItemPriceTable(name string, unitPrice double);

from CleansedOrderStream as O join ItemPriceTable as T
    on O.item == T.name
select O.custId, O.item, O.amount * T.unitPrice as price
insert into EnrichedOrderStream;
```

Table backed with DB

Join Query

Database Caching

Joining table with Cache (Preloads data for high read performance)

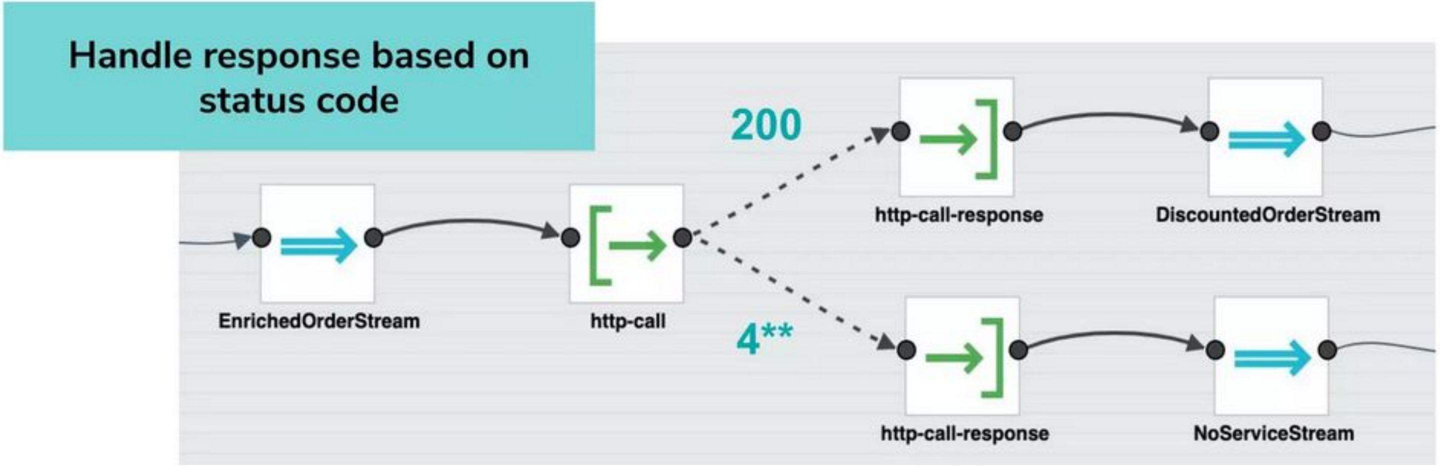
```
define stream CleansedOrderStream
          (custId string, item string, amount int);

@store(type='rdbms', …, @cache(cache.policy='LRU', … ))
@primaryKey('name')
@index('unitPrice')
define table ItemPriceTable(name string, unitPrice double);

from CleansedOrderStream as O join ItemPriceTable as T
    on O.item == T.name
select O.custId, O.item, O.amount * T.unitPrice as price
insert into EnrichedOrderStream;
```

Table with Cache

Join Query

Service Integration and Error Handling

- Enriching data with HTTP and gRPC service calls
  - Non blocking
  - Handle responses based on status codes



SQL for HTTP Service Integration

Call external HTTP service and consuming the response

Call service

```
@sink(type='http-call', publisher.url="http://mystore.com/discount",
    sink.id="discount", @map(type='json'))
define stream EnrichedOrderStream (custId string, item string, price double);
```

Consume Response

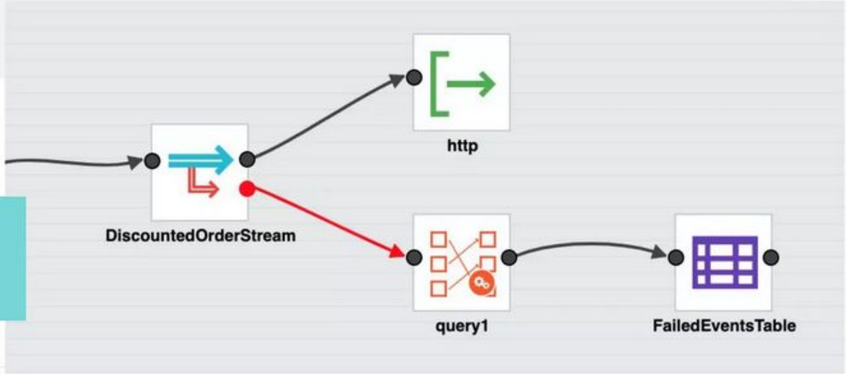
```
@source(type='http-call-response', http.status.code="200",
    sink.id="discount", @map(type='json',
      @attributes(custId ="trp:custId", ..., price="$.discountedPrice")))
define stream DiscountedOrderStream (custId string, item string, price double);
```

Error Handling Options

- Options when endpoint is not available
  - Log and drop the events
  - Wait and back pressure until the service becomes available
  - Divert events to another stream for error handling
- In all cases system continuously retries for reconnection

Events Diverted into Error Stream

```
@onError(action='stream')
@sink(type='http', publisher.url = 'http://localhost:8080/logger',
   on.error='stream', @map(type = 'json'))
define stream DiscountedOrderStream (custId string, item string, price double);

from !DiscountedOrderStream
select custId, item, price, _error
insert into FailedEventsTable;
```



Diverting connection failure events into table.

Data Summarization

- Type of data summarization
- Time based
  - Sliding time window
  - Tumbling time window
  - On time granularities (secs to years)
- Event count based
  - Sliding length window
  - Tumbling length window
- Session based
- Frequency based

- Type of aggregation
  - Sum
  - Count
  - Avg
  - Min
  - Max
  - DistinctCount
  - StdDev

Summarizing Data Over SHorter Period of Time

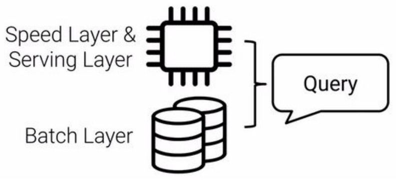
- Use window query to aggregate orders over time for each customer

```
define stream DiscountedOrderStream (custId string, item string, price double);

from DiscountedOrderStream#window.time(10 min)
select custId, sum(price) as totalPrice
group by custId
insert into AlertStream;
```

Window query
with aggregation and
rate limiting

Aggregation over multiple Time Granularities

- Aggregation on every second, minute, hour, ..., year
- Built using λ architecture
  - In-memory real time data
  - RDBMS based historical data




```
define aggregation OrderAggregation
   from OrderStream
   select custId, itemId, sum(price) as total, avg(price) as avgPrice
   group by custId, itemId
   aggregate every sec ... year;
```

Data Retrieval from Aggregations

- Query to retrieve data for relevant time interval and granularity

```
from OrderAggregation
   within "2019-10-06 00:00:00",
          "2019-10-30 00:00:00"
   per "days"
select total as orders;
```

Data being retrieved both from memory and DB with milliseconds accuracy

Rule Processing

- Types of predefined rules
- Rules on **single event**
  - Filter, If-then-else, Match, etc.
- Rules on **collection of events**
  - Summarization
  - Join with window or table
- Rules based on **event occurrence order**
  - Pattern detection
  - Trend (sequence) detection
  - Non-occurrence of event

Alert based on event occurrence order

- Use pattern query to detect event occurrence order and non-occurrence

