Suitability of stream processing engines for an exemplary student project

Jörg Einfeldt (je051@hdm-stuttgart.de), Jonas Häfele (jh176@hdm-stuttgart.de)

Abstract—Processing huge amounts of data is a common requirement in the day-to-day business of ultra large scale systems. Often referred to as Big Data, this field concerns itself with the continuous enhancement of the evaluation process of data in order to handle even more information. The latest development is the increasing demand not only to be able to process large amounts of input, but also to complete this task as fast as possible, which is summarised by the term Fast Data. Stream engines provide low end-to-end latencies and high throughput in comparison to traditional approaches by eliminating the initial storage of data and in-memory processing of data streams. This paper illustrates the general concepts of stream engines and provides more detailed information with reference to the Apache Flink framework. An experiment is presented which compares the Java 8 Stream API with Apache Flink in terms of efficiency. For this a word count function for both has been implemented and was executed on a gradually increased input size. Flink achieved a lower latency compared to Java 8 after an initial delay. The measured results stress the potential of stream engines like Apache Flink in terms of enhancing the performance, not only for massively scaled systems, but also for student projects.

Index Terms—Computer Science and Media, Ultra large scale systems, stream data processing

1 Introduction

Data processing is an intensely debated field, which has become more complex due to an almost infinite data amount provided by log records, sensor events and user requests [4]. Collecting, aggregating and evaluating these datasets as qickly as possible plays an increasingly important role in the business of ultra large scale systems, as this enables service providers to gain key insights and react to events near real-time. Historically, the most common processing model in the field of Big Data has been *MapReduce* [2]. This approach evaluates datasets by specifying a *map* function, which generates intermediate key/value pairs based on the input. These pairs are then merged dependent on a given *reduce* function. There can be multiple functions which are executed sequentially, this results in a pipeline for the data. The general principle of MapReduce is illustrated in Figure 1.
The functional style enables implementations to be run parallelized automatically and therefore they can be executed on large clusters. This distributed and parallel operating style makes efficient processing of huge datasets possible.

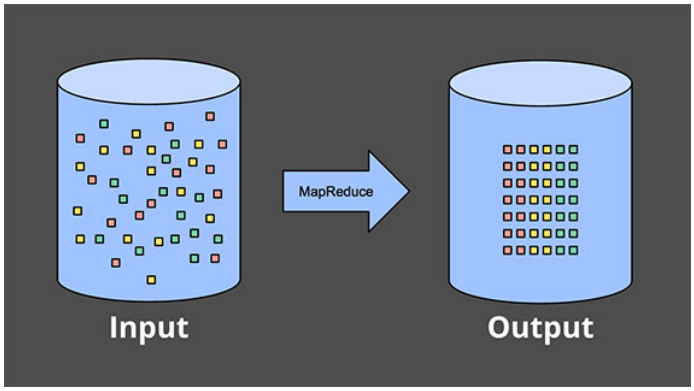

Fig. 1. Data processing with MapReduce [7]

MapReduce has a process-after-store approach, meaning that data is firstly stored in a database or filesystem and later structured, transformed and evaluated. This is generally how *batch* processing engines generate output and even though they are able to process huge amounts of data, the results are only available after a certain delay [7]. Their form of processing is not applicable to streams of data, which need analysis *on the fly* to enable real-time decision taking, as they rely on the stored information

[6]. Big Data solutions are not sufficient for these requirements anymore and there has been a rising demand for *Fast Data*. In order to be able to process near infinite streams of data, a new paradigm called *stream processing* has been proposed. This paper examines whether stream processing is an exclusive topic for massively scaled systems or if applications in the context of university projects can benefit from stream engines in terms of efficiency. In section 2 the general concept of stream engines is described followed by an overview of current frameworks and their capabilities in 3. The functionality of the Apache Flink framework is shown in detail in section 4. An experiment opposing the Java 8 Stream API with the Flink framework in terms of end-to-end latency is presented in 6 followed by a conclusion.

2 Stream processing

Stream processing engines have been developed to cope with the challenges of sheer endless streams of data, which need to be evaluated as soon as they arrive in order to be highly reactive and adaptive. The goal of stream processing engines is to execute SQL-like queries over datasets, which are continuously pushed to the system. These datasets consist of a sequence of tuples, which are generated in real time. The engine receives input tuples and returns output tuples based on its execution semantics [12].
In practice the architecture of stream engines often consists of multiple processing nodes that each dataset has to pass. These nodes perform different operations on the data as it flows trough. They execute their assigned actions concurrently and the scheduling and movement of the data is handled by a streaming platform. Stream processing engines achieve a low latency compared to batch processing engines due to the in-memory processing of the data without storing it. Most delays in traditional approaches arise typically from load processes and physical storage updates, which are reduced in stream processing [13]. But due to the omitted initial storing of the data, stream engines need mechanisms in order to guarantee a consistent persistence layer and no information loss in case of machine failure.

