Daniel Gergely 03/09/2015

Assignment

Introduction

This assignment shows how Storm and Flink solves the issue of reading from a Kafka topic, making a decision based on the read message and writing back to Kafka.

Task

Read from kafka topic *neverwinter*, each message is a serialized form of an avro schema. It contains a field called *random*, that has a value of 1, 2 or 3. By using Storm and Flink read the messages from the *neverwinter* topic and based on the value of *random* field, save it to the topics *random1*, *ranom2*, *random3*. Then measure the performance of the solutions.

Requirements

In order to run the applications, Linux operating system is required with the following programs:

- Java JDK 7 or 8
- Maven 3.0.5
- Curl

The script `setup_requirements.sh` can be used to install these applications.

Docker environment

In order to have a running instance of Kafka and Zookeeper, a docker image was created. Making installation easier, there is a script to handle everything in the folder of `kafka_image`:

`setup.sh`

Parameters:

- `-h` or `--help:` help message for parameters
- `-i`: install docker and add the current user to the docker group
- `-b`: build docker image with the tag of: `nventdata/kafka_dani`
- `-r`: run the image with the name of `kafka_dani` and do port bindings for `localhost:9092` and `localhost:2181`

If no parameters are set, then all of them are enabled.

First run with only `-i`, because it is needed to sign out an in again at the end of installing docker. Then it can be run by `-r -b`.
As a result Zookeeper can be accessed on `localhost:2181` and Kafka can be accessed on `localhost:9092`.