```
define stream OrderStream (custId string, orderId string, ...);
define stream PaymentStream (orderId string, ...);

from every (e1=OrderStream) ->
           not PaymentStream[e1.orderId==orderId] for 15 min
select e1.custId, e1.orderId, ...
insert into PaymentDelayedStream;
```

Non occurrence of event

Serving Online and Predefined ML Models

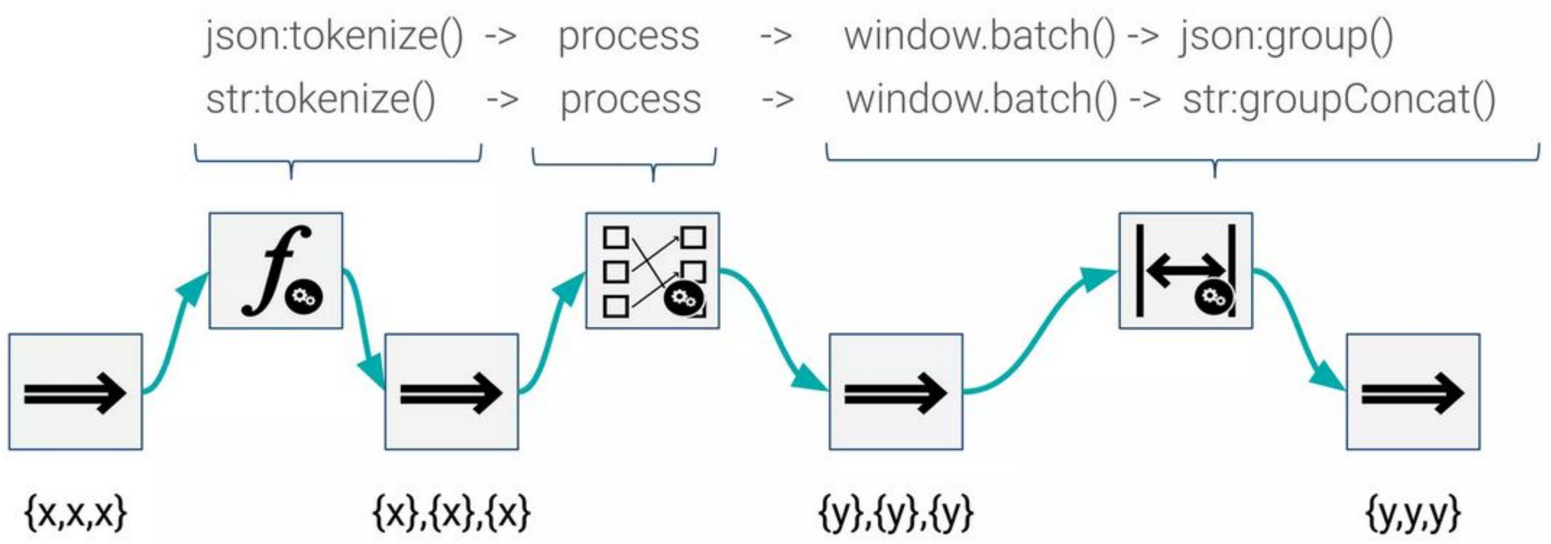
- Type of machine learning and artificial intelligence processing
  - Anomaly detection
    - Markov model
  - Serving pre-created ML models
    - PMML (build from Python, R, Spark, H2O.ai, etc.)
    - Tensorflow
  - Online machine learning
    - Clustering
    - Classification
    - Regression

Find recommendations