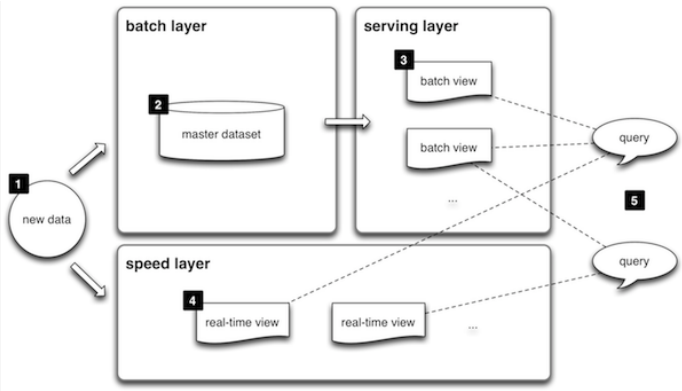

Fig. 2. Concept of the Lambda Architecture [11]

This trade-off between low latency and reliability often results in systems that combine batch and stream processing, which is called *Lambda Architecture*. This architecture consists of a data processing layer that focuses on low latency and another layer that enables strong consistency and reliability. Figure 2 displays the conceptual approach. The speed layer is used for fast computations and evaluating recent data while the batch layer is responsible for maintaining a master data set and processing archived data. Incoming queries are then answered by merging the results of both layers [11].

3 Frameworks

In [5] a set of widely used open source stream processing frameworks is described and evaluated in the context of high throughput and low-latency. Namely these are: Apache Storm, Apache Storm Trident, Apache Spark Streaming, Google Cloud Dataflow and Apache Flink. The above mentioned frameworks are using different mechanisms for fault tolerance, which have a strong impact on the architecture of a framework and the supported programming model. The goal of an efficient streaming architecture is to enable fault-tolerant as well as performant stream processing. A stream is a continuous flow of data wich has no particular starting or endpoint. In contrast to batch processing where a file can be processed again from start to finish it is not possible with

streaming to re-run jobs after failure. As a stream can run for a long period of time, e.g. for several months, buffering records and replaying them in case of failure is not practical. With this approach a large storage would be needed and the benefit of real-time computation would be lost. Streaming is a stateful computation. On failure the system not only has to restore the output of the computation but also the operator state.
A framework for the experiment in this paper has been chosen based on the results in [5].
The decision criteria for a streaming framework derive from the requirements of streaming applications:

- Recovery after failure without duplicates and correct restoration of the operator state.
- Processing of large amount of data in real time.
- Smooth integration into the system with low overhead and a controllable flow of data.

Based on these criteria the Apache Flink framework has been chosen. It fits the requirements described in this paper best by providing a strong computational model, the seperation of application and fault tolerance, low overhead, natural flow control, high throughput with very low end-to-end latency and exactly-once guarantees [5].

4 Apache Flink

Apache Flink is an open source plattform for distributed stream and batch data processing. The streaming dataflow engine forms the core of Flink's platform by providing distributed communication and data processing over streams. Usage of the core functionalities are simplified by different APIs or field specific libraries [8].
The framework provides exactly once guarantees through distributed snapshots which are described in [5] and shortly summarised hereafter. The continuous computation of data is provided by the *Flink Streaming API*. *Data Streams* are the basic abstraction offered by the API, which can be created by defining an external data source or by transforming other data streams. Streaming jobs are compiled into connected graphs of tasks. Data elements are called from external sources and pushed through the task graph in the form of pipelines. Tasks are continuously manipulated by the input and generating new output. Constant processing of data demands a guarantee regarding the correct state of the computation and the restoration after failure. Flink provides exactly-once guarantees by using distributed snapshots, which save the current state of a system into a lasting storage. The idea of snapshot algorithms was introduced in 1985 by Chandy and Lamport [1]. A benefit of the architecture is the seperation of application development and the controlling of flow and throughput ensuring always correct received data on application side.
As already stated earlier, a central part of Apache Flink's fault tolerance mechanism are distributed snapshots. Snapshots, also called consistent checkpoints, maintain the state of a consistent data stream and the states of it's operators. A snapshot is bounded by streaming barriers, which define the element set belonging to one snapshot. *Streaming Barriers* are inserted into the stream without altering it, as seen in Figure 3. The barriers are identified by