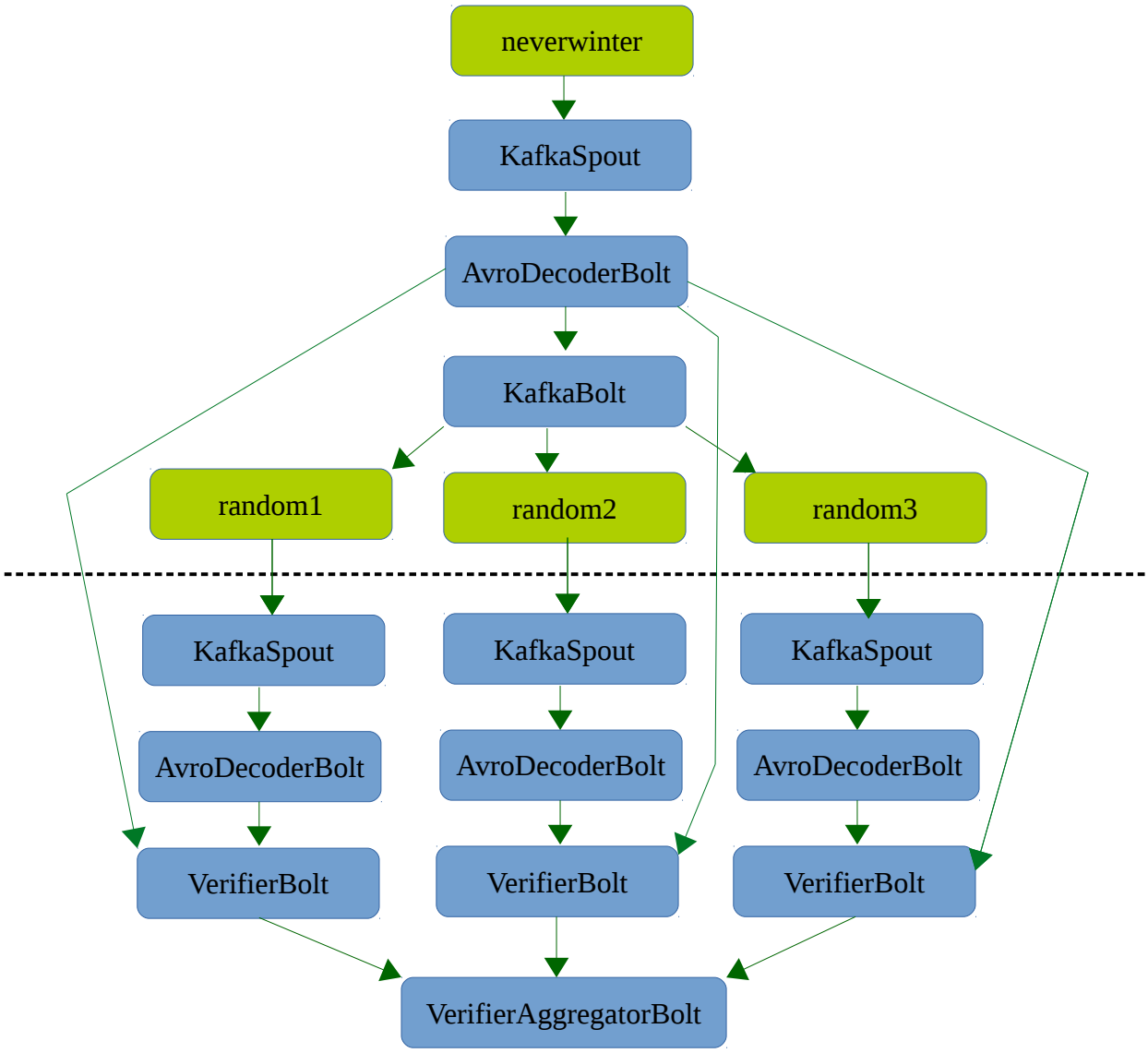
Storm

Storm has building blocks of spout and bolt. Spouts are data sources and bolts are for transforming data or do actions with it.

There are 2 approaches to solve this issue. Both of them read a kafka messages from the source topic and deserialize them to a GenericRecord of avro to get the value from the *random* field. The difference comes from this point: one of the solutions forwards the raw kafka message (bytearray), the other one forwards the GenericRecord object.

The bytearray solution can have better performance, because it is not needed to be serialized before storing it to the output kafka topic. However the object solution can be more flexible, since the deserialized object can be used.

Architecture



KafkaSpout reads the messages from the source topic and forwards it to the AvroDecoderBolt, which deserializes the messages and makes a tuple of <*random field value*, *message*>. Based on the

random field value, KafkaBolt sends the messages to the proper output topic by using a topic selector. This topic selector returns the topic's name based on the received tuple. In the object type solution there is an extra deserialization step in the KafkaBolt to obtain raw kafka message from GenericRecord object.

Verification part is below the dashed line. This can be switched off by a flag in the configuration. The output topics are read, each of them has a KafkaSpout and an AvroDecoderBolt to get the message and the value of the *random* field. Verifier bolts deal with 2 sources, one of them is the AvroDecoderBolt output from the source topic, and the other one is from the AvroDecoderBolt of the given target topic. So basically it should get every message twice. Input from the source topic is filtered in this VerifierBolt, only those messages are checked that has the given random value (e.g. on the branch where the topic is *random2*, VerifierBolt filters the messages from the source topic and allows only those that have the value 2 in the *random* field.)

VerifierBolt counts the messages and handles the pairs. The messages are verified, if all the messages are received from both inputs. Eventually a VerifierAggregatorBolt groups the results together and logs the results.

Parallelism

Storm use multiple threads to make the work more efficient. For my computer I set up 4 threads, so 4 KafkaSpouts, 4 AvroDecoderBolts and 4 KafkaBolts were working. Verification is done by one thread per output topic.

Flink

Flink can be used for solving similar issues, it handles streams and offers different types of transformation methods. Flink uses a higher level api than Storm, so there might be problems that can be solved more effectively by using Storm.