```
from OrderStream
     #pmml:predict("/home/user/ml.model",custId, itemId)
insert into RecommendationStream;
```

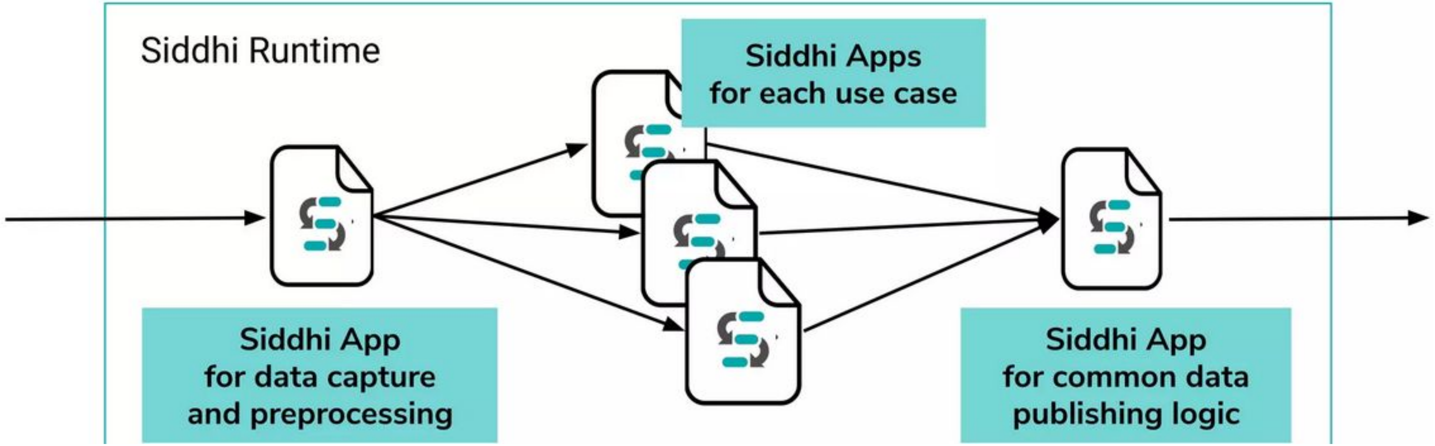
Scatter-gather and Data Pipelining

Divide into sub-elements, process each and combine the results



Modularization

- Create Siddhi App per use case (Collection of queries)
- Connect multiple Siddhi Apps using in-memory source and sink
- Allow rules addition and deletion at runtime



Periodically Trigger Events

- Periodic events can be generated to initialize data pipelines
  - Time interval

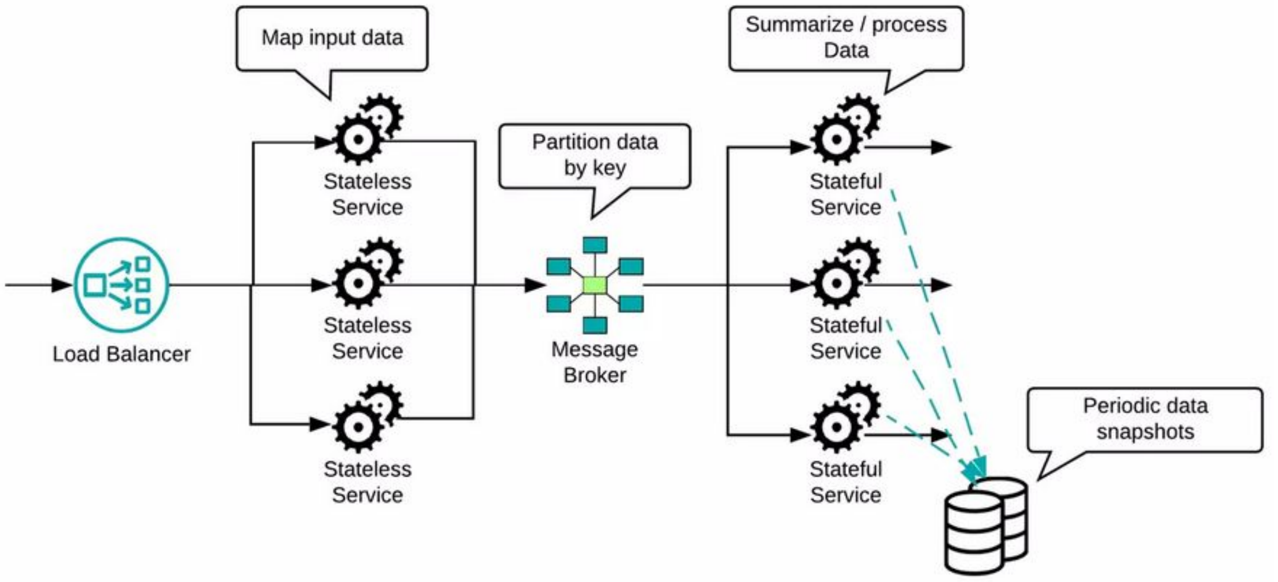
    ```
    define trigger FiveMinTrigger at every 5 min;
    ```
  - Cron expression

    ```
    define trigger WorkStartTrigger at '0 15 10 ? * MON-FRI';
    ```
  - At start

    ```
    define trigger InitTrigger at 'start';
    ```

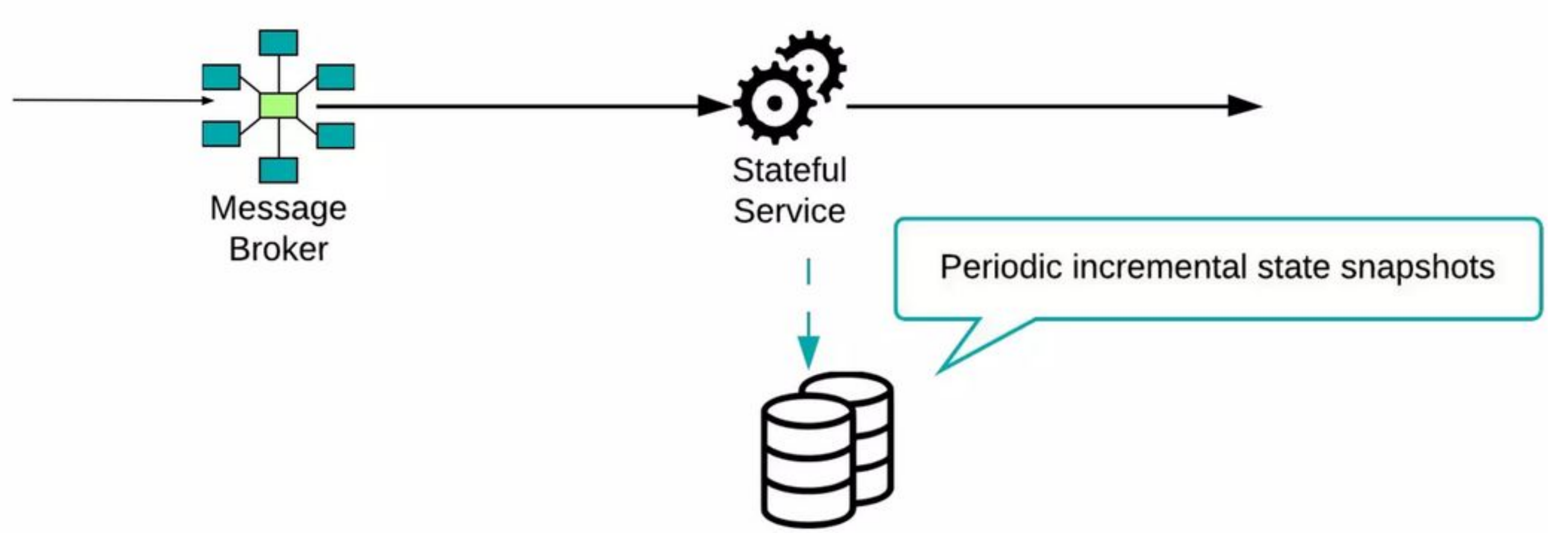
Scalable Stateful Apps

- Data kept in memory
- Perform periodic state snapshots and replay data from NATS
- Scalability is achieved partitioning data by key



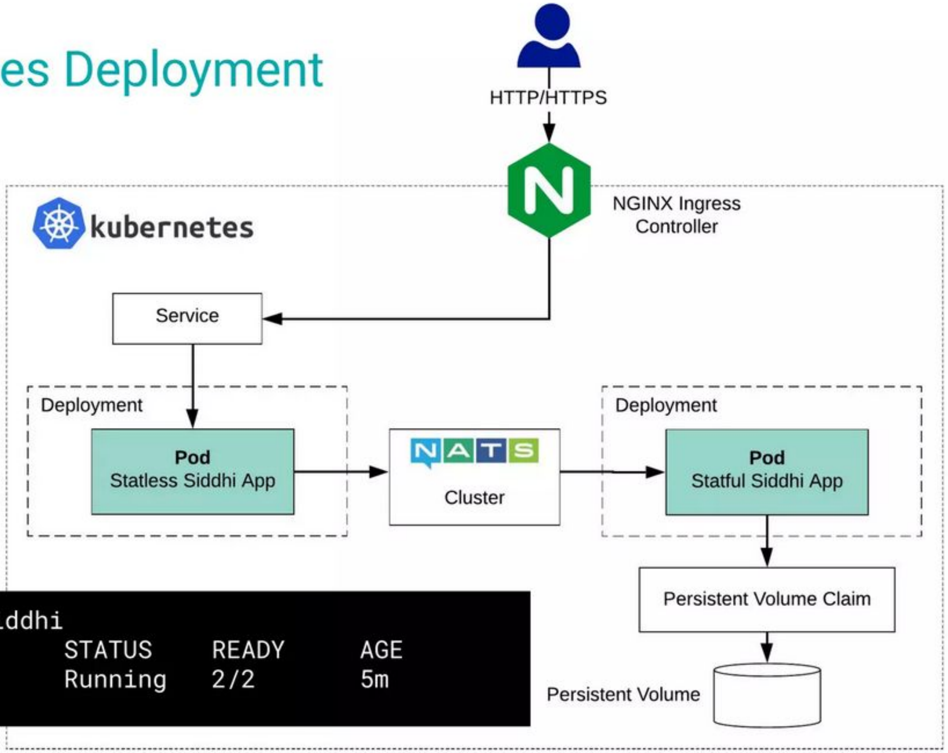
Incremental checkpointing

- System snapshots periodically, replay data from sources upon failure



Siddhi on Kubernetes

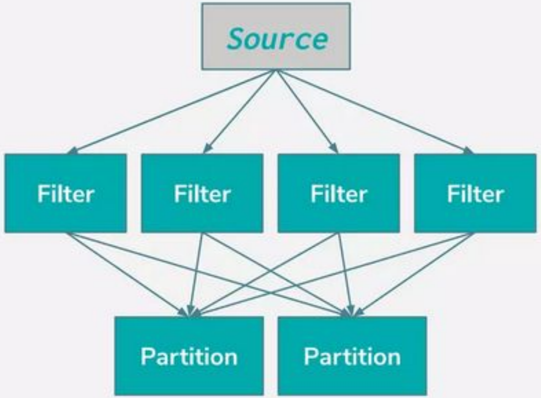
Kubernetes Deployment



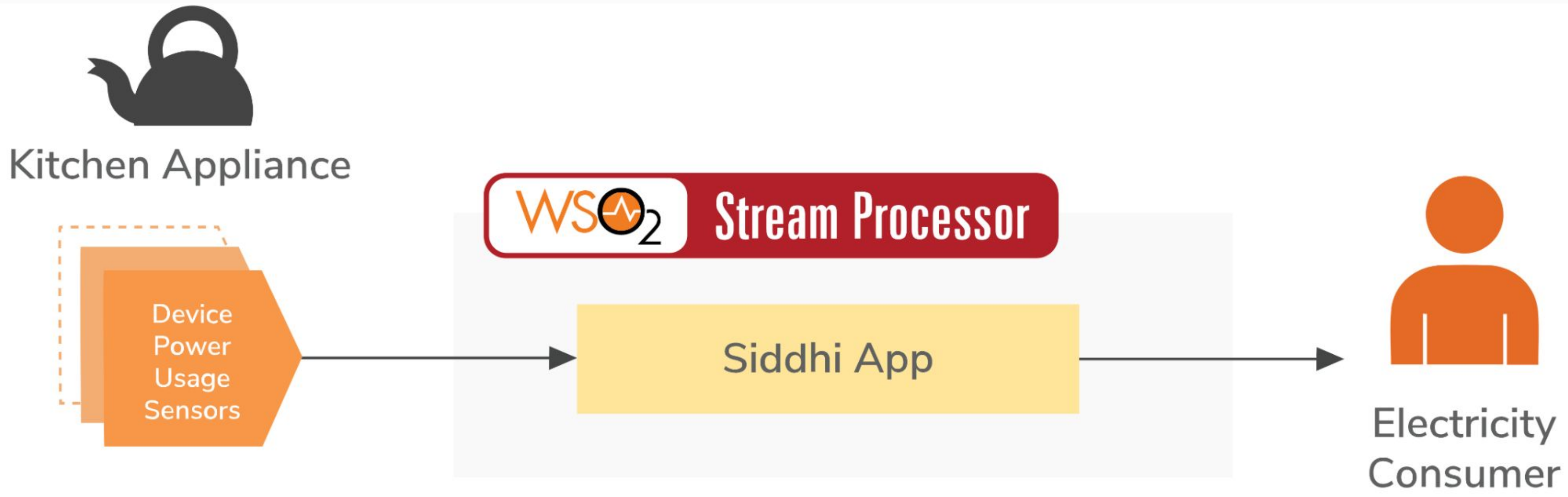

