Kafka with ELK on swarm

Mikor elindítunk egy Kafa példányt, akkor valójában egy kafka brokert indítunk el. A brokereknek van száma, ezeket nem lehet konfigurációból felskálázni. Ha producer-ek mindig egy brokerhez csatlakoznak. A teljes konfiguráció zookeeper-ben van tárolva. A zookeeper tudja értesíteni a klienseket ha a konfiguráció változik, ezért hamar elterjed a hálózaton a változás.

A producer-ek egy megadott topic-kra dobálják be az üzeneteket, amit onnan a consumer-ek kiolvasnak. Egy topic tetsz?leges számú partícióból állhat. Egy partíció az a logikai egység, aminek rá kell férnie egy lemezre. A topic-kot úgy kell felskálázni, hogy egyre több partíciót adunk hozzá, amik különböz? brokereken fognak létrejönni. Minden partíciónak lehet egy vagy több replikája, amik biztonsági másolatok. Mikor a producer beküld egy üzenetet egy partícióba, akkor fog committed üzenetnek min?sülni, ha minden replikára is eljutott.

Azt, hogy egy producer melyik partícióba dobja az üzenetet vagy a kulcs határozza meg, vagy round-rubin módon mindig egy másikba teszi. Ha van kulcs, akkor az abból készült hash fogja meghatározni, hogy melyik partícióba kerüljön. Ugyan az a kulcs így mindig ugyan abba a partícióba fog kerülni. De a kulcs nem kötelez?. A sorrend tartás csak egy partíción belül garantált, de ott nagyon. Ha nagyon kritikus bizonyos üzenetek sorrendje, akkor azokat egy partícióba kell rakni azonos kulcsot használva. Loggolásnál ez nem kritikus, egyrészt mert a logstash sorba rakja az üzeneteket, másrészt mikor elastichsearch-be szúrjuk, ott a dátum lesz az egyik attribútum, ami alapján már sorba lehet majd újra rendezni a logokat. Az meg amúgy sem kritikus, ha a log egy része enyhe csúszással kerül be az adatbázisba, lényeg, hogy végül helyes lesz a sorrend.

A comsumer-eket úgynevezett consumer-group-okba szervezzük az azonosítójuk szerint. Egy csoport mindig ugyan azon topic üzeneteit olvassa, de minden egyes consumer a csoporotban más és más partícióból. Minden partíció csak egy consumer-hez rendelhet? hozzá egy csoporton belül. De ha nincs annyi consumer a csoportban mind ahány partíció, akkor egy consumer több partíciót is fog olvasni. Viszont ha több consumer van mint partíció egy csoportban, akkor bizonyos consumer-ek mindig idle állapotban lesznek. Minden csoporton belül van egy vezet? consumer, általában az aki el?ször csatlakozott. ? teríti a többieknek a cluster információkat. A kafka nem tudja értelmezni sem a kulcsot sem az üzenetet. Ez számára egy bájt tömb. Az, hogy egy objektumból hogy lesz bájt tömb kulcs és bájt tömb üzenet a producer-ben lév? serializátor dolga. A consumer-ben pedig a deserializázor dolga, hogy a bájt folyamból újra értelmes objektumot állítson el?.

minden partíció újabb üzenete mindig a partíció végére íródik. A partíció elejét?l számoljuk az üzenetek sorszámát, ezt hívjuk offset-nek. Mikor egy consumer kiolvas egy üzentet, attól az még ott marad a partícióba egészen addig, amíg len nem jár, alapértelmezetten ez egy nap. Tehát ez eltér a hagyományos sor kezelést?l. A Kafka nyilvántartja, hogy melyik consumer egy adott partícióban melyik offset-nél tartott. Ezt egy speciális topic-ban tartja nyilván: "__...". Ha újra is indul a világ, akkor is tudni fogják a consumer-ek hogy hol tartottak, és onnan folytatják.

Producer -> serializator -> partitioner -> batch (partícionként) ->|idáig tart a producer | kafa broker

Producer:

- bootstrap.servers: a brokerek listája. Itt nem kell az összes broker-t felsorolni, mert ha már az egyikhez hozzá tud csatlakozni, az elküldi a teljes cluster topológiát. Viszont ajánlatos többet megadni hibat?rés céljából.
- key.serializer: Akkor is meg kell adni, ha nem használunk kulcsot. A kulcs is mindig bájt tömb, ezért a serializátor implementációnknak bájt tömböt kell visszaadni. Vannak beépítettek a java kliensbe a java alaptípusokra.
- value.serializer

Contents

zookeeper

A zookeeper egy elosztott konfiguráció manager, ...

a zookeeper cluster-t ensemble-nek (ánszámbol) hívják ami együttest jelent. A quorum (minimum létszám a szavazás képességhez) miatt mindig páratlan számú node-ot kell a cluster-be rakni, 3-at, 5-öt vagy 7-et. Hét fölé már performancia okokból nem érdemes menni. 3 esetén 1 node kiesését viseli el a cluster, 5 node esetén 2-öt.

