Master Thesis

Processing home gateway traces with Spark

Student :
Junyang Shi
University Supervisor :
Prof. Guillaume Pierre

Company Supervisors :
M. Christoph Neumann
M. Pascal Le Guyadec
Technicolor R&D France

June 2016

Abstract

Apache Spark Streaming framework is now in widespread use for processing continuous events data in real time. Performance tuning is required to get the best performance out of a Spark Streaming application on a cluster. In this thesis, we design and implement performance benchmarking experiments for processing home gateway traces on the Apache Spark Streaming framework in an experimental environment. We consider two main benchmark areas, which are application workload characteristics comparison and application performance tuning. In terms of workload characteristics comparison, we implement the same Spark Streaming analytics tasks corresponding to different home gateway traces versions combined with different serialization methods and different communication protocols. With respect to performance tuning, we increase levels of parallelism to reduce data processing latency as well as set the right batch interval to process each batch as fast as it is received. We further evaluate the cluster throughput and data processing latency among benchmark experiments with a series of benchmarks. We also illustrate the limitations of the current workload and predict the future work in a real deployment environment.

Keywords: Spark Streaming, time series dataset, benchmark, performance tuning, throughput, latency, workload characteristics, serialization method, communication protocol, parallelism, batch interval

Acknowledgement

The work presented in this thesis was performed at Technicolor R&D France during 11.1.2016 – 8.7.2016. Foremost, I would like to express my sincere gratitude to my two instructors, M. Christoph Neumann and M. Pascal Le Guyadec, for giving me this great opportunity to working on this fantastic project. I especially wish to thank their patience and motivation at all the stages of the thesis. In addition, I want to thank all my colleague for their encouragement during my work.

I also want to thank Prof. Guillaume Pierre, my supervisor at Université de Rennes 1, for his constructive feedback, especially during the thesis writing process.

Finally, I want to thank my parents and friends for their kind support during my work.

Rennes, July 5, 2016

SHI, Junyang

Contents

1 Introduction

1.1 Problem Statement and Objectives

Technicolor is receiving a continuous stream of usage data and statistics from a large set of home gateways. This longitudinal collection of data with more than two years of data history constitutes a large and unique dataset of time series. The platform that handles and analyses this data is currently being redesigned and implemented. The new platform relies on Apache Spark, Apache Kafka and Apache Cassandra. It is designed to handle massive quantities of data by taking advantage of both batch and stream processing.

The performance of this new platform in terms of stream processing needs to be explored. As mentioned in [1], the throughput and latency of Spark Streaming is related to various factors, including batch sizes, data ingestion rates, variations in available resources and workload characteristics, etc. Currently, one of the main exploring area is to compare and evaluate the performance of our data processing platform with different workload versions of our time series dataset.

The records in the raw dataset that Technicolor has collected from gateways currently often requires to be processed the same order as they are collected in the database to calculate the desired output. Since the order of records that arrive at Spark cluster is not guaranteed, we need to drop some messages in the Spark Streaming job. In order to efficiently take advantage of Spark Streaming, we need to investigate some possible operations that need to be performed on the home gateways before it sends the dataset to our data processing platform. In addition, since the dataset need to be sent over the network, technological choices include serialization methods and communication protocols, which also need to be investigated for impact on the performance of our data processing platform. As a result, we need to take all these factors into consideration and implement different workload versions for our time series dataset.

Besides the performance benchmark for the dataset workload characteristics, we also look into the performance tuning of Spark Streaming in order to get the best performance out of a Spark Streaming application.

In the thesis, we aim to benchmark the above mentioned areas of our data processing platform. The following four objectives are set for this thesis:

- Implement Spark jobs corresponding to different home gateway dataset versions and benchmark their performance.
- Investigate and benchmark several technological choices in the software

stack including serialization methods and communication protocols.

- Tune levels of parallelism in Spark Streaming data processing to handle high throughput and efficiently using cluster resources.
- Find out the right batch interval for Spark Streaming to process data as fast as it is being received for each parallelism configuration.