```
$ kubectl get siddhi
NAME                STATUS     READY     AGE
Sample-app          Running    2/2       5m
```

A sample distributed Siddhi application

```
@source(type = 'HTTP', …, @map(type = 'json'))
define stream ProductionStream (name string, amount double, factoryId int);

@dist(parallel = '4', execGroup = 'gp1')
from ProductionStream[amount > 100]
select *
insert into HighProductionStream ;

@dist(parallel = '2', execGroup = 'gp2')
partition with (factoryId of HighProductionStream)
begin
     from HighProductionStream#window.timeBatch(1 min)
     select factoryId, sum(amount) as amount
     group by factoryId
     insert into ProdRateStream ;
end;
```



Siddhi Distributed Stream Processing - Appliance Monitoring



Siddhi Distributed Stream Processing - Appliance Monitoring

EnergyAlert.siddhi

```
@App:name('Energy-Alert-App')
@App:description('Energy consumption and anomaly detection')

-- Streams
@source(type = 'http', receiver.url=' ', topic = 'device-power',
@map(type = 'json'))
define stream DevicePowerStream (type string, deviceID string, power int);

@sink(type = 'email', to = '{{autorityContactEmail}}', username = 'john', address = 'john@gmail.com', password = 'test', subject = 'High power
consumption of {{deviceID}}',
@map(type = 'text',
@payload('Device ID: {{deviceID}} of room : {{roomID}} power is consuming {{finalPower}}kW/h. ')))
define stream AlertStream (deviceID string, roomID string, initialPower double, finalPower double, autorityContactEmail string);
```

Siddhi Distributed Stream Processing - Appliance Monitoring

```
-- Tables
@Store(type="rdbms", jdbc.url="jdbc:mysql://localhost:3306/sp", username="root", password="root" , jdbc.driver.name="com.mysql.jdbc.Driver",field.length="symbol:100")
define table DeviceIdInfoTable (deviceID string, roomID string, autorityContactEmail string);

-- Queries
@info(name = 'monitored-filter')
from DevicePowerStream[type == 'monitored']
select deviceID, power
insert current events into MonitoredDevicesPowerStream;
```

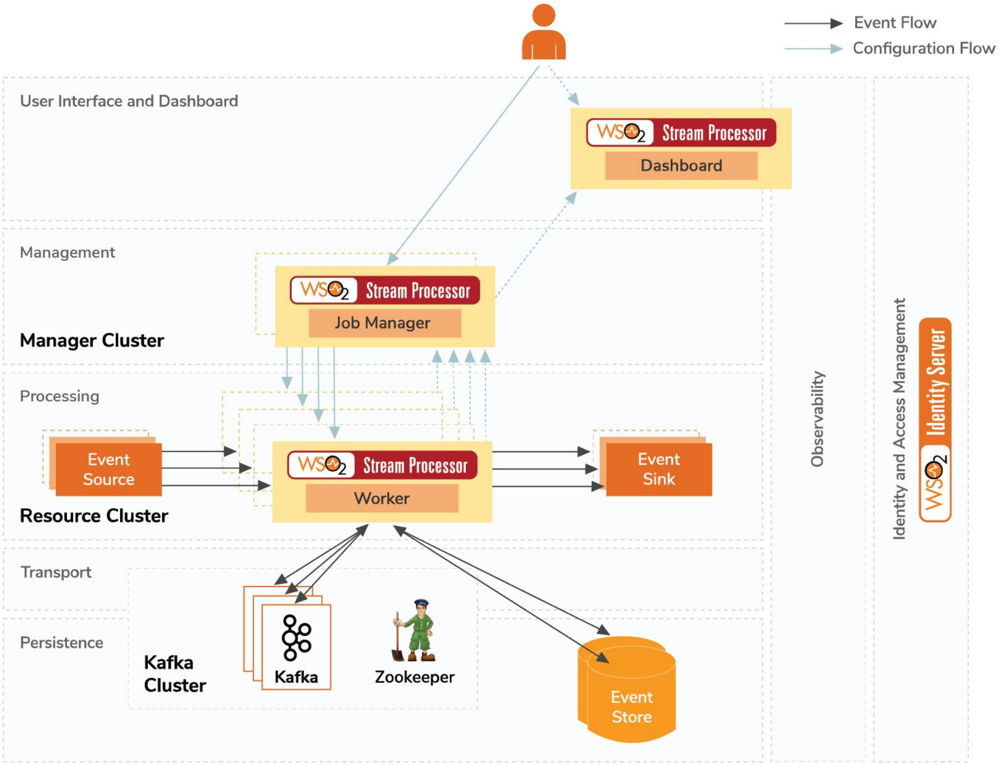
Siddhi Distributed Stream Processing - Appliance Monitoring

