

Real Time Analytics with KAFKA

Beginner's guide to Stream processing

Satish Sharma | 28-June-2019

Table of Contents

Introduction.....	3
Kafka offers all three capabilities of streaming platform	3
Publish and subscribe to streams of records	3
Store streams of records in a fault-tolerant durable way.	3
Process streams of records as they occur.	4
First few concepts	4
Where does Kafka fits in	4
Kafka in production	5
Kafka Use Cases.....	5
Kafka is used in two broad class of application.....	5
Some of the major use cases of Kafka.....	5
Kafka API's.....	6
Producer API.....	6
Consumer API.....	6
Streams API.....	6
Connector API	6
How to Setup single node Kafka cluster	7
Setup Zookeeper	7
Setup Kafka.....	7
Test your created cluster	9
Create a topic in kafka	9
Send records to newly created topic.....	9
Consume records using the console consumer	9
Kafka Steams Concepts.....	10
KAFKA Refresher	10
Stream	11
Processor Topology.....	11
Stream application.....	12
Stream Processor.....	12
Source Processor.....	12
Sink Processor.....	12
KStream.....	13
KTable	13

State Stores	13
Time.....	13
Event Time	14
Ingestion Time	14
Processing Time.....	14
Transformations on Kafka Streams	14
Stateful Transformations	14
Stateless Transformations	15
Interactive Queries.....	15
Use Case: Fleet Management Dashboard	17
Scenario.....	17
Functional Requirements.....	17
Solution.....	17
Assumptions	17
Architecture Diagram.....	18
Environment.....	18
Kafka (Topics & StateStores)	18
Applications (spring boot).....	18
Show me some Code.....	19
GPS event	19
Stream processor	19
Query the store	20
Response for Offline & Online vehicle count	20

Introduction

Apache Kafka is an open-source distributed stream processing platform originally developed by LinkedIn and was donated to Apache in 2011.

We can describe KAFKA as collection of files, filled with messages that are distributed across multiple machines. Most of Kafka analogies revolving around tying these various individual logs together, routing messages from producers to consumers reliably, replicating for fault tolerance, and handling failure gracefully. Its architecture inherits more from storage systems like HDFS, HBase, or Cassandra than it does from traditional messaging systems that implement JMS (Java Message Service) or AMQP (Advanced Message Queuing Protocol). The underlying abstraction is a partitioned log—essentially a set of append-only files spread over several machines—which encourages sequential access patterns. A Kafka cluster is a distributed system, spreading data over many machines both for fault tolerance and for linear scale-out. Kafka has quickly evolved from a messaging system to full-fledged streaming platform having below capabilities

- Scalable
- Fault tolerant
- A great publish-subscribe messaging system
- Capable of higher throughput compared with most messaging systems
- Highly durable
- Highly reliable
- High performant

Kafka offers all three capabilities of streaming platform

Publish and subscribe to streams of records

we already have a lot of messaging systems then why one more. The answer to this could be

At heart of KAFKA lies the humble, immutable commit log, and from there you can subscribe to it, and publish data to any number of systems or real-time applications. Unlike messaging queues, Kafka is a highly scalable, fault tolerant distributed system

Kafka has stronger ordering guarantees than a traditional messaging system, too. A traditional queue retains records in-order on the server, and if multiple consumers consume from the queue then the server hands out records in the order they are stored. However, although the server hands out records in order, the records are delivered asynchronously to consumers, so they may arrive out of order on different consumers.

Kafka does it better. By having a notion of parallelism, the within the topics, Kafka can provide both ordering guarantees and load balancing over a pool of consumer processes

Store streams of records in a fault-tolerant durable way.

In Kafka Data is written to disk in a fault tolerant way using replication of data. Kafka allows producers to wait for acknowledgment for completion, and a write is not considered complete until it is fully replicated and guaranteed to persist even if the server written to fails. Kafka will perform the same whether you have 50 KB or 50 TB of persistent data on the server. As a result, we can think of Kafka as a kind of special purpose distributed filesystem dedicated to high-performance, low-latency commit log storage, replication, and propagation

An abstraction of a distributed commit log commonly found in distributed databases, Apache Kafka provides durable storage. Kafka can act as a 'source of truth', being able to distribute data across multiple nodes for a highly available deployment within a single data center or across multiple availability zones.

Process streams of records as they occur.

A streaming platform would not be complete without the ability to manipulate that data as it arrives. The Streams API within Apache Kafka is a powerful, lightweight library that allows for on-the-fly processing

In Kafka a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input, and produces continual streams of data to output topics.