1.2 Contributions of This Thesis

In the thesis, we have outlined our data processing platform and present corresponding technical background of real time processing components with focus on Spark Streaming and Kafka. In terms of benchmark experiments, we firstly illustrate the major technical choices that involved in our benchmark experiments, including three client workload versions (counter, cumulative counter and rate), two serialization methods (JSON and Avro) and two communication protocols (HTTP and WebSocket). Then we describe and explaine the design and deployment of our benchmark experiments. Furthermore, we have implemented Spark jobs corresponding to all the combinations of the technical choices mentioned above and compare their performance in terms of throughput and latency. Finally, we tune performance of our Spark Streaming applications in two aspects in order to achieve stable processing with low data processing latency, which are increasing levels of parallelism and setting the right batch interval. Specifically, we compare and evaluate the performance of different levels of parallelism in Kafka and Spark Streaming. We focus on the throughput in the Kafka cluster and the latency in the Spark Streaming cluster. We also evaluate the batch interval to stabilize our applications for each level of parallelism.

1.3 Related Work

Today, we live in the age of Big Data in which data and information technology play important roles in dictating the quality of our lives. We are witnessing explosive growth in the variety, velocity, and volume of data generated by a wide variety of data sources, including Internet search, social media, mobile devices, the Internet of Things, business transactions, scientific computing, etc. Big Data is opening up new opportunities for enterprises to extract insight from huge volumes of data in real time [2]. As a result, both academia and industry have dedicated significant effort towards deploying and operating Big Data systems to enable enterprise have a better handle on data processing [3].

Most of what a enterprise does can be thought of as streams of events [4], for example, streams of orders, sales and shipments in a retail company, streams of stock prices, trades and other financial time series in a financial company, streams of requests, errors, machine metrics, and logs in a software company. Under the development of Big Data in these days, companies start to capture these events data and put them to use for analysis, optimization, and decision making [4]. One popular successful example is Google, which transformed the stream of ad clicks and ad impressions into a multi-billion dollar business.

In terms of handling event data, batch data processing and real-time stream data processing are two typical approaches. Systems like Hadoop are used to handle batch event storage and processing [5] while stream processing frameworks such as Spark Streaming, Apache Storm and Apache Flink aim to process continuous events data near real time [6]. Compared to batch data processing which is usually performed offline, real time stream data processing allows an organization the ability to take immediate action for those times when acting within seconds or minutes is significant [7]. One of the big challenges of stream processing is that one needs to get fast access to historical data on the fly for predictive modeling with real time data from the stream, resulting in demand for batch and stream processing integration [8].

Spark is one of the few data processing frameworks that allows seamless integration of batch and stream processing [9]. In the last year Spark summit, Edelson [8] introduces the lambda architecture with Spark Streaming, Kafka, Cassandra, Akka and Scala, which is designed to handle massive quantities of data by taking advantage of both batch and stream processing. Batey [10] presents a reference application killrweather [11] from Databricks, showing how to easily integrate streaming and batch data processing with Spark Streaming, Cassandra, Kafka and Akka for fast, streaming computations on time series data in asynchronous Akka event-driven environments. The time series data is collected and analyzed in sequence at extremely high velocity to get the clearest picture to predict and forecast future weather changes [11]. Our data processing platform is similar to these architectures, which relies on Kafka, Spark Streaming and Cassandra for both batch and stream processing on time series data collected from Technicolor's gateways.

Performance tuning is required to get the best performance out of a Spark Streaming application on a cluster [12]. Batch sizes, data ingestion rates, variations in available resources and workload characteristics etc can all have impact on the cluster throughput and end-to-end latency. A number of interesting works have been done to tune parameters and configurations of Spark Streaming application in order to improve the performance. Tathagata *et al.* [1] explore the effects of the batch size on the performance of streaming workloads. Brosinski [13] focuses on optimizing Spark Streaming application with three steps, which are stabilizing the app by tuning batch interval, streaming rate parameters and then scaling the app by tuning parallelism, shuffle and memory parameters and finally iterate the above two steps. Wilgenburg [14] states that with the the new back pressure feature introduced in Spark 1.5, the optimal rate for a Spark job is now done automatically which releases the need to set a maximum data ingestion rates. In terms of benchmarking the performance of our data processing platform, we will focus on comparing and evaluating the performance of various workload characteristics, batch sizes and parallelism levels.