```

@info(name = 'power-increase-pattern')
partition with (deviceID of MonitoredDevicesPowerStream)
begin
@info(name = 'avg-calculator')
from MonitoredDevicesPowerStream#window.time(2 min)
select deviceID, avg(power) as avgPower
insert current events into #AvgPowerStream;

@info(name = 'power-increase-detector')
from every e1 = #AvgPowerStream -> e2 = #AvgPowerStream[(e1.avgPower + 5) <= avgPower] within 10 min
select e1.deviceID as deviceID, e1.avgPower as initialPower, e2.avgPower as finalPower
insert current events into RisingPowerStream;
end;
```

Siddhi Distributed Stream Processing - Appliance Monitoring

```
@info(name = 'power-range-filter')
from RisingPowerStream[finalPower > 100]
select deviceID, initialPower, finalPower
insert current events into DevicesWithinRangeStream;

@info(name = 'enrich-alert')
from DevicesWithinRangeStream as s join DeviceIdInfoTable as t
on s.deviceID == t.deviceID
select s.deviceID as deviceID, t.roomID as roomID, s.initialPower as initialPower, s.finalPower as finalPower, t.autorityContactEmail as autorityContactEmail
insert current events into AlertStream;
```

Siddhi Distributed Stream Processing - Appliance Monitoring



Siddhi Distributed Stream Processing - Appliance Monitoring

EnergyAlertDistributed.siddhi

```
@App:name('Energy-Alert-App')
@App:description('Energy consumption and anomaly detection')

-- Streams
@source(type = 'http', receiver.url=' ', topic = 'device-power',
@dist(parallel ='3')
@map(type = 'json'))
define stream DevicePowerStream (type string, deviceID string, power int);

@sink(type = 'email', to = '{{autorityContactEmail}}', username = 'john', address = 'john@gmail.com', password = 'test', subject
= 'High power consumption of {{deviceID}}',
@map(type = 'text',
@payload('Device ID: {{deviceID}} of room : {{roomID}} power is consuming {{finalPower}}kW/h. ')))
define stream AlertStream (deviceID string, roomID string, initialPower double, finalPower double, autorityContactEmail string);
```

Siddhi Distributed Stream Processing - Appliance Monitoring

```

-- Tables
@Store(type="rdbms", jdbc.url="jdbc:mysql://localhost:3306/sp", username="root", password="root" , jdbc.driver.name="com.mysql.jdbc.Driver",field.length="symbol:100")
define table DeviceIdInfoTable (deviceID string, roomID string, autorityContactEmail string);

-- Queries
@info(name = 'monitored-filter')
@dist(execGroup='group1', parallel ='3')
from DevicePowerStream[type == 'monitored']
select deviceID, power
insert current events into MonitoredDevicesPowerStream;

```

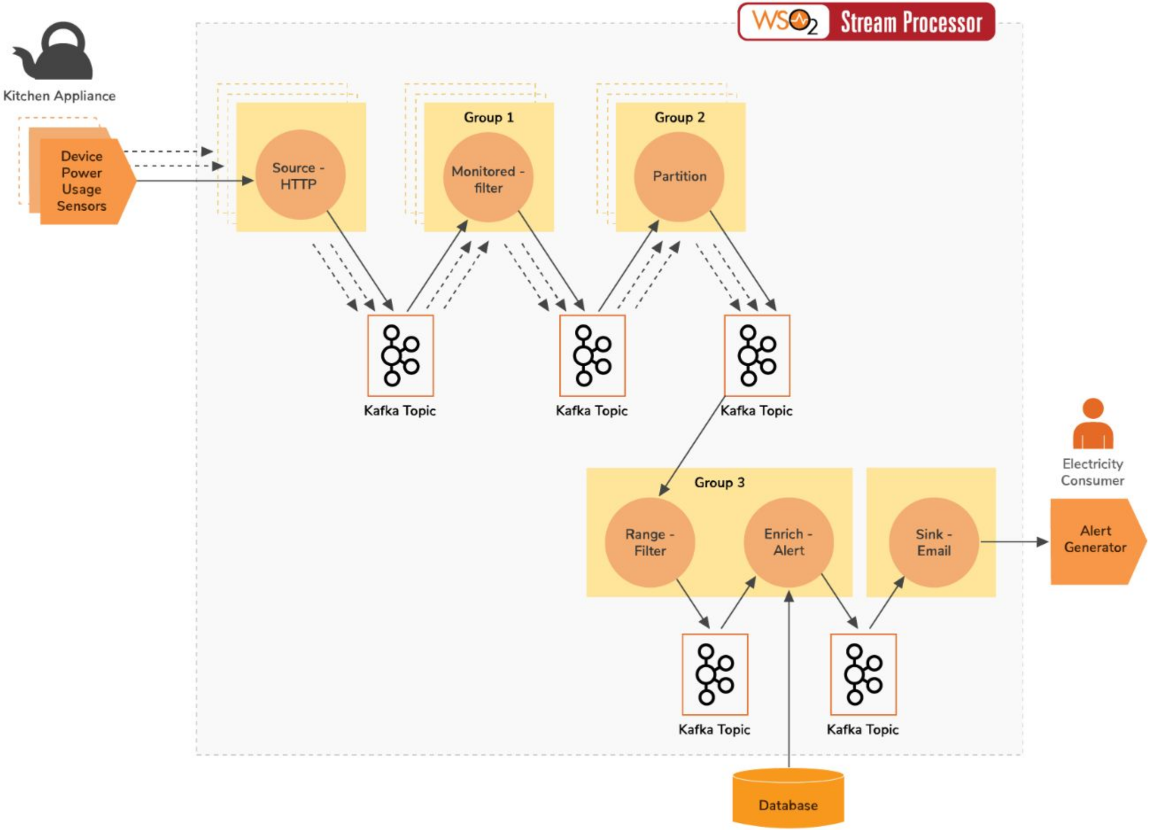
Siddhi Distributed Stream Processing - Appliance Monitoring