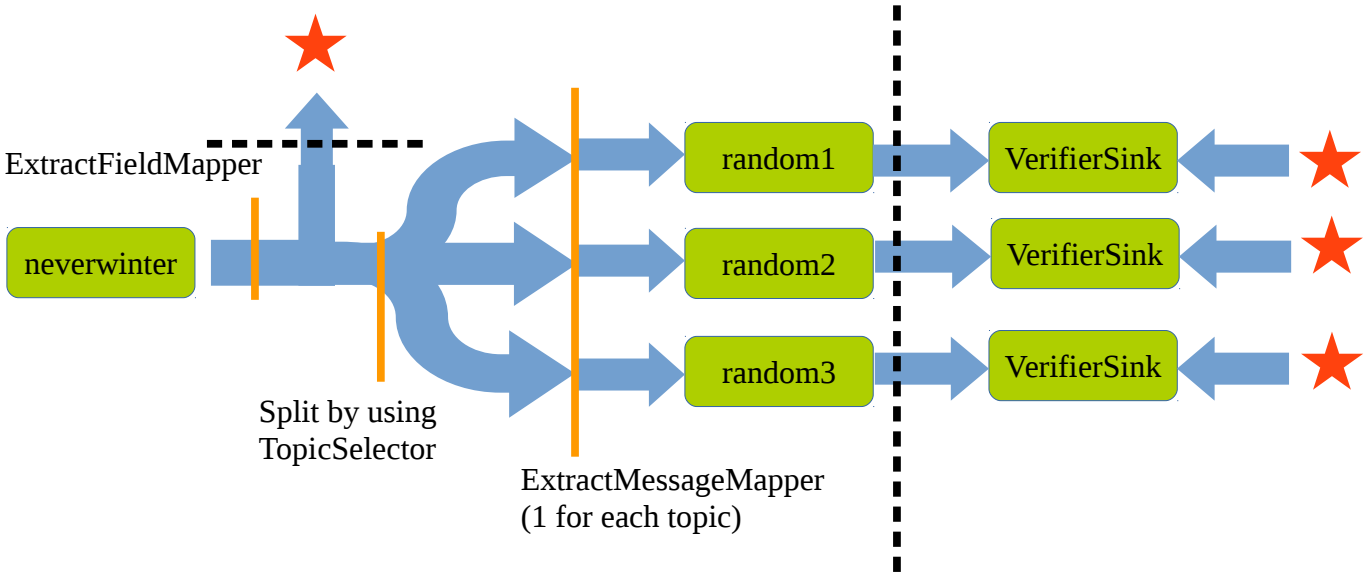
Flink has the building blocks source and sink. A source returns a stream, on this stream operations can be done, that modifies the stream, finally a sink receives the stream.

Architecture

As the architecture shows, there is a source for kafka topic *neverwinter*, then an *ExtractFieldMapper* creates a tuple from the input message: <*random field value*, *kafka message*>. In the case of the object version, kafka message is a GenericRecord, in the case of bytearray version, it is a bytearray. Then the stream is splitted into 3 and every message is directed to the right one. Finally the tuple is reduced to the kafka message only and written out to the proper topic. (the object version does the serialization just before writing out)

Verification is similar to the solution mentioned at Storm. The stream from the source topic and the stream from a target topic is merged. Before the merge, source stream is filtered to allow only the messages with the proper random field value. Then the messages are handled by a VerifierSink, which check if all the messages arrives exactly twice.

In the following figure the verification part is over the dashed line, it can be switched of by a configuration flag.