In addition to tuning the performance of Spark Streaming applications, much research has been devoted to improving the performance of the Spark framework from three widely-accepted mantras, namely, network optimization, disk access optimization and straggler task optimization. Zhang *et al.* [15] propose an optimization framework that reasons about user-defined functions and data shuffling, leveraging application semantics to prioritize network traffic. Rasmussen *et al.* [16] present a MapReduce implementation that minimizes the number of disk I/O operations. Ousterhout *et al.* [17] address task launch overheads and scalability limitations by proposing a task execution framework that can efficiently support tiny tasks. Ousterhout *et al.* [18] develop a methodology for quantifying performance bottlenecks in distributed computation frameworks and apply it to analyze the Spark framework's performance on SQL benchmarks. In conclusion, we are likely to perform further benchmarks in the future to identify performance bottlenecks of the Spark framework with our production workload and to optimize the usage of network, disk and cpu resources.

1.4 Outline

The rest of the thesis is organized as follows. Section 2 introduces the background about Spark, Spark Streaming, Kafka, Serialization Methods and Communication Protocols. Section 3 discusses the components in our real time processing pipeline, presents the client workload as well as illustrates the design and implementation of the benchmark experiments. Section 4 provides performance evaluation of benchmark experiments. In the end, Section 5 concludes the thesis.

2 Background

This section illustrates the main background related with benchmark experiments in several subsections, including Spark & Spark Streaming, Kafka, Serialization Methods and Communication Protocols. Each subsection contains a list of key concepts as described below.

2.1 Spark & Spark Streaming

2.1.1 Spark Overview

Apache Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics [19]. It was originally developed at UC Berkeley in 2009. Spark offers unified API across workloads, storage systems and environments as shown in figure 2.1. In Spark, the data structures and algorithms of specialized components are implemented, users are left to control data partitioning and caching. The unified feature is implemented based on the core concept of Spark – Resilient Distributed Datasets (RDDs). Different specialized components are easily integrated with RDDs. Furthermore, RDDs allows data to be quickly shared across execution steps, which reduces overall processing latency significantly. In addition, RDDs can be stored in memory or disk across the cluster, thus allowing Spark to achieve in-memory computing.

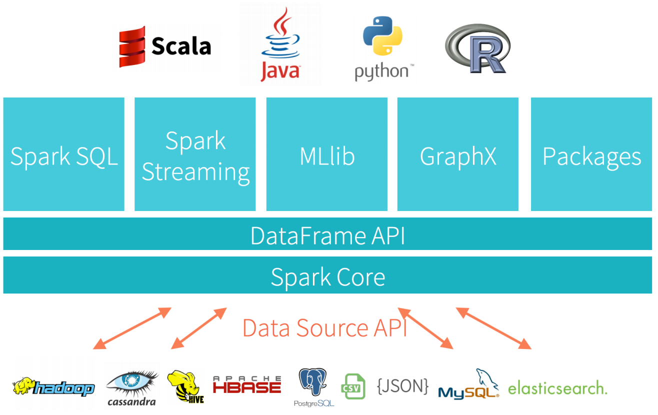


Figure 2.1: Spark Application [20]