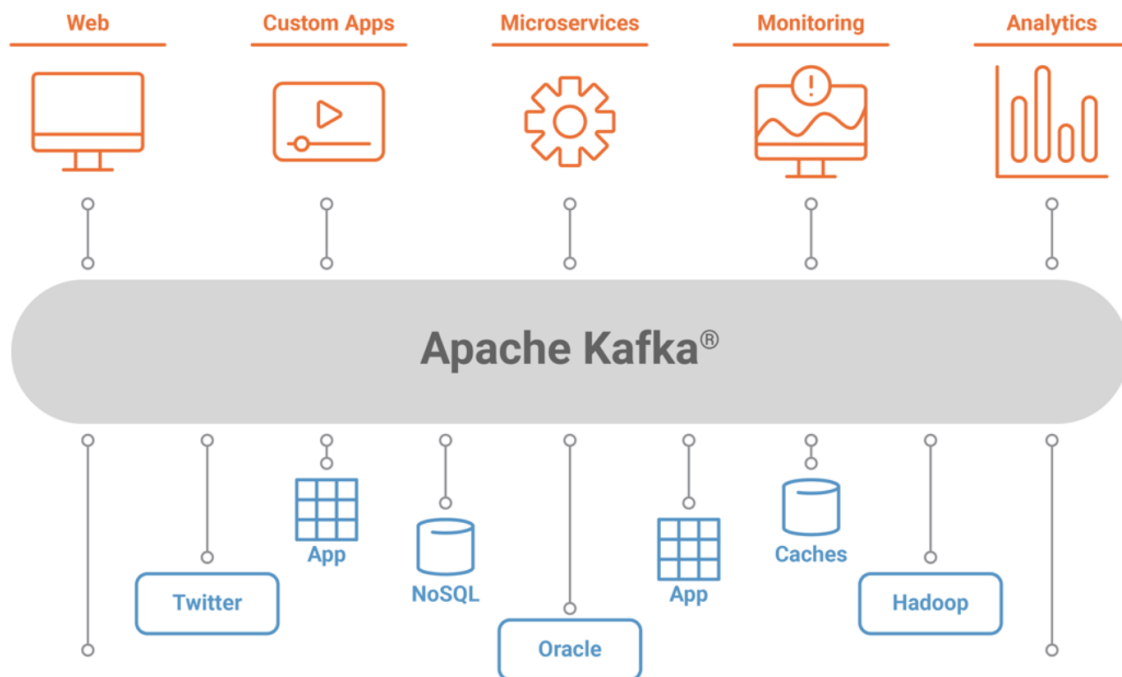
For example, a retail application might take in input streams of sales and shipments and output a stream of reorders and price adjustments computed off this data.

Simple processing can be done directly using the producer and consumer APIs, and for more complex transformations Kafka provides a fully integrated Streams API.

First few concepts

- Kafka is run as a cluster on one or more servers that can span multiple datacenters.
- The Kafka cluster stores streams of *records* in categories called *topics*.
- Each record consists of a key, a value, and a timestamp.

Where does Kafka fits in



Kafka in production

Kafka has a long list of clients running Kafka in production. Be it UBER for using passenger driver matching or providing real time analytics and predictive maintenance for British Gas' smart home and performing numerous real-time processing's across LinkedIn.

Kafka Use Cases

Kafka is used in two broad class of application

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

Some of the major use cases of Kafka

- Messaging
- Real Time Website Activity Tracking
- Metrics
- Log Aggregation
- Stream processing (We shall be having deep dive)
- Event sourcing
- Commit Log

Kafka API's

Kafka has below for core APIs

Producer API

This API allows application to publish stream of records to one or more Kafka topics.

Consumer API

Consumer API allows applications to connect to one or more topics and process the records as they are pushed to those topics.

Streams API

This API allows the application to work as stream processors. Consume records from one or more topics process, transform and produce the stream to one or more topics

Connector API

Connector allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.

How to Setup single node Kafka cluster

Setup Zookeeper

Kafka needs Zookeeper to run. So first we need to setup and start Zookeeper. There are two options for setting up zookeeper

1. For quick and easy setup Kafka is distributed bundled with Zookeeper. You can get a quick-and-dirty single-node ZooKeeper instance. The convenience script is there in "bin" folder of Kafka distribution.