Mi egy három tagú zookeeper ensembe-t fogunk készíteni. Minden egyes cluster tagnak ugyan az kell legyen a konfigurációja. </br> /conf/zoo.cfg

```
clientPort=2181
dataDir=/data
dataLogDir=/datalog
tickTime=2000
initLimit=5
syncLimit=2
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
server.1=zookeeper1:2888:3888
server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
```

- **clientPort**: Ez az a port, ahol a Kafka broker-ek csatlakozni fognak a zookeper node-okhoz. Ez is a közös swarm overlay hálózaton lesz elérhet? a broker-ek számára.
- **server.X=zookeeper1:peerPort:leaderPort**

- **X**: Az X egy egész szám lehet csak, fontos hogy egyedi legyen.
- **peerPort**: A ensemble node tagok ezen a porton kommunikálnak egymással. A node-ok is a közös swarm overlay hálózaton érik el egymást. Az itt megadott host név a swarm service neve, amit a swarm fel tud oldani.
- **leaderPort**: Ezen a porton zajlik a vezet? választás

A kafka cluster jelentéktelen terhelést fog eredményezni a zookeeper ensemble (cluster)-ben. Ezért egy központi zookeeper cluster általában elég egy szervezeten belül. Viszont arra figyelni kell, hogy ha lehet a zookeepr ensemble más alkalmazásokat ne szolgáljon ki csak a Kafa cluster-eket. A Kafka broker-ek allergiásak a zookeeper kommunikációban fellép? késleltetésre. Ha nem kapnak közel azonnal választ, akkor úgy gondolhatják, hogy bizonyos node-ok nem elérhet?ek és széteshet a cluster.

A consumer-ekben be lehet állítani, hogy az aktuális offset értéküket a Kafak-ban egy erre a célra fenntartott topic-ban tárolják (__consumer_offsets) vagy zookeeper-ben. Az utóbbi nem ajánlatos, mert ezzel is túlterhelhetjük a zookeeper ensemble-t (cluster-t).

Web-GUI

https://github.com/qiuxiafei/zk-web

/app/zk-web/conf/zk-web-conf.clj

```
{
 :server-port 8080
 :users {
         "admin" "12345"
         ;; map of user -> password
         ;; you can add more
         }
 :default-node "zookeeper1:2181/"
}
```

Egyszerre mindig csak egy zookeeper node-hoz tud csatlakozni, de új kapcsolatokat meg lehet adni a web-es gui-n keresztül. A default-node paraméterrel meg lehet adni, hogy melyik zookeeper node legyen az alapértelmezett az web-gui indulásakor. Ezt amúgy csak akkor kell a felületen átállítani egy másik node-ra, ha a zookeeper1 kiesne karbantartás vagy hiba miatt.

Swarm stack

```
version: '3'


services:
  zookeeper1:
    image: zookeeper
    networks:
      - kafa-net
    volumes:
      - "zookeeper1-conf:/conf"
      - "zookeeper1-data:/data"
      - "zookeeper1-datalog:/datalog"
    deploy:
      placement:
        constraints:
          - node.role == worker
      restart_policy:
        condition: on-failure
      resources:
        reservations:
          memory: 100m
...
  zookeeper-gui:
    image: tobilg/zookeeper-webui
    networks:
      - kafa-net
    volumes:
      - "zookeeper-gui:/app/zk-web/conf"
    ports:
      - 8089:8080
    deploy:
      placement:
        constraints:
          - node.role == worker
      restart_policy:
        condition: on-failure

networks:
  kafa-net:
    driver: overlay

volumes:
  zookeeper1-conf:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/dockerStore/zookeeper/node1/conf/
...
  zookeeper1-data:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/dockerStore/zookeeper/node1/data/
 ...
  zookeeper1-datalog:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/dockerStore/zookeeper/node1/datalog/
 ...
  zookeeper-gui:
    driver: nfs
    driver_opts:
      share: 192.168.42.1:/home/adam/dockerStore/zookeeper/zk-web/
```

Produkciós futtatás

In a typical production use case, a minimum of 8 GB of RAM should be dedicated for ZooKeeper use. Note that ZooKeeper is sensitive to swapping and any host running a ZooKeeper server should avoid swapping.

you should consider providing a dedicated CPU core to ensure context switching is not an issue.

