Complex workflows implementation in stream data processing

Accepted 6th December, 2020

ABSTRACT

Real time response is critical in many areas and requires fast processing of streaming data continuously coming from different sources. This study presents some approaches and implementation issues for developing complex workflows in stream data processing, which is one of the main challenges in big data processing.

Key words: Stream data processing, workflows, big data, parallel processing, scalability, persistent strategy, event-driven applications, fault tolerance.

INTRODUCTION

Currently, when data become too big, usually in terms of Volume, Velocity and Variety (3V) (Ankur, 2016), the challenge is how to process it in order to discover knowledge.

In most cases, the big data are located on distributed computer systems, so parallel processing is required.

One of the main challenges is the real time response, which requires fast processing of streaming data (Barga, 2016) coming continuously from different sources.

Real time processing and immediate action are critical in many areas such as Security (Fast response to cyber-attacks, etc.), Healthcare (Vital functions monitoring), Environment (Fast response in case of disasters, etc.), Gaming and Gambling (Real time user activity monitoring), e-Commerce (Finding the lowest price), e-Media (Personalization), Transport and Industry (Fault detection and prevention), IoT, etc.

The stream data is a kind of data that is continuously coming and is like a stream as opposed to batch processing where the data is more like a lake. Stream processing systems should provide low latency, which is necessary for the real time processing and in the same time high throughput, but those two requirements are antagonistic.

There are many studies and comparison analysis between stream data processing systems (Chintapalli et al., 2016, 2015), but there are still no widely accepted benchmarks.

In general, batch processing provides best throughput, but also a higher latency.

Apache Spark streaming (https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html”) uses micro-batches to mimic streams and when the batches are small enough, the low latency requirements are partially covered. This approach approximates a real time processing, but the underlying technology is batch-processing (Figure 1).

Stream processing frameworks, such as Apache Flink (https://flink.apache.org) and Apache Storm (http://storm.apache.org), are originally designed for such kind of processing (Figure 2). They use native streaming model providing low latency and many useful features (Ellen Friedman, 2015).

The existing frameworks provide high level of abstraction and significantly simplify the development process. At present, it is relatively easy to build distributed “Hello World” application using high level APIs and there are plenty of tutorial on how to do it, but the real applications are more complex and there are many options to consider and take into account.

Therefore, the aim of the present study is to present some principle and experimental issues of building complex workflow in stream processing systems.

SYSTEM ARCHITECTURE AND DEVELOPMENT

Before dealing with the architecture and development issues, we have to consider some important requirements
such as large-scale distribution, exact consistency, scalability, low latency, high throughput, fault tolerance, easy to use, etc.

To satisfy the aforementioned requirements, we need some kind of computer cluster and distributed programs to run on it. The computer cluster is a set of loosely coupled computers and there are many options on how to set it up.

Developing of distributed systems from scratch requires a lot of effort and solid knowledge in computing, so at present, there are many tools providing high level of abstraction and simplification of the process. One emblematic example is the map-reduce programming model (Dean and Ghemawat, 2008).

The recent streaming data frameworks provide even higher level of abstraction.

There are many factors to consider when choosing the right framework and it is also very individual to any specific project, team and environment. We decided to use Apache Flink for our project development, because it combines many required capabilities, so the examples are based on that particular framework, but the principles are common.

Figure 3 shows a high-level view of typical stream processing architecture including stream processing and message transport systems. The latter one decouples the different sources form the stream processing and buffers the events.

We use the popular message system Kafka (https://kafka.apache.org) with Schema Registry (https://docs.confluent.io/current/schema-registry/docs/index.html) in order to have transparent access to the data with less need of manual configuration.

Apache Flink is a layered system and provides high level of abstraction, Figure 4 (https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/general_arch.html).

The core of the stream processing is the “Flink runtime”. It is a distributed system that accepts streaming programs in the form of a JobGraph (a generic parallel data flow that consume and produce data streams) and can run in local Java virtual machine, standalone mode or clusters such as YARN (Yet Another Resource Negotiator) or Mesos.

Flink offers developer-friendly APIs that layer on top of the runtime and generate these streaming dataflow programs. There are libraries bundled with the generic DataSet and DataStream APIs like Table for queries on logical tables, FlinkML for Machine Learning, and Gelly for graph processing, etc.

Apache Flink provides many useful features such as exactly-once guarantees and data windows based on event time that make the developing process faster and with less possible bugs.

**Data exchange strategies**

Major problem in any distributed program is how to exchange and synchronize data between the computing nodes of the cluster.

The general programming model is master-worker where the master takes the input data end distribute it between the available workers for processing.

Execution resources in Flink are defined through Task Slots (https://ci.apache.org/projects/flink/flink-docs-master/internals/job_scheduling.html). The JobManager
receives the JobGraph and transforms it into an ExecutionGraph. TaskManagers execute the jobs consisting of multiple tasks, Figure 5.

The specific orchestration depends on the particular deployment (local, standalone cluster, YARN, Mesos etc.).