Linux -> `bin/zookeeper-server-start.sh config/zookeeper.properties`

Windows -> `bin\windows\zookeeper-server-start.bat <Path to config file>`

2. Or you can download full-fledged ZooKeeper distributions from(
<https://zookeeper.apache.org/releases.html>)

Un-tar the distribution and Start ZooKeeper server

Linux -> `bin/zkServer.sh <Path to config file>`

Windows -> `bin\zkServer.cmd <Path to config file>`

```
C:\work_dir\RD\kafka\zookeeper-3.4.12\bin>zkServer.cmd
C:\work_dir\RD\kafka\zookeeper-3.4.12\bin>call "C:\Program Files\Java\jdk1.8.0_181\bin\java -Dzookeeper.log.dir=C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\.. -Dzoo
keeper.root.logger=INFO,CONSOLE" -cp "C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\build\classes;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\build\lib\*;C:\work_dir\
RD\kafka\zookeeper-3.4.12\bin\..\lib\*;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\conf\org.apache.zookeeper.server.*" org.apache.zookeeper.server.
quorum.QuorumPeerMain "C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\conf\zoo.cfg"
2019-02-19 10:45:52,047 [myid:] - INFO [main:QuorumPeerConfig@136] - Reading configuration from: C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\conf\zoo.cfg
2019-02-19 10:45:52,079 [myid:] - INFO [main:DatadirCleanupManager@78] - autopurge.snapRetainCount set to 3
2019-02-19 10:45:52,079 [myid:] - INFO [main:DatadirCleanupManager@79] - autopurge.purgeInterval set to 0
2019-02-19 10:45:52,079 [myid:] - INFO [main:DatadirCleanupManager@101] - Purge task is not scheduled.
2019-02-19 10:45:52,079 [myid:] - WARN [main:QuorumPeerMain@116] - Either no config or no quorum defined in config, running in standalone mode
2019-02-19 10:45:52,158 [myid:] - INFO [main:QuorumPeerConfig@136] - Reading configuration from: C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\conf\zoo.cfg
2019-02-19 10:45:52,158 [myid:] - INFO [main:ZooKeeperServerMain@98] - Starting server
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:zookeeper.version=3.4.12-e5259e437540f349646870ea94dc2658c4e44b3b, built on 03/27/20
18 03:55 GMT
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:host.name=LP-5CD7292PTZ.HCLT.CORP.HCL.IN
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.version=1.8.0_181
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.vendor=Oracle Corporation
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.home=C:\Program Files\Java\jdk1.8.0_181\jre
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.class.path=C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\build\classes;C:\work_
dir\RD\kafka\zookeeper-3.4.12\bin\..\build\lib\*;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\lib\*;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\lib\log4j-1.2.17.jar;C:\w
ork_dir\RD\kafka\zookeeper-3.4.12\bin\..\lib\netty-3.10.6.Final.jar;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\lib\slf4j-api-1.7.25.jar;C:\work_dir\RD\kafka\zooke
per-3.4.12\bin\..\lib\slf4j-log4j12-1.7.25.jar;C:\work_dir\RD\kafka\zookeeper-3.4.12\bin\..\conf
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.library.path=C:\Program Files\Java\jdk1.8.0_181\bin;C:\WINDOWS\Sun\Java\bin;C:\
WINDOWS\system32;C:\WINDOWS\CCM;C:\Program Files\MySQL\MySQL Utilities 1.6;C:\Program Files\Git\cmd;C:\Program Files\nodejs\;C:\Users\satish-s\AppData\Local\Microsoft\
WindowsApps;C:\Program Files\Java\jdk1.8.0_181\bin;C:\work_dir\softwares\apache-maven-3.5.4\bin;C:\Users\satish-s\AppData\Roaming\npm;C:\Users\satish-s\AppData\Local\Pr
ograms\Microsoft VS Code\bin;
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.io.tmpdir=C:\Users\satish-s\AppData\Local\Temp\
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:java.compiler=<NA>
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:os.name=Windows 10
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:os.arch=amd64
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:os.version=10.0
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:user.name=satish-s
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:user.home=C:\Users\satish-s
2019-02-19 10:45:52,190 [myid:] - INFO [main:Environment@100] - Server environment:user.dir=C:\work_dir\RD\kafka\zookeeper-3.4.12\bin
2019-02-19 10:45:52,253 [myid:] - INFO [main:ZooKeeperServer@835] - tickTime set to 2000
2019-02-19 10:45:52,253 [myid:] - INFO [main:ZooKeeperServer@844] - minSessionTimeout set to -1
2019-02-19 10:45:52,253 [myid:] - INFO [main:ZooKeeperServer@853] - maxSessionTimeout set to -1
2019-02-19 10:45:52,443 [myid:] - INFO [main:ServerCnxnFactory@117] - Using org.apache.zookeeper.server.NIOServerCnxnFactory as server connection factory
2019-02-19 10:45:52,443 [myid:] - INFO [main:NIOServerCnxnFactory@89] - binding to port 0.0.0.0/0.0.0.0:2181
```

Setup Kafka

- Download Kafka from
(https://www.apache.org/dyn/closer.cgi?path=/kafka/2.1.0/kafka_2.11-2.1.0.tgz)
- un-tar the distribution (on windows use utilities like 7zip)

`tar -xzf kafka_2.11-2.1.0.tgz`

- Edit the file "**server.properties**" present in "**config**" directory. You can create the copy of file as well. You need to update the property "**log.dirs**" to a location where the logs shall be persisted on file system

"log.dirs" = <Folder location to be used for persisting logs>

for ex.

Linux -> log.dirs=/tmp/kafka-logs

Windows -> log.dirs= C:\tmp

```
##### Log Basics #####  
# A comma separated list of directories under which to store log files  
log.dirs=/tmp/kafka-logs
```

- Start the server

Linux -> bin/kafka-server-start.sh config/server.properties

Windows -> kafka-server-start.bat ../../config/server.properties

```
C:\work_dir\RND\kafka\kafka_2.12-2.1.0\bin\windows>kafka-server-start.bat ../../config/server.properties  
[2019-02-20 10:05:22,395] INFO Registered kafka:type=kafka.Log4jController MBean (kafka.utils.Log4jControllerRegistration$)  
[2019-02-20 10:05:23,614] INFO starting (kafka.server.KafkaServer)  
[2019-02-20 10:05:23,614] INFO Connecting to zookeeper on localhost:2181 (kafka.server.KafkaServer)  
[2019-02-20 10:05:23,692] INFO [ZooKeeperClient] Initializing a new session to localhost:2181. (kafka.zookeeper.ZooKeeperClient)  
[2019-02-20 10:05:23,708] INFO Client environment:zookeeper.version=3.4.13-2d71af4dbe22557fda74f9a9b4309b15a7487f03, built on 06/29/2016 10:05:23, built by rg.apache.zookeeper.ZooKeeper)
```

If everything is right, then the kafka server shall start on port 9092

Test your created cluster

Create a topic in kafka

Use below command to create a topic

```
kafka-topics.sh --create --zookeeper <zookeeper_host>:<zookeeper_port> --  
replication-factor <factor> --partitions <partition> --topic <topic-name>
```

```
Linux -> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1  
--partitions 1 --topic tutorial-topic
```

```
Windows -> bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --  
replication-factor 1 --partitions 1 --topic tutorial-topic
```

This will create a topic named "**tutorial-topic**" with the replication of 1

```
C:\work_dir\RND\kafka\kafka_2.12-2.1.0\bin\windows> kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic tutorial-topic  
Created topic "tutorial-topic".
```

Send records to newly created topic

```
Linux -> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic tutorial-topic
```

```
Windows -> bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --  
topic tutorial-topic
```

<type your message>

```
C:\work_dir\RND\kafka\kafka_2.12-2.1.0\bin\windows>kafka-console-producer.bat --broker-list localhost:9092 --topic tutorial-topic  
>This is test message for tutorial
```