```
@info(name = 'power-increase-pattern')
@dist(execGroup='group2', parallel ='3')
partition with (deviceID of MonitoredDevicesPowerStream)
begin
@info(name = 'avg-calculator')
from MonitoredDevicesPowerStream#window.time(2 min)
select deviceID, avg(power) as avgPower
insert current events into #AvgPowerStream;

@info(name = 'power-increase-detector')
from every e1 = #AvgPowerStream -> e2 = #AvgPowerStream[(e1.avgPower + 5) <= avgPower] within 10 min
select e1.deviceID as deviceID, e1.avgPower as initialPower, e2.avgPower as finalPower
insert current events into RisingPowerStream;
end;
```

Siddhi Distributed Stream Processing - Appliance Monitoring

```

@info(name = 'power-range-filter')
@dist(execGroup='group3' ,parallel ='1')
from RisingPowerStream[finalPower > 100]
select deviceID, initialPower, finalPower
insert current events into DevicesWithinRangeStream;

@info(name = 'enrich-alert')
@dist(execGroup='group3' ,parallel ='1')
from DevicesWithinRangeStream as s join DeviceIdInfoTable as t
on s.deviceID == t.deviceID
select s.deviceID as deviceID, t.roomID as roomID, s.initialPower as initialPower, s.finalPower as finalPower, t.autorityContactEmail as autorityContactEmail
insert current events into AlertStream;
```

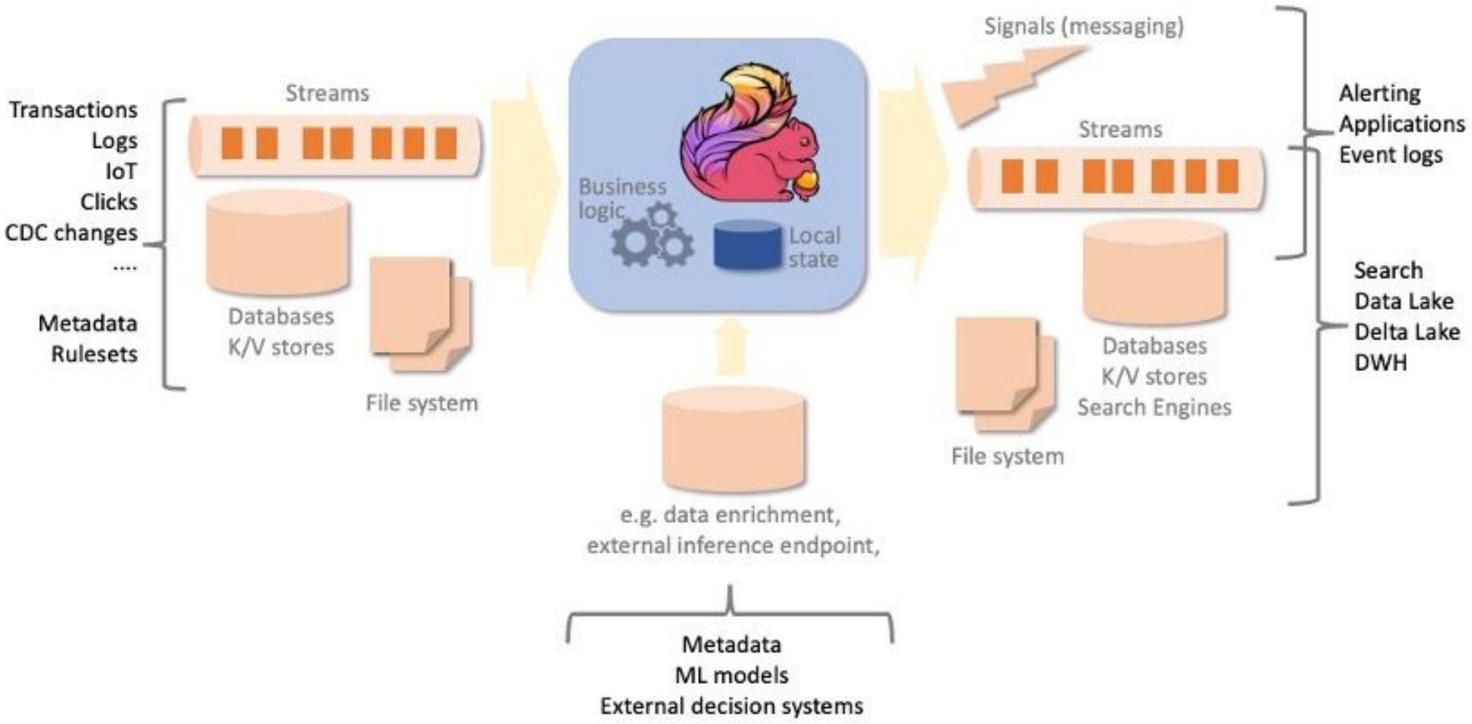
Siddhi Distributed Stream Processing - Appliance Monitoring



Apache Flink

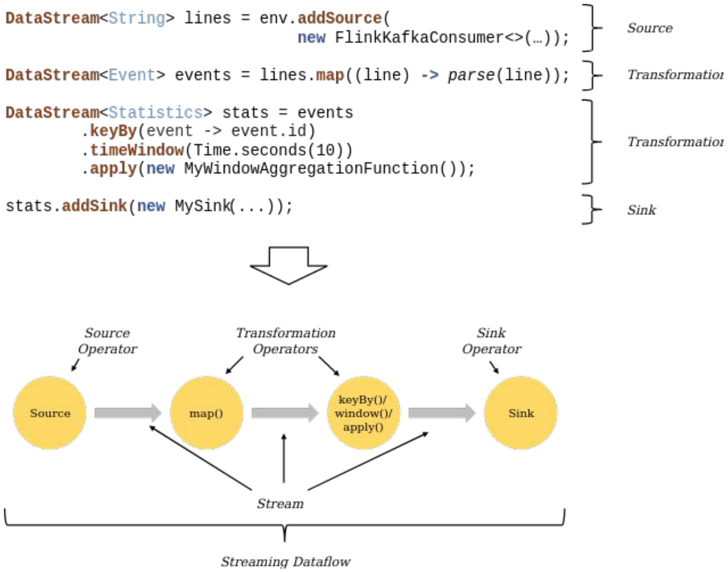
- Apache Flink is an open-source, distributed engine for stateful processing over unbounded (streams) and bounded (batches) data sets.
- Stream processing applications are designed to run continuously, with minimal downtime, and process data as it is ingested.
- Apache Flink is designed for low latency processing, performing computations in-memory, for high availability, removing single point of failures, and to scale horizontally.
- Apache Flink's features include advanced state management with exactly-once consistency guarantees, event-time processing semantics with sophisticated out-of-order and late data handling.
- Apache Flink has been developed for streaming-first, and offers a unified programming interface for both stream and batch processing.

Apache Flink (Contd.)



Structure of a Flink Application

In Flink, applications are composed of streaming data flows that may be transformed by user-defined operators. These dataflows form directed graphs that start with one or more sources, and end in one or more sinks.



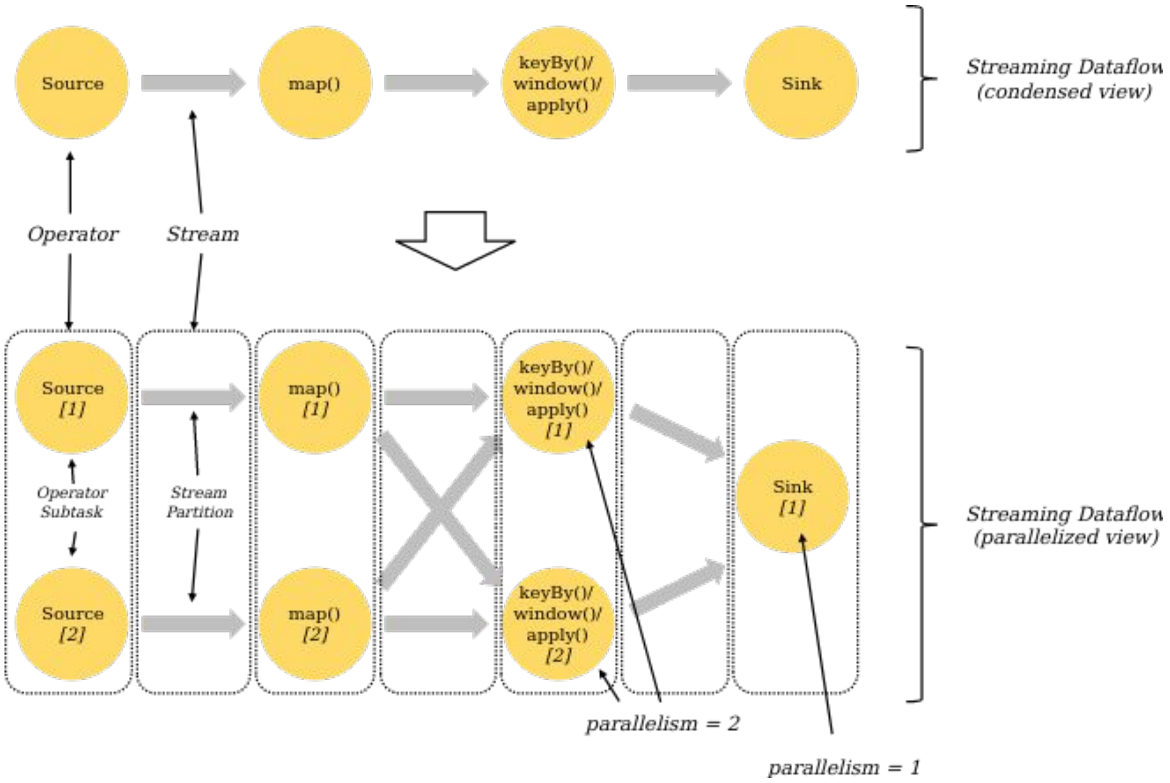
https://nightlies.apache.org/flink/flink-docs-stable/docs/learn-flink/overview/

Structure of a Flink Application (Contd.)

Programs in Flink are inherently parallel and distributed. During execution, a stream has one or more stream partitions, and each operator has one or more operator subtasks. The operator subtasks are independent of one another, and execute in different threads and possibly on different machines or containers.

The number of operator subtasks is the parallelism of that particular operator. Different operators of the same program may have different levels of parallelism.