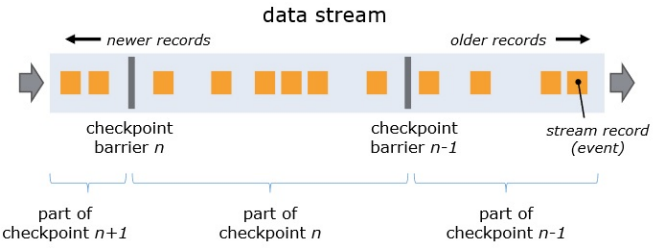

Fig. 3. Continous Checkpoints in Streams [10]

the ID of the snapshot which is located in front of it. In one stream barriers of different snapshots can be present and respectively snapshots are done concurrently. Barriers are injected at the source of the parallel data stream and the position of the checkpoint is reported to the coordinator (Flink's Jobmanager). At every operator position in the stream a new barrier is emitted to their output streams as soon as all barriers, belonging to one snapshot, are acknowledged. Receives a sink operator (endpoint of a stream) all barriers for one snaphot,

it signals the checkpoint coordinator that this snapshot has been correctly arrived and is no longer needed. Operators may have multiple input streams, which makes correct alignment of data and barriers necessary. As soon as an operator encounters a barrier of one incoming stream it stops the execution of the stream and buffers the records until the barriers of all other incoming streams have been arrived, to avoid mixture of data between streams. After all barriers are acknowledged the operator frees the dataflow of the first stream again and emits the barrier as well (Figure 4) [10].

Additionally the state of operators has to

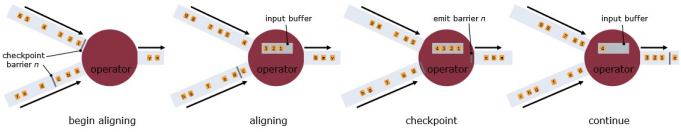

Fig. 4. Buffering of checkpoint barriers until barriers of other streams are received [10]

be added to the snapshot. For this purpose, operators take a snapshot of their state, when all snapshot barriers of it's incoming streams are received. After that the barriers are emitted to the output streams. The resulting snapshots consist of the start position in the stream for each parallel stream data source and a pointer to the state which was stored as part of the snapshot for each operator. Due to the buffering mechanism latencies of a few milliseconds arise. If an application does not allow such delays it is possible to give at least once guarantees by skipping the alignment of the barriers and processing all incoming records. In case of recovey of the system the records of both snapshots are replayed resulting in duplicated records. To regain the correct state of the system after failure, the latest full snapshot is loaded, the records are replayed and the corresponding states are assigned to the operators [10].

5 Use Case

The use case for this paper is an exemplary student project which processes a given text and maps a word to its frequency in the document. This example is often referred to as *word count* and has been chosen as it is a common introductory piece of code in the field of Big Data. Since Flink runs on the Java Virtual Machine (JVM) the reference implementation has been written in Java 8. It makes use of the *Stream* library which has been introduced to the language in order to give it a more functional approach. The word count function for Apache Flink has been taken from the frameworks official quickstart guide [9]. The Java 8 implementation can be found in the official repository [14] of [3]. Both implementations have been slightly adjusted for the requirements of the experiment, for example the regular expression, that seperates words in the text, has been matched.

As an input text for both functions the play *Hamlet* [15] is used, which roughly consists of 30.000 words. The amount of text is gradually increased by duplicating the text up until ten times of the original size. In every iteration the text length is doubled and afterwards the word count functions are executed. The required time for the word count to finish and the number of words that have been processed are written to a text file. The experiment measures the end-to-end latency in order to show the efficiency of the different implementations. Both programs have been tested on an *Amazon Web Service* (AWS) machine of type Mikro.T2. The machine is equipped with one virtual CPU and one gibibyte memory. The experiment has been run five times and in this paper the average values are presented.

6 Measurement

The results of the measurement are shown in Figure 5. The diagram shows that Apache Flink provides faster processing of data in general, once it is up and running. The data sizes scale from 194 KB with 30.000 words up to 1,92 MB with 15.360.000 words. Noticeable are the long delay of Apache Flink in the first iteration, which leads to Flink actually producing the highest measured latency and the interruption of the Java 8 implementation after the eighth iteration. The initial time delay of Apache Flink can be explained with all of its dependencies beeing loaded into the JVM at startup. After