There are several ways to assign data to tasks in a physical dataflow graph. The importance of this will be explained with practical example of tracking users’ activities.

Let say that the program is supposed to track the online users’ activities and alert if someone logs in more than 5 times.

Data structure such as dictionary with user ID as a Key and Value equal to the number of logins can be used to keep the data. When login event with the same ID occurs, the Value is increased by 1. In case of new ID, the new entry is created and this is a very common pattern.

The problem is that in distributed environment the data structure (dictionary in our case) is local (Figure 6) and this leads to incorrect results. The common scheduler distributes each event for processing on different tasks, so the tracked 5 Login events for the user ID = 1 total, will not cause an alert.

One possible way to overcome this problem is to "broadcast" every data item to all parallel tasks of an operator, but this replicates the data and involves a lot of network communication, which is very expensive.

Better approach is usage of key attribute and keyed streams in order to guarantee that data items having the same key will be processed by the same task.

This also simplifies the development process and allows the programmer to use primitive data type like integer instead of dictionary, because each key belongs to exactly one parallel instance of a keyed operator, Figure 7.

**Persistent strategies**

In traditional stream processing, there is no global state as each event is processed individually. Streaming architecture connects variety of different producers and consumers and there are many options to persist the data.

One option is to use a centralize repository like database that any parallel process can access. The main disadvantage of this approach is that the access to such external centralize repository is slow and it becomes a bottleneck of the parallel performance.

Flink provides bundled connectors to different systems. Some of the predefined data sources or/and sinks includes Kafka, Cassandra, Elasticsearch, RabbitMQ, NiFi, etc.

For some systems such as Cassandra, Elasticsearch and Hadoop FileSystem only data sinks are officially provided, that is, data sources need to be implemented if needed. For such reasons, Flink offers an API for Asynchronous I/O allowing this kind of enrichment efficiently and easy.

Let continue with the example of tracking users’ activities. The number of login of each users can be stored in NoSQL database, for example Redis, which is an open-source in-memory database project implementing a distributed, in-memory key-value store.

We tried this solution and found improvement in the performance, but sending data out of the compute node still slow it down.

Recent trend in stream data processing is to keep data closer to where it is processed (Celebi, 2017), because the cross system interaction is usually the biggest bottleneck. Advanced stream processing technologies allow building applications directly on streams. This approach reduces the usage of external database and improves the parallel performance.

Flink has state as first class citizen. There might be limitations by its size, but for many applications is quite enough.
The Queryable State interface allows the state to be queried on demand. Using this state support improves the performance and simplifies the architecture.

**Complex workflows**

Stream data processing examples usually includes data source, transformations and data sink, Figure 8, but real life processes are usually more complex, Figure 9.

Programs can combine multiple transformations into sophisticated dataflow topologies. For example, connect operator “connect” two data streams allowing shared state between them.

One of the connected streams contains the rules and the others the elements that have to be processed against the rules.

The rules have to be updated, because they change over time and those updates are coming as a stream too. There is an initial set of rules that must be updated, before start of the processing of the events.

The split operator “split” the stream into two or more streams according to some criterion.

In complex workflows, different parts of the same message might be required for different microservices, which are incorporated in the different stages in the process, Figure 9.

Fortunately, the same stream can be consumed many times. This allows to connect incoming stream with the ones from the later stages of the workflow. Other approach is to keep all incoming messages that might be required for later processing in external database and retrieved when needed.

**Granularity**

When the workflow become too complex, let say dozens of stages and microservices, it is much harder for debugging and one should consider some sort of decomposition.

One job can process several Kafka topic or can process only one. For example, login events, click events, etc.

The code also can be decomposed into several subproject. This effect the structure of the project, having one parent Maven project and several child modules.

The price is some redundancy in the code, but the modules are much easier for deployment and maintenance.

**EXPERIMENTS**

Our development environment includes Flink cluster running on YARN2. The input streams are generated by program that reads data form static dataset and send it to Kafka in order to simulate real time data stream.

One of the main obstacles for the companies to enter into the stream data technologies is the lack of DevOps, who can setup and support such systems.

Cloud technologies provide Platform as a Service (PaaS) and make the setup process much faster and reliable.

Google Cloud Community provides scripts to specify initialization actions for different frameworks including Flink (https://github.com/GoogleCloudPlatform/dataproc-initialization-actions/tree/master/flink) on Google Cloud Dataproc. This makes the setup process much easier, but the deployment of the code and debugging is harder.

Flink has web interface providing many useful features like tasks status and back pressure monitoring, graphical
CONCLUSION

Driven by the huge demand, stream data technologies are developing very rapidly and differ from traditional big data. Many businesses are trying to transform their information systems in order to take advantage of stream processing, but using obsolete paradigms and models, which affects in facing bottlenecks that often compromise the desired results.

Unlocking the true potential of cutting-edge stream data technologies for creating scalable, reliable and intelligent solutions is an important competitive advantage.

REFERENCES

Chintapalli S, Dagit D, Evans B (2016). “Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming,” in International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA.
https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html.

Figure 10: Consuming one stream several times.