Consume records using the console consumer

For quick testing lets start a handy console consumer, which reads messages from specified topic and displays them back on console.

```
Linux -> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic  
tutorial-topic --from-beginning
```

```
Windows -> bin\windows\kafka-console-consumer.bat --bootstrap-server  
localhost:9092 --topic tutorial-topic --from-beginning
```

```
Microsoft Windows [Version 10.0.16299.785]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\work_dir\RND\kafka\kafka_2.12-2.1.0\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic t
utorial-topic --from-beginning
This is test message for tutorial
```

Kafka Streams Concepts

KAFKA streams API is a java library that allows you to build real time application. A unique feature of the Kafka Streams API is that the applications you build with it are normal Java applications. These applications can be packaged, deployed, and monitored like any other Java application -- there is no need to install separate processing clusters or similar special-purpose and expensive infrastructure!

Its fast with milliseconds of latency, record at a time processing with high throughput.

Some of the key feature of Streams API are

Its powerful supporting highly scalable, fault tolerant, distributed stateless and stateful processing.

Its lightweight with low barrier of entry to run smoothly on local development, but powerful enough to support real time data processing at of large scale production systems like at UBER.

KAFKA Refresher

Before we have a deep dive in Kafka streams, just a quick refresher of kafka concepts

Producer, Consumer and Broker

Publishers publish data to kafka brokers and consumer consumes data from brokers. Both producer and consumer are decoupled from each other and run outside of broker.

Kafka Cluster

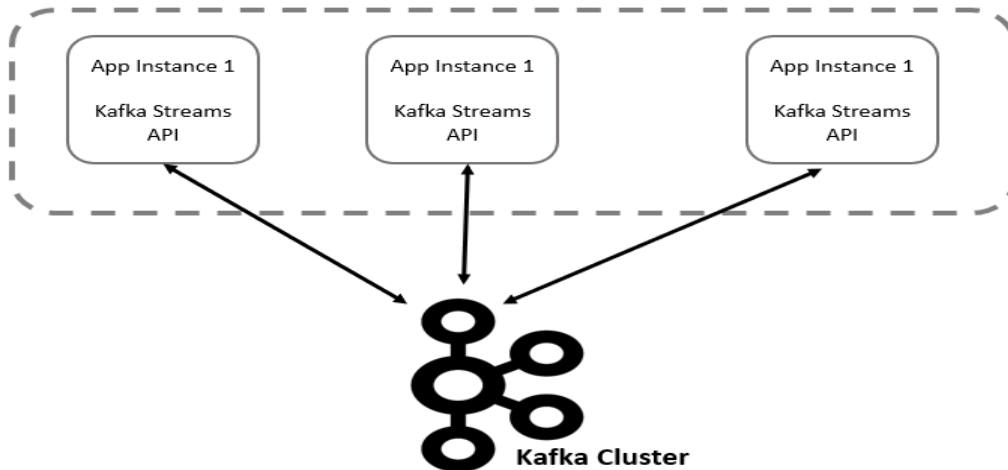
Kafka cluster or sometime referred as just cluster is a group of one or more Kafka brokers

Data

Data is stored in Kafka "*Topics*" and every topic is split into one or more "*partitions*"

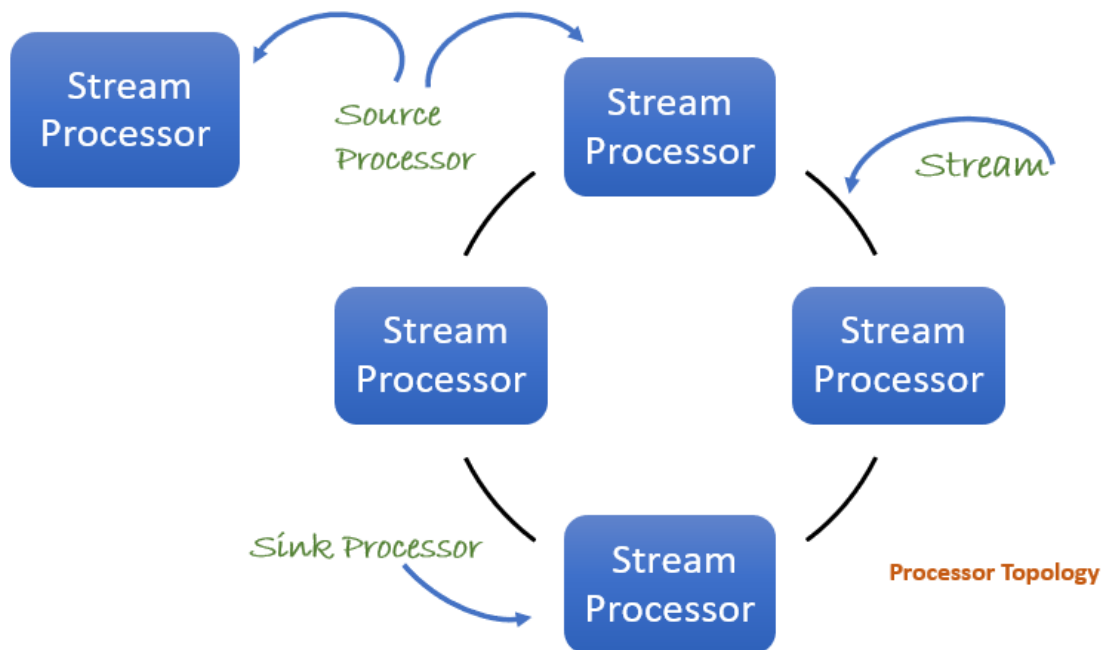
Stream

Stream is powerful abstraction provided by Kafka Streams. It represents an unbound continuously updating dataset. Just like a topic in Kafka, a stream in the Kafka Streams API consists of one or more stream partitions. A stream partition is an ordered, replayable, and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair.