Structure of a Flink Application (Contd.)



Setup and run Apache Flink on your system

https://nightlies.apache.org/flink/flink-docs-master/docs/try-flink/local_installation/

Practical Session

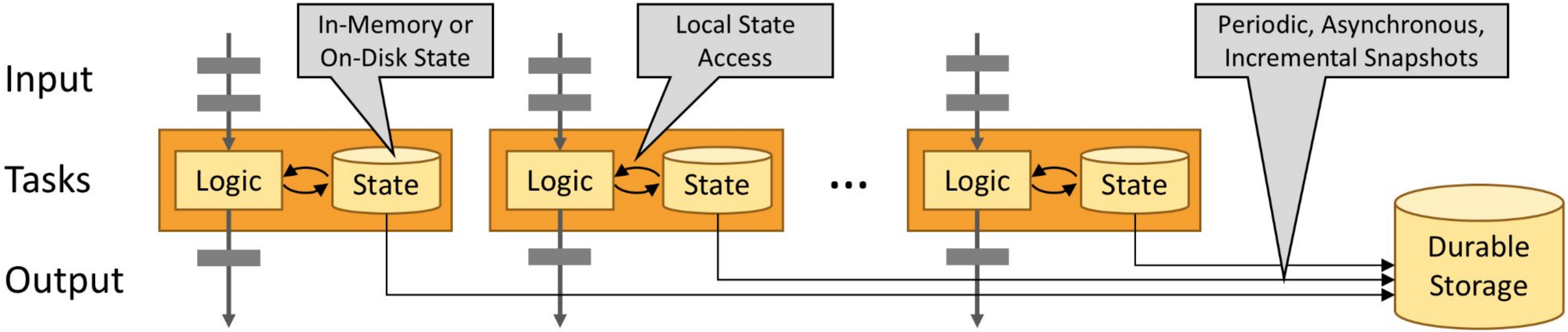
Why would we use Apache Flink?

- Apache Flink is used to build many different types of streaming and batch applications, due to the broad set of features.
- Some of the common types of applications powered by Apache Flink are:
  - **Event-driven applications**, ingesting events from one or more event streams and executing computations, state updates or external actions. Stateful processing allows implementing logic beyond the Single Message Transformation, where the results depend on the history of ingested events.
  - **Data Analytics applications**, extracting information and insights from data. Traditionally executed by querying finite data sets, and re-running the queries or amending the results to incorporate new data. With Apache Flink, the analysis can be executed by continuously updating, streaming queries or processing ingested events in real-time, continuously emitting and updating the results.
  - **Data pipelines applications**, transforming and enriching data to be moved from one data storage to another. Traditionally, extract-transform-load (ETL) is executed periodically, in batches. With Apache Flink, the process can operate continuously, moving the data with low latency to their destination.

How does Apache Flink Work?

- Flink is a high throughput, low latency stream processing engine.
- A Flink application consists of an arbitrary complex acyclic dataflow graph, composed of streams and transformations.
- Data is ingested from one or more data sources and sent to one or more destinations.
- Source and destination systems can be streams, message queues, or datastores, and include files, popular database and search engines.
- Transformations can be stateful, like aggregations over time windows or complex pattern detection.
- State is always accessed locally, which helps Flink applications achieve high throughput and low-latency. You can choose to keep state on the JVM heap, or if it is too large, in efficiently organized on-disk data structures.

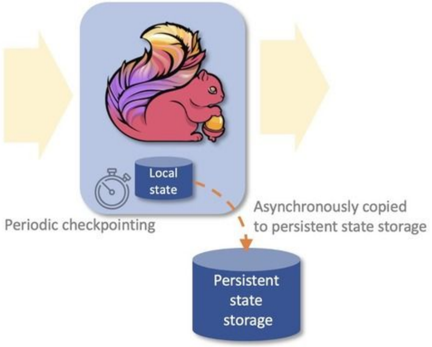
How does Apache Flink Work?



https://nightlies.apache.org/flink/flink-docs-stable/docs/learn-flink/overview/

How does Apache Flink Work? (Contd.)

- Fault tolerance is achieved by two separate mechanisms: automatic and periodic checkpointing of the application state, copied to a persistent storage, to allow automatic recovery in case of failure; on-demand savepoints, saving a consistent image of the execution state, to allow stop-and-resume, update or fork the Flink job, retaining the application state across stops and restarts.
- Checkpoint and savepoint mechanisms are asynchronous, taking a consistent snapshot of the state without “stopping the world”, while the application keeps processing events.



Benefits of Apache Flink