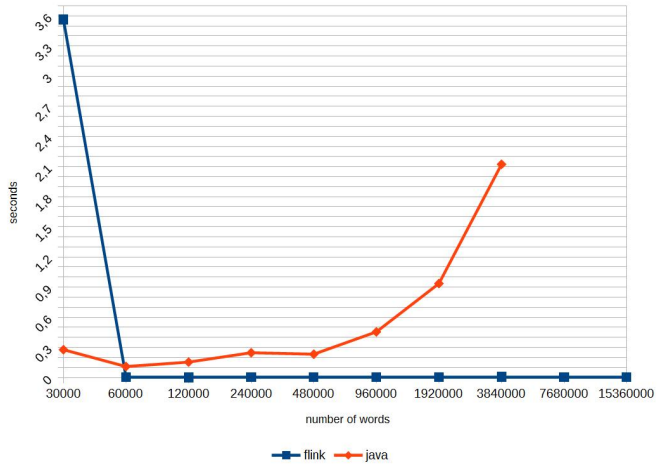

Fig. 5. Measurement results of the experiment

the initialization has been completed, Flink produces stable values and has a low end-to-end latency. With the standard heap size for the JVM, the execution of the Java 8 implementation stops with an OutOfMemoryException after doubling the input size five times. After adjusting the heap size and rerunning the experiment, eight iterations were possible before receiving a message from the virtual machine indicating insufficient available memory for the Java Runtime Environment (JRE). Increasing the heap size even further was not possible due to limitations regarding the test machine. Apache Flink runs in a JVM and uses memory optimizations enabling faster processing of the provided data compared to stream processing in Java. Therefore it was possible to complete all ten iterations without a forced interruption from the operating system.

7 Conclusion

The metrics in [5] and the results of the conducted experiment show the potential of Apache Flink in the context of data processing. The simple word count example demonstrates that Flinks capabilities in terms of efficiency are superior to the traditional approach. Other stream engines described in this paper are dependent on a distributed system architecture to unfold their full potential, but even on a single machine the usage of the Flink framework showed a significant advantage. If executed on a large cluster, Flink has features to improve the performance of data evaluation to a maximum. Its high abstraction level and strong efficiency are arguments that speak in its favor, not only in reference to ultra large scale systems, but also in the context of smaller environments. Furthermore in comparison to other stream engines its accessible functionality and mature architecture make Flink a noteworthy option for student projects that need to evaluate continuous datasets.
Even on infrastructures with limited ressources a better end-to-end latency can be achieved, as the experiment demonstrated. Another interesting field is the use and evaluation of Apache Flink's context specific libraries such as for machine learning and displaying graphs. Stream engines are going to be more advanced and designed in the future and based on the experiment it can be concluded, that not only are they an important development in the field of data processing in general, but they can also bring real benefits to a multitude of applications.

References

[1] K. Mani Chandy and Leslie Lamport. *chandy.pdf*. Jan. 1985. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/chandy.pdf (visited on 12/30/2015).

[2] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. 2004. URL: http://research.google.com/archive/mapreduce.html (visited on 02/25/2016).

[3] Richard Warburton. *Java 8 Lambdas Functional Programming for the Masses*. 1st ed. Sebastopol, California: OŔeilly Media, 2014.

[4] Tyler Akida, Robert Bradshaw, and Craig Chambers. *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing*. 2015. URL: http://research.google.com/pubs/pub43864.html (visited on 02/25/2016).

[5] Kostas Tzoumas, Stephan Ewen, and Robert Metzger. *High-throughput, low-latency, and exactly-once stream processing with Apache Flink — data Artisans*. May 2015. URL: http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/ (visited on 12/30/2015).

[6] Fatos Xhafa, Victor Naranjo, and Santi Caballe. *Processing and Analytics of Big Data Streams with YahooS4*. 2015.

[7] Tyler Akidau. *The world beyond batch: Streaming 101*. URL: http://radar.oreilly.com/2015/08/the-world-beyond-batch-streaming-101.html (visited on 02/25/2016).

[8] *Apache Flink 0.10.0 Documentation: Overview*. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.10/ (visited on 12/30/2015).

[9] *Apache Flink 0.10.2 Documentation: Quickstart: Setup*. URL: https://ci.apache.org/projects/flink/flink-docs-release-0.10/quickstart/setup_quickstart.html (visited on 02/20/2016).

[10] *Apache Flink 1.0-SNAPSHOT Documentation: Data Streaming Fault Tolerance*. URL: https://ci.apache.org/projects/flink/flink-docs-master/internals/stream_checkpointing.html (visited on 02/24/2016).

[11] *Applying the Big Data Lambda Architecture*. URL: http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604 (visited on 02/25/2016).

[12] Jeong-Hyon Hwang et al. *High-Availability Algorithms for Distributed Stream Processing*. URL: http://cs.brown.edu/research/aurora/hwang.icde05.ha.pdf (visited on 02/20/2016).

[13] *Stream Processing Explained*. URL: http://www.sqlstream.com/stream-processing/ (visited on 02/25/2016).

[14] *Support material for the book Java 8 Lambdas*. URL: https://github.com/RichardWarburton/java-8-lambdas-exercises/blob/master/src/main/java/com/insightfullogic/java8/answers/chapter5/WordCount.java (visited on 02/25/2016).

[15] *The Tragedy of Hamlet, Prince of Denmark*. URL: http://shakespeare.mit.edu/hamlet/full.html (visited on 02/25/2016).