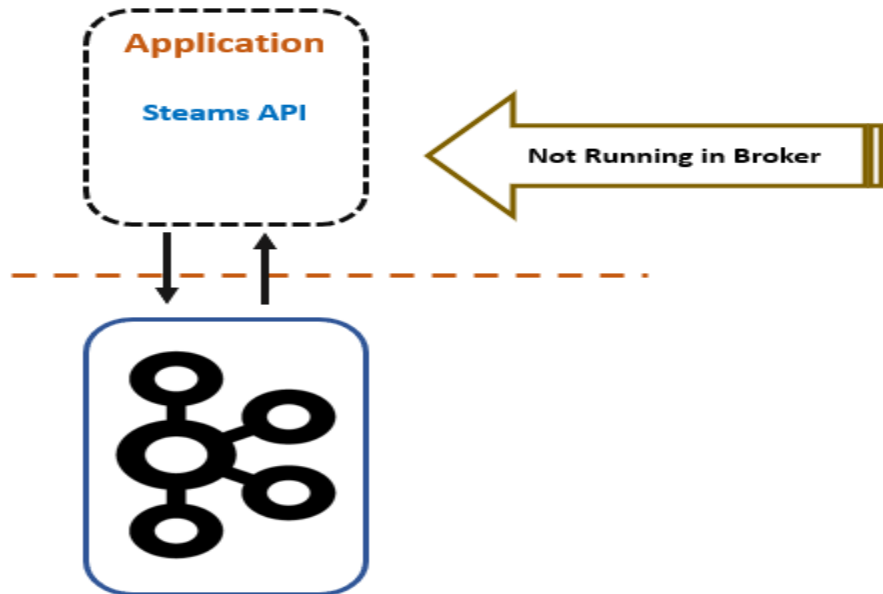
Processor Topology

A processor topology is a topology the computational logic for the stream. A topology is a graph of stream processors (nodes) that are connected by streams (edges). A topology can be defined by using high level stream DSL or low-level Processor API.



Stream application

A stream processing application is any program that uses Kafka streams library. In most of the cases it's the application created by you which can define computational logic in terms of processor topology. This application does not run inside broker. It runs in its own separate JVM, may be in separate cluster altogether.



Stream Processor

the stream processors (represented as nodes) represents a processing step in a processor topology. One of the most common application of node is to transform data. Standard operations such as map or filter, joins, and aggregations are examples of stream processors that are available in Kafka Streams out of the box. We have two options to define stream processors.

- Use high level Stream DSL provided by Kafka streams API
- For more fine grain control and flexibility use processor API. Using this api you can define and connect custom processors as well as directly interact with state stores.

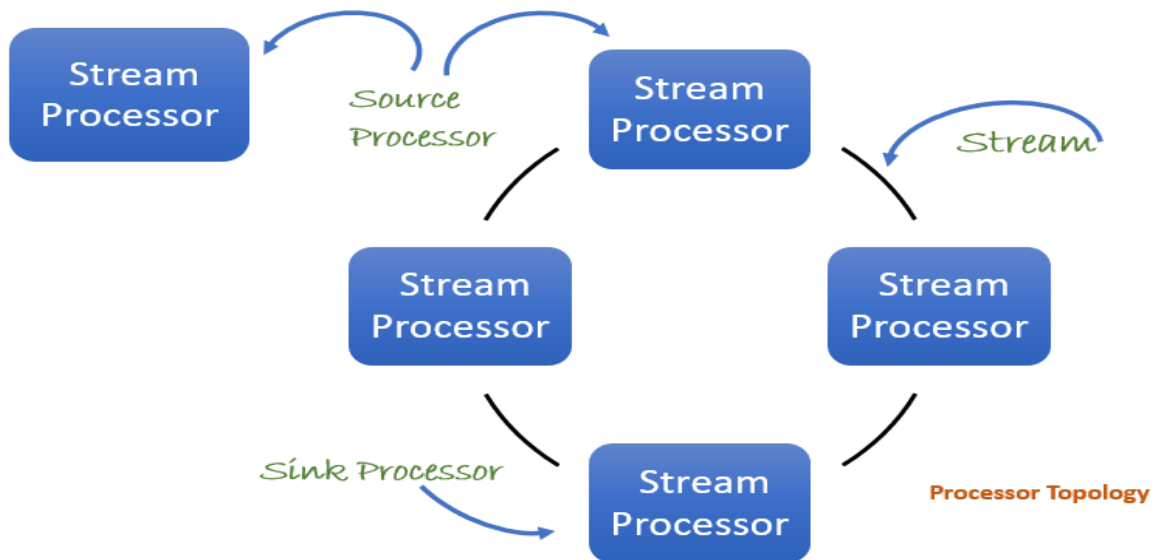
There are two types of processor

Source Processor

A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forward them to its down-stream processors.

Sink Processor

A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.



KStream

A KStream is an abstraction of a record stream, where each data record represents a self-contained datum in the unbounded data set. Data records are analogous to an "INSERT" -- think: adding more entries to an append-only ledger -- because no record replaces an existing row with the same key. Examples are a credit card transaction, a page view event, or a server log entry.