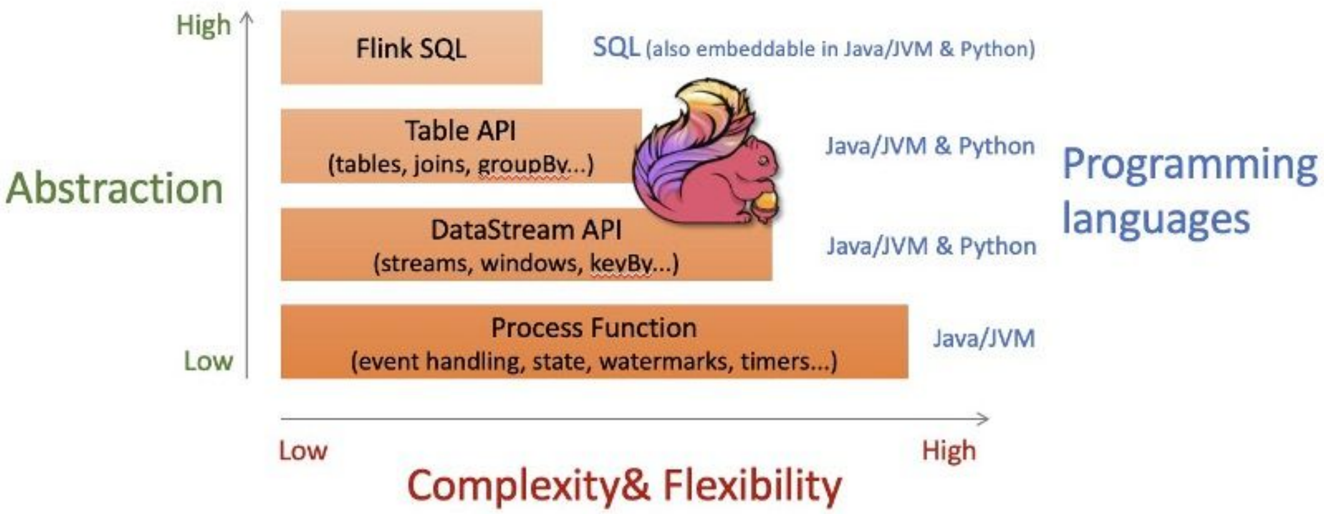
- Process both unbounded (streams) and bounded (batches) data sets
  - Apache Flink can process both unbounded and bounded data sets, i.e., streams and batch data. Unbounded streams have a start but are virtually infinite and never end. Processing can theoretically never stop.
- Run applications at scale
  - Apache Flink is designed to run stateful applications at virtually any scale. Processing is parallelized to thousands of tasks, distributed multiple machines, concurrently.
- In-memory performance
  - Data flowing through the application and state are partitioned across multiple machines. Hence, computation can be completed by accessing local data, often in-memory.

Benefits of Apache Flink (Contd.)

- Exactly-once state consistency
  - Applications beyond single message transformations are stateful. The business logic needs to remember events or intermediate results. Apache Flink guarantees consistency of the internal state, even in case of failure and across application stop and restart. The effect of each message on the internal state is always applied exactly-once, regardless the application may receive duplicates from the data source on recovery or on restart.
- Wide range of connectors
  - Apache Flink has a number of proven connectors to popular messaging and streaming systems, data stores, search engines, and file system. Some examples are Apache Kafka, Amazon Kinesis Data Streams, Amazon SQS, Active MQ, Rabbit MQ, NiFi, OpenSearch and ElasticSearch, DynamoDB, HBase, and any database providing JDBC client.

Benefits of Apache Flink (Contd.)

Multiple levels of abstractions

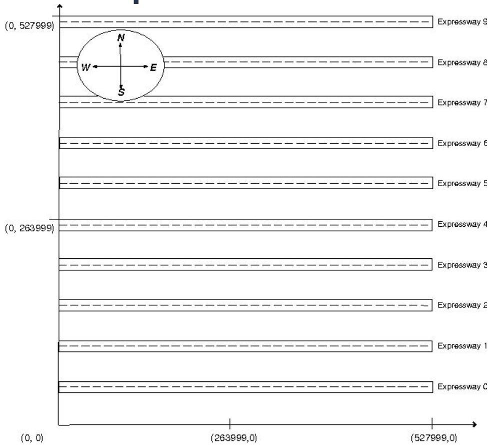


Stream Processor Performance Tuning and Benchmarking

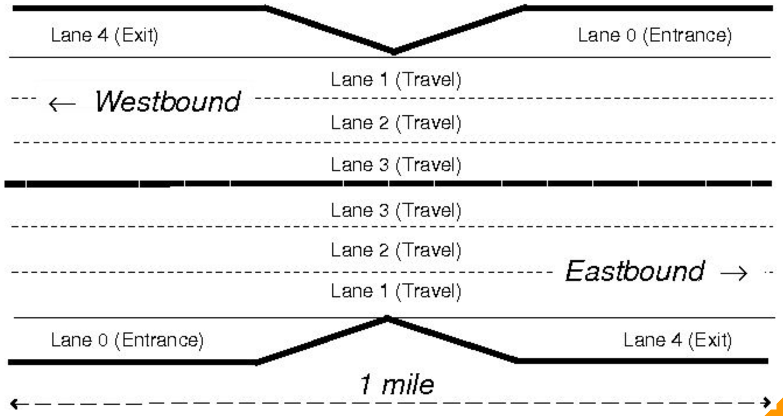
- EP system performance is a critical quality of service aspect worth of investigation.
- EP benchmarking itself is a vast area with significant amount of literature.
- However, EP benchmarking still has number of issues to be addressed.
- There are few widely agreed upon benchmarks for EP.
- Most of the workload characterization and performance studies have been conducted on microbenchmarks.

Linear Road Benchmark

- One of the earliest and established benchmarks is Linear Road (LR)
- It simulates a highway toll system and it has been implemented on multiple different CEP systems.
- Although LR has been introduced circa 2004 it has been widely implemented in multiple CEP systems



The Geometry of Linear City



An Example Expressway Segment

NEXMark Benchmark

- NEXMark (Niagara Extension to XMark) is a benchmark which is being currently developed in order to benchmark queries over continuous data streams.
- These are multiple queries over a three entities model representing on online auction system:
  - **Person** represents a person submitting an item for auction and/or making a bid on an auction.
  - **Auction** represents an item under auction.
  - **Bid** represents a bid for an item under auction.

https://datalab.cs.pdx.edu//niagaraST/NEXMark/

Thank you!