Disk performance is vital to maintaining a healthy ZooKeeper cluster. Solid state drives (SSD) are highly recommended as ZooKeeper must have low latency disk writes in order to perform optimally. Each request to ZooKeeper must be committed to to disk on each server in the quorum before the result is available for read. A dedicated SSD of at least 64 GB in size on each ZooKeeper server is recommended for a production deployment

ZooKeeper runs as a JVM. It is not notably heap intensive when running for the Kafka use case. A heap size of 1 GB is recommended for most use cases and monitoring heap usage to ensure no delays are caused by garbage collection.

Kafka

Configuration

- Broker Configuration
- Topic Defaults
- Hardware Selection

Avro

Avro: serializer, desierializer:

http://cloudurable.com/blog/avro/index.html

Avro supports direct mapping to JSON as well as a compact binary format. It is a very fast serialization format. Avro is widely used in the Hadoop ecosystem. Avro supports polyglot bindings to many programming languages and a code generation for static languages. For dynamically typed languages, code generation is not needed. Another key advantage of Avro is its support of evolutionary schemas which supports compatibility checks, and allows evolving your data over time.

Schema registry

https://dzone.com/articles/kafka-avro-serialization-and-the-schema-registry

The Kafka producer creates a record/message that is an Avro record. The record contains a schema ID and data. With the Kafka Avro Serializer, the schema is registered if needed and then it serializes the data and schema ID. The Kafka Avro Serializer keeps a cache of registered schemas from the Schema Registry their schema IDs.

Consumers receive payloads and deserialize them with Kafka Avro Deserializers, which use the Confluent Schema Registry. The Deserializer looks up the full schema from the cache or Schema Registry based on ID. You can manage schemas via a REST API with the Schema registry. We will need to start up the Schema Registry server pointing to our ZooKeeper cluster

Írni kell pl JAVA-ban olyan Kafka Producer-eket és és Kafa Consumer-eket, amikbe beállítjuk hogy a serializáló és deserializáló az a Avro lesz:
Properties props = new Properties();

```
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "AvroProducer");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                LongSerializer.class.getName());
        // Configure the KafkaAvroSerializer. <<--------- itt mondjuk meg hogy az Avro-t használja
       props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        // Schema Registry location. <<------- itt állítjuk be a Schema reigstry-t az Avro-nak.
        props.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");
```

Aztán ezt a Kafa producer-t ugyan úgy kell használni, mint ha nem Avro-t használnánk, de a Comsumer-nek ugyan így kell majd deserializálnija.

Log4J with Kafa

https://logging.apache.org/log4j/2.x/manual/appenders.html#KafkaAppender

```
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-log4j-appender</artifactId>
<version>1.0.0</version>
</dependency>
```

AVRO

http://avro.apache.org/docs/current/spec.html#schemas

```
{"type": "typeName" ...attributes...}
```

A typeName értéke lehet primitív típus vagy összetett típus.

Összetett típusok: records, enums, arrays, maps, unions, fixed

A record típus

- name: a JSON string providing the name of the record (required).
- namespace, a JSON string that qualifies the name;

- doc: a JSON string providing documentation to the user of this schema (optional).
- aliases: a JSON array of strings, providing alternate names for this record (optional).
- fields: a JSON array, listing fields (required). Each field is a JSON object with the following attributes:
    - name: a JSON string providing the name of the field (required), and
    - doc: a JSON string describing this field for users (optional).
    - type: A JSON object defining a schema, or a JSON string naming a record definition (required).

```
package com.ippontech.kafkatutorials;

class Person(
        String firstName;
        String lastName;
        Date birthDate;
)
```

```
{
  "type": "record",
  "name": "Person",
  "namespace": "com.ippontech.kafkatutorials",
  "fields": [
    {
      "name": "firstName",
      "type": "string"
    },
    {
      "name": "lastName",
      "type": "string"
    },
    {
      "name": "birthDate",
      "type": "long"
    }
  ]
}
```

https://docs.confluent.io/current/installation/docker/config-reference.html

https://kafka.apache.org/quickstart