KTable

A KTable is an abstraction of a changelog stream, where each data record represents an update. The value in a data record is interpreted as an "UPDATE" of the last value for the same record key, if any (if a corresponding key doesn't exist yet, the update will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT aka INSERT/UPDATE because any existing row with the same key is overwritten. Also, null values are interpreted in a special way: a record with a null value represents a "DELETE" or tombstone for the record's key.

State Stores

Kafka Streams provides state stores, which can be used by stream processing applications to store and query data. This serves as an important capability when implementing stateful operations.

The Kafka Streams DSL, for example, automatically creates and manages such state stores when you are calling stateful operators such as `count()` or `aggregate()`, or when you are *windowing* a stream.

Time

A critical aspect in stream processing is the notion of time. There are three notions of time in Kafka Streams. Let's try to understand this with an example. Assume that there is a fleet management system which processes the data emitted by the GPS signal emitter fitted in vehicles.

Event Time

This shall be the point in time when the record is emitted by the device fitted in vehicles.

Ingestion Time

The time when the emitted record is stored in Kafka topics by the broker. Depending on the data communication mechanism the Event time and Ingestion time will vary. In most of the cases Ingestion-time is almost same as event-time, as a timestamp gets embedded in the data record itself.

Processing Time

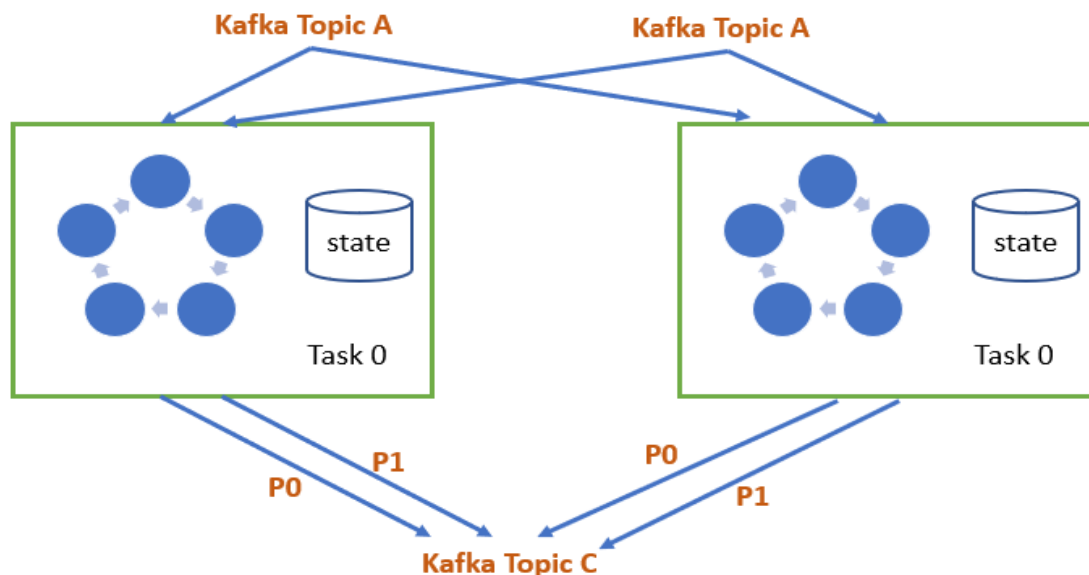
The point in time when the event or data record happens to be processed by the stream processing application, i.e. when the record is being consumed. The processing-time may be milliseconds, hours, or days etc. later than the original event-time.

Transformations on Kafka Streams

The KTable and KStream interface support a variety of transformations. These transformations belong to either of below two types.

Stateful Transformations

Stateless transformations do not require state for processing and they do not require a state store associated with the stream processor. Kafka 0.11.0 and later allows you to materialize the result from a stateless KTable transformation. This allows the result to be queried through Interactive Queries.



Two stream tasks with their dedicated local state stores

To materialize a KTable, each of the [stateless operations](#) can be augmented with an optional `queryableStoreName` argument.

Stateless Transformations

Kafka streams library offer lot of operations out of the box. Some of which are

Aggregation

An aggregation operation takes one input stream or table and yields a new table by combining multiple input records into a single output record. Examples of aggregations are computing counts or sum.

Joins

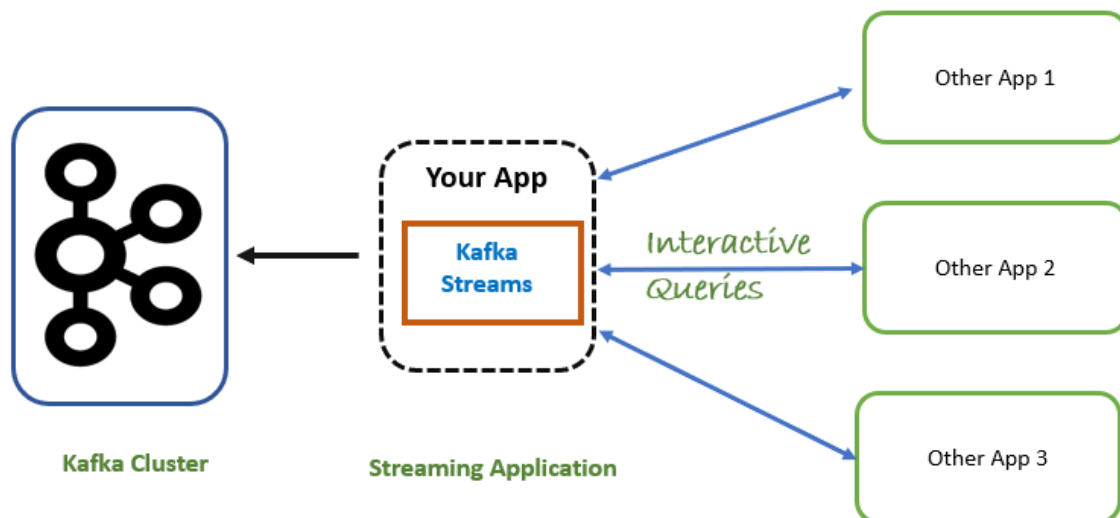
A join operation merges two input streams and/or tables based on the keys of their data records and yields a new stream/table. The join operations available in the Kafka Streams DSL differ based on which kinds of streams and tables are being joined; for example, KStream-KStream joins versus KStream-KTable joins.