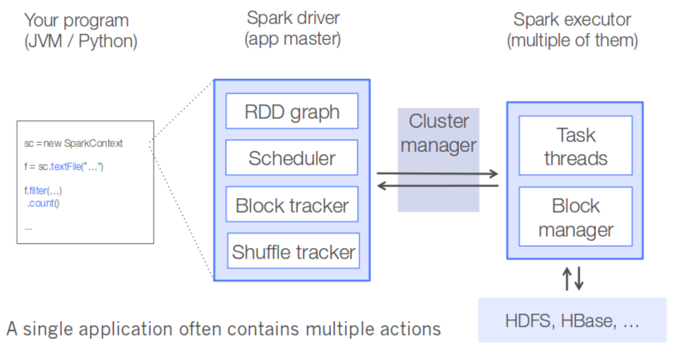
2.1.2 RDD

Resilient Distributed Datasets (RDDs) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel [21]. RDD is an interface, each RDD is defined by five properties: partitioning, dependencies, compute, (optional) partitioner and (optional) preferred locations. Specifically, partitioning refers to a method to divide the work into partitions, dependencies states a list of parent RDDs, compute represents a method to compute a partition given its dependencies, partitioner defines how keys are hashed and preferred locations presents a list of preferred hosts for a given partition. The size of partition depends on the data source used and the amount of partition can be specified in the application.

RDDs have two fundamental types of operations: transformations and actions. Transformations are lazy operations on an RDD that return RDD objects or collections of RDDs. Actions are operations that return values which evaluates the RDD graph. An RDD graph is a graph of what transformations need to be executed after an action has been called.

2.1.3 Spark Application

As shown in Figure 2.2, the application main program is called the driver program, which is a JVM process. The driver process owns an instance of SparkContext which is essentially a client of Spark's execution environment and which acts as the master of the Spark application. SparkContext accesses a number of components including RDD Graph, DAGScheduler, TaskScheduler, SchedulerBackend, ListenerBus and SparkEnv, etc. It closely works with executors and is responsible for submitting jobs, scheduling jobs and handling failed jobs, etc.



```
sc = new SparkContext
f = sc.textFile("…")
f.filter(…)
 .count()
...
```

Figure 2.2: Spark Application [22]

2.1.4 Cluster Mode Architecture

As shown in Figure 2.3, the cluster manager is responsible for allocating the cluster resources across applications, it can be either Spark's own standalone cluster manager, Mesos or Yarn. An executor is a JVM process launched for an application on a worker node and stores data in memory or disk storage. A task is a unit of work that will be sent to one executor and is executed as one or multiple threads in the executor JVM. In the Spark parallelism model one task is correspond to one RDD partition.

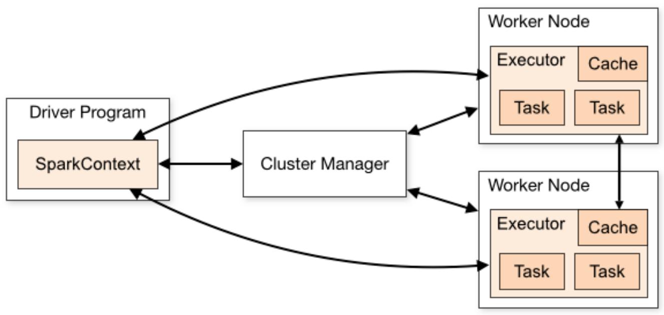


Figure 2.3: Cluster Mode Architecture [23]

Each driver process connects to the cluster manager to request resources and gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. Applications are isolated from each other on both the scheduling (driver) side and the executor side. When the application starts, the processing of the job is split up into stages, and each stage is split into tasks. Each task is scheduled separately. Each executor is configured with a certain amount of cores aka slots (each can be assigned with multiple physical cores) and each task occupies one slot in the parent executor.

2.1.5 Job Submission Process

As mentioned before, an RDD graph aka a DAG is built right after the call to an action on RDD. A single application often contains multiple actions thus generating multiple DAGs and DAGs are submitted as jobs in order inside the driver. Furthermore, the DAGScheduler is responsible for splitting a DAG into stages. It submits sets of tasks for each stage to the TaskScheduler. The TaskScheduler is in charge of sending the tasks to the executors to run and is also responsible for retrying if there are failures and mitigating stragglers. Internally, the SchedulerBackend receives the worker list from the cluster manager and launches tasks at the executor. The BlockManager at each executor will help it to deal with shuffled or cached RDDs.

2.1.6 Spark Streaming Overview

Spark Streaming is a component that runs on top of the Spark execution engine for doing large-scale stream processing. It offers various connectors for upstream systems like Kafka, Flume, Kinesis and downstream systems like HDFS, HBase, Cassandra as well as simple batch-like API for implementing complex algorithms for data processing.

Based on the unified programming model of Spark components, Spark Streaming provides some unique benefits over traditional streaming systems which includes fast failure and straggler recovery, better load balancing, combining of streaming data with static datasets and interactive queries, etc.

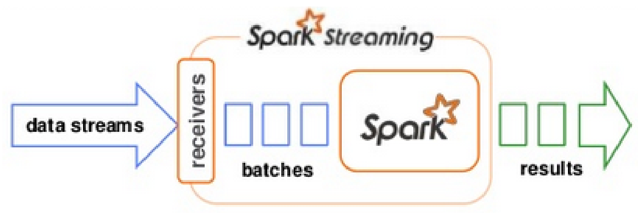


Figure 2.4: Spark Streaming [24]

As shown in Figure 2.4, Spark Streaming runs a streaming computation as a series of very small, deterministic batch jobs. Specifically, receivers receive data streams and chop them up into batches of seconds, Spark treats each batch of data as an RDD and processes them using RDD operations. In each batch interval, the data collected at the previous batch interval is provided to Spark and receivers start filling the next bucket of data.

2.1.7 Discretized Streams

Discretized Stream (DStream) is a sequence of RDDs representing a continuous stream of data [25]. Like RDD, transformations and output operations (actions) can be applied on a DStream. As shown in Figure 2.5, RDD graph is computed from DStream graph in every batch interval. For each output operation on an RDD, a Spark job is created, thus achieving a programming pattern quite similar to Spark. The method to compute an RDD defined on DStream along with functions applied to an RDD in output operation on DStream are executed in the driver process.

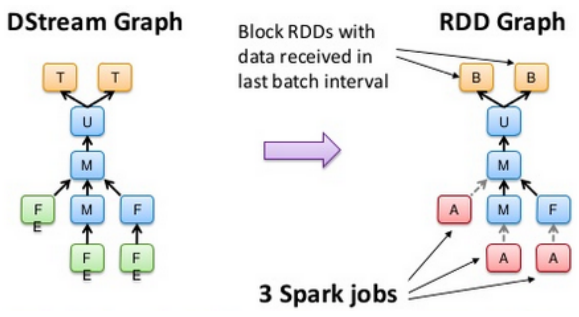


Figure 2.5: DStream Graph [26]

DStream interface defines how to generate an RDD for each batch interval. Similar to RDD, a DStream is defined by a list of dependent DStreams, a time interval at which it will generate batches, and a method to compute an RDD for a given time interval. In terms of input DStreams that receive data from the network, the dependency is none, the time interval uses the interval of the streaming context and the compute function creates a BlockRDD with all blocks of data received in the last batch interval. Each input DStream is associated with a Network Receiver object. The driver program runs a Receiver within an executor as a long-running task.

2.1.8 Data Receiving Parallelism

Each external data source needs one input DStream implementation with the number of input DStream specified. The number of consumer threads per input DStream can be configured to increase read parallelism. Multiple DStreams can be unioned together to create a single DStream on which a transformation is applied to. In each batch interval, receiver divides the data stream into blocks and delivers it to the BlockManager to store in the executor memory. Data blocks are also replicated to another executor and written to a Write Ahead Log (WAL) in a fault tolerant file system. The number of blocks in each batch can be specified by receiver, which determines the number of RDD partitions created by the input DStream.

2.1.9 Spark Streaming Execution

The life cycle of the data received from an external data source is shown in Figure 2.6, which mainly contains four steps including data receiving, driver notification, data processing and computation checkpoint. In terms of notifying the driver, the metadata of the received blocks are sent to the StreamingContext which includes the block location reference id and the offset information in the WAL. With respect to data processing, the StreamingContext uses block information to generate RDDs and SparkContext executes corresponding jobs to process the in-memory blocks in the executors. The streaming computation metadata is periodically checkpointed for failure recovery.