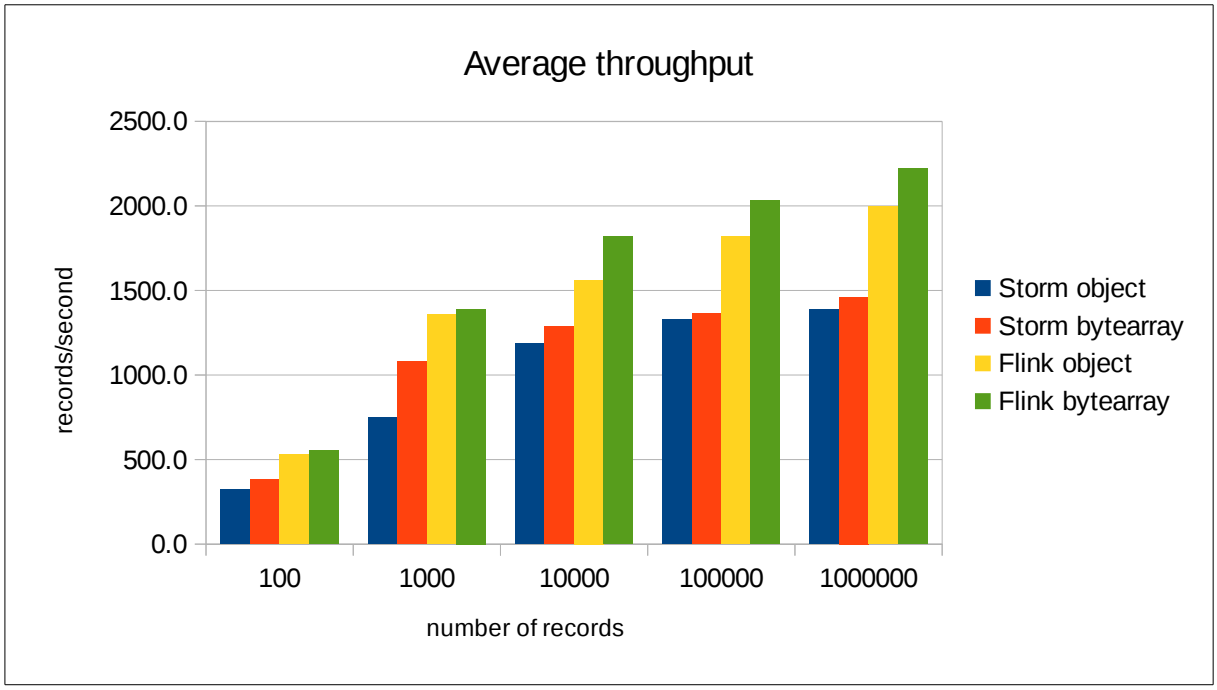


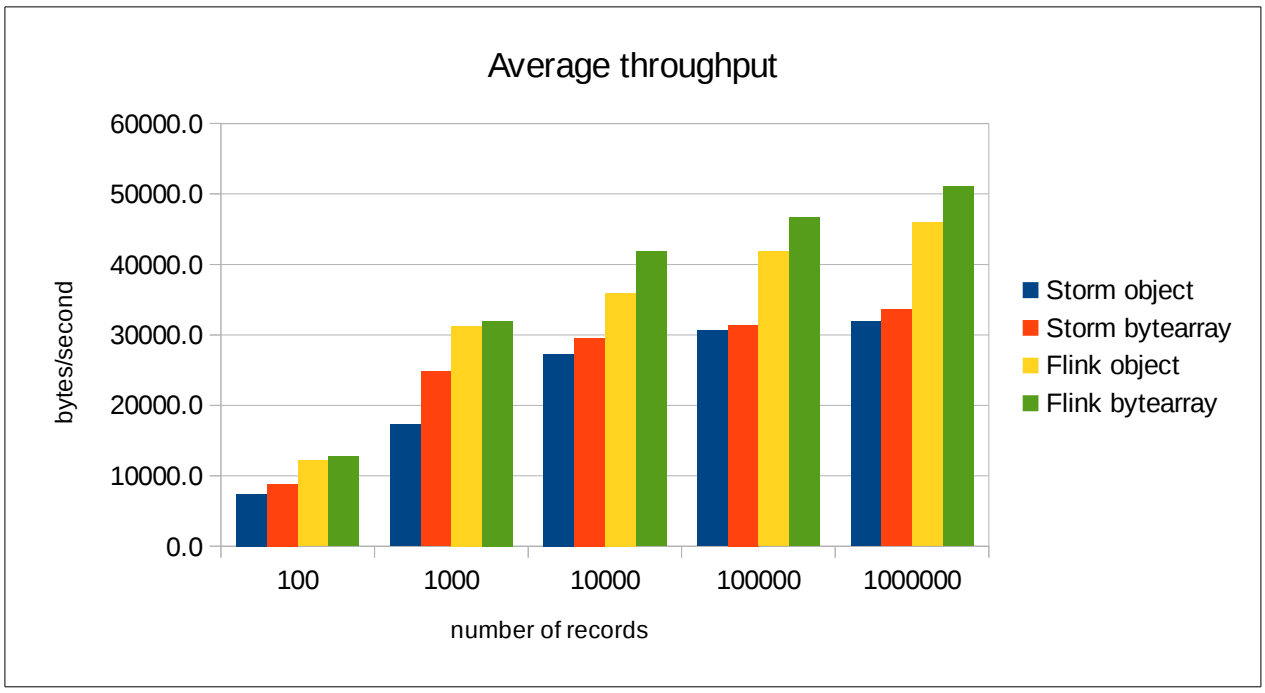
Parallelism

By default Flink uses the number of processor cores as the degree of parallelism. Each thread has its own source and sink, except VerifierSink, each thread has only 3 of them (for the 3 output topic) instead of 12 to make it easier to verify results.

Performance

Performance measurements were done by using all the 4 application (Storm/Flink and object/bytearray versions). The environment was a virtual Ubuntu Linux 15.04 with 8GB of RAM, 2 core CPU with HT, SSD. Kafka generation script and docker were on the same machine where the applications ran. Both Flink and Storm was run on a local cluster.





The measurement of the process starts before deserializing kafka message of the source topic, and ends after sending it to the target kafka topic.

In the figures it can be seen that Flink is around 50% faster than Storm. This can be because Flink uses a higher level api, so building blocks are simpler, so can be better optimized. Moreover the Storm application implementation might not be as optimal as it could be.

It also can be seen, that the deserialization step reduces performance by around 5-10%.

The bottleneck during the processing was the CPU.

As a first approach a single python kafka producer was used, but it turned out that it is to slow and

that is the bottleneck, so I started 4 to increase performance.

All the measurement details can be found in the other document.

Ideas to improve results

- Using separated non-virtual computers for running Kafka, producing messages and running these applications. In this case network overhead can have an impact on performance.
- Running application on a grid or cluster to benefit from the distributed resources.
- Using headless and very thin operating systems to be able to utilize CPU and RAM better.
- Learning more about Storm and Flink to know the frameworks better and create better implementations.

Running the applications

All the applications can be compiled by running `compile.sh` and can be started by running `start.sh`.

After compiling the application, JAR files can be obtained in the target folder. For Flink this is called `flink-0.0.1-SNAPSHOT-jar-with-dependencies.jar`, for Storm there are 2 JARs, one for running it with local cluster (`storm-0.0.1-SNAPSHOT-jar-with-dependencies.jar`) and one that can be uploaded to Storm as a topology (`storm-0.0.1-SNAPSHOT-jar-with-dependencies-storm.jar`). To upload it to Storm, use the command

```
storm jar path/to/storm-0.0.1-SNAPSHOT-jar-with-dependencies-storm.jar
nventdata.storm.runnable.StormTopology path/to/config.xml
```

Configuration file

As default the configuration file should be next to the `start.sh` when starting the application. This is an xml file with the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
      <comment>Application configuration</comment>
      <entry key="zookeeper.rootpath"></entry>
      <entry key="zookeeper.url">localhost:2181</entry>
      <entry key="kafka.url">localhost:9092</entry>
      <entry key="kafka.consumerid">consumer_1</entry>
      <entry key="kafka.sourcetopic">neverwinter</entry>
      <entry key="kafka.targettopicprefix">random</entry>
      <entry key="avro.fieldname">random</entry>
      <entry key="avro.valueset">1,2,3</entry>
      <entry key="verify">true</entry>
      <entry key="performance">false</entry>
      <entry key="local">true</entry>
</properties>
```

This configuration file can be used for all the applications:

- **zookeeper.rootpath**: root path for zookeeper, empty string as default
- **zookeeper.url**: the url for zookeeper, localhost:2181 as default
- **kafka.url**: the url for kafka, localhost:9092 as default
- **kafka.consumerid**: cunsumer id for kafka, comsumer_1 as default
- **kafka.sourcetopic**: name of the source topic, neverwinter as default
- **kafka.targettopicprefix**: name prefix for the output topics
- **avro.fieldname**: field name in the avro schema, based on the value of this field, the separation is done, random as default
- **avro.valueset**: possible values for the random field, separated by comma, empty string as default
- **verify**: flag for doing verification or not (verify and performance should not be enabled in the same time), false as default
- **performance**: flag for doing performance test or not (verify and performance should not be enabled in the same time), false as default
- **local**: has effect with Storm. If enabled, local cluster is created, otherwise it is for uploading to a storm cluster as a topology, true as default

Simple scenario

1. Install docker, build image and run it: `kafka_image/setup.sh`
2. Compile maven project by calling: `storm/compile.sh`
3. Change `config.xml` if you want and start application by using: `storm/start.sh`

Logging

In order to get information about the results and runtime details, logs are created. Each application has 2 log files, that contains different subset of messages:

- `flink.log` and `storm.log`: contains the relevant information for the application (general information, debug messages, verification and performance test messages).
- `flink_root.log` and `storm_root.log`: contains log messages of all the other components.

Obtaining results

Performance measurement and verification results can be found in the `flink.log` or `storm.log` file. For performance measurement, the logged information is the average throughput in records/second and bytes/second.

In the case of verification, the logged lines contains a timestamp, the topic name and the status of verification. The messages are verified if the last logged line for each topic is “verified”.

Summary

Flink and Storm are two great tools that can be used for stream processing. While Storm has a low level api and can control things in a more detailed way, Flink has a less complex dataflow and by using transformations on streams, complex things can also be built.

I would use Flink for simple applications and Storm for complex computational tasks.

After measuring the performance, I found that Flink is around 50% faster than Storm on my configuration, but I am sure that improving the implementation, this difference can be decreased.

This was the first time I used Kafka, Zookeeper, Flink and Storm, and eventually I managed to implement the task in time. Learning more about these technologies would really help me to improve my skills.