Windowing

Windowing lets you control how to group records that have the same key for stateful operations such as aggregations or joins into so-called windows. Windows are tracked per record key.

Interactive Queries

Interactive queries allow you to treat the stream processing layer as a lightweight embedded database, and to directly query the latest state of your stream processing application. You can do this without having to first materialize that state to external databases or external storage.



Some of the examples for applications that benefit from interactive queries:

Real-time monitoring

A front-end dashboard that provides threat intelligence (e.g., web servers currently under attack by cyber criminals) can directly query a Kafka Streams application that continuously generates the relevant information by processing network telemetry data in real-time.

Risk and fraud detection

A Kafka Streams application continuously analyzes user transactions for anomalies and suspicious behavior. An online banking application can directly query the Kafka Streams application when a user logs in to deny access to those users that have been flagged as suspicious.

Trend detection

A Kafka Streams application continuously computes the latest top charts across music genres based on user listening behavior that is collected in real-time. Mobile or desktop applications of a music store can then interactively query for the latest charts while users are browsing the store.

Use Case: Fleet Management Dashboard

Scenario

Consider a hypothetical fleet management company needs a dashboard to get the insight of its day to day activities related to vehicles. Each vehicle with this fleet management company is fitted with GPS based geo location emitter, which emits a location data containing following information

- Vehicle Id: Unique id given to each vehicle on registration with the company.
- Latitude & Longitude: geo location information of vehicle
- Availability: The value of this field signifies weather the vehicle is available to take booking or not.
- Current Status (Online/Offline): Denotes weather the vehicle is on duty or not.

Functional Requirements

1. The dashboard shall represent the total number of vehicles online and offline at any point of time.
2. TBD

Solution

There could be multiple ways the above requirements could be met. But we will try to keep this simple and follow below steps to fulfill the need

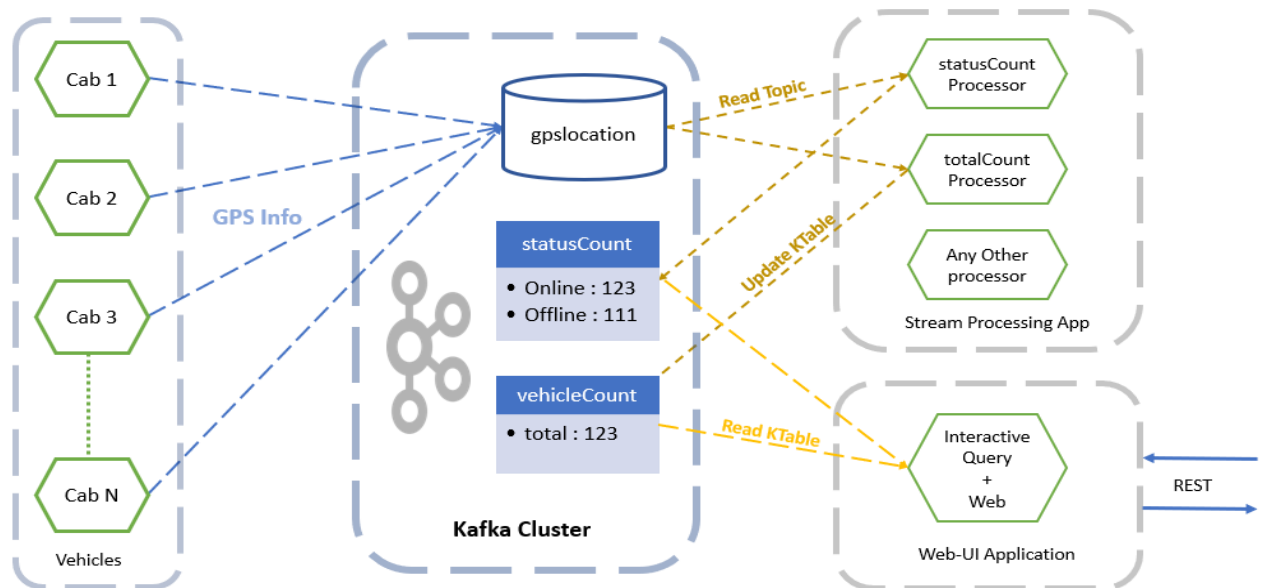
1. All the GPS signal will be sent to a topic.
2. Our stream processor will read the records from this topic and perform the required grouping, aggregation and materialize them to kafka state-stores.
3. The rest interface will be exposed which will read and server the data from kafka state-stores, which were created in earlier step.

Assumptions

Though we can publish the calculated value to topics and build auto-refreshing dashboards using web-sockets. For the sake of simplicity, we are only targeting on demand query of values, thus exposing simple REST endpoints.

We can add windowing operations for more real-time result. But we are processing all the events from the beginning

Architecture Diagram



Environment

Kafka (Topics & StateStores)

- **gpslocation:** A kafka topic that receive all the messages which are being emitted by the vehicles.
- **statusCount:** A state store which will maintain the "Online" and "Offline" vehicle counts. This is a key value store where the key will be status of vehicle and value will be count of the vehicle.
- **totalCount:** A state store which will maintain the count of vehicles. This is done by counting the unique vehicle Id's from all messages.

Use below command to create topic

"kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic"

Applications (spring boot)

As spring-boot offer's rich libraries to interact with kafka. We will try to implement following components as spring boot based applications.

- **common-libs:** this will contain the POJO classes to model the different objects like *Vehicle*, *VehicleCount*, *VehicleLocation*. This application will be added as dependency to other two applications eliminating the concerns of model mismatch.
- **vehicle-simulator:** the responsibility of this application will be to simulate the behavior of GPS signals being emitted by the device fitted in each vehicle.
- **tracker-dashboard:** this application will be our stream processing component. Though we can have separate application for querying the state-stores of Kafka, but for the sake of simplicity we add this functionality in this application itself.
- **vehicle-tracker:** The overall project wrapping all other applications as Modules.

Show me some Code

GPS event

Below is the POJO class to model a GPS event sent by vehicle.

```
11 /**
12  * @author satish-s
13  *
14  * Model class for location of a vehicle
15  */
16 @Getter
17 @Setter
18 @NoArgsConstructor
19 @AllArgsConstructor
20 public class VehicleLocation implements Serializable{
21
22     private int vehicleId; //unique id for each vehicle
23     private boolean online; //weather vehilce is online or offline
24     private boolean available; // is vehilcle ready to take bookings or not
25     private double latitude;
26     private double longitude;
27     public VehicleLocation(boolean online, boolean available, double latitude, double longitude) {
28         super();
29         this.online = online;
30         this.available = available;
31         this.latitude = latitude;
32         this.longitude = longitude;
33     }
34 }
```

Stream processor

Below is our stream processor to process the records form topic “*gpslocation*” and store them in a state store “*statusCount*”. The key in this state-store shall be

- i. Online: key for online vehicle count
- ii. Offline: key for offline vehicle count

```
24 @Component
25 public class VehicleStatusCountProcessor {
26     @Autowired
27     private KafkaProperties kafkaProperties; //default properties
28
29     @Bean //configure key-value serdes
30     public Map<String, Object> consumerConfigs() {
31         Map<String, Object> props = new HashMap<>(kafkaProperties.buildProducerProperties());
32         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, IntegerDeserializer.class);
33         props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, JsonSerde.class);
34         return props;
35     }
36
37     @Bean
38     public KStream<String, Long> statusCountStreamProcessor(StreamsBuilder streamsBuilder) {
39         KStream<Integer, VehicleLocation> stream = streamsBuilder.stream("gpslocation", //Read from topic
40             Consumed.with(Serdes.Integer(), new JsonSerde<>(VehicleLocation.class))); //using Integer and JSON serde
41         return stream.map((k,v)-> {
42             // transform they key as Online/Offline based on status
43             String online = v.isOnline() == true ? "Online" : "Offline";
44             return new KeyValue<>(online, v);
45         })
46         .groupByKey(Serialized.with(Serdes.String(), new JsonSerde<>(VehicleLocation.class))
47             )
48         .count(Materialized.as("statusCount")) // materialize this value to state store
49         .toStream();
50     }
51 }
52 }
```

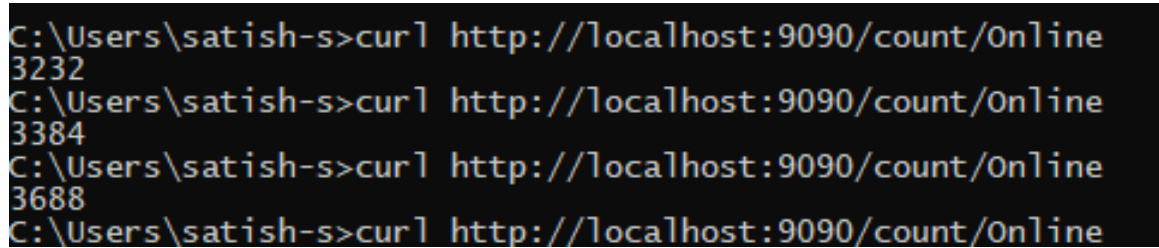
Query the store

Below is the code for REST interface to get the state store and query for specific key (Online/Offline).

```
14
15 @RestController
16 public class VehicleQueryService {
17     @Autowired
18     private StreamsBuilderFactoryBean kStreamBuilderFactoryBean;
19
20     @GetMapping("/count/{status}")
21     public long getVehicleCountForStatus(@PathVariable("status") String status) {
22         // Get the state-store
23         ReadOnlyKeyValueStore<String, Long> keyValueStore= kStreamBuilderFactoryBean.getKafkaStreams()
24                                                         .store("statusCount", QueryableStoreTypes.keyValueStore());
25         return keyValueStore.get(status); //get the count for the key viz. Offline/Online
26     }
27 }
```

Response for Offline & Online vehicle count

Query local host URL exposed by the tracker-dashboard application for total events received as "Online"



```
C:\Users\satish-s>curl http://localhost:9090/count/Online
3232
C:\Users\satish-s>curl http://localhost:9090/count/Online
3384
C:\Users\satish-s>curl http://localhost:9090/count/Online
3688
C:\Users\satish-s>curl http://localhost:9090/count/Online
```

You should be able to get Online/Offline vehicle count changing as the records are published.

Hope this article left you with some insights about Kafka and its Stream processing capabilities

Full code available at this [git repo](#). Please feel free to share your valuable feedback at Satish-s@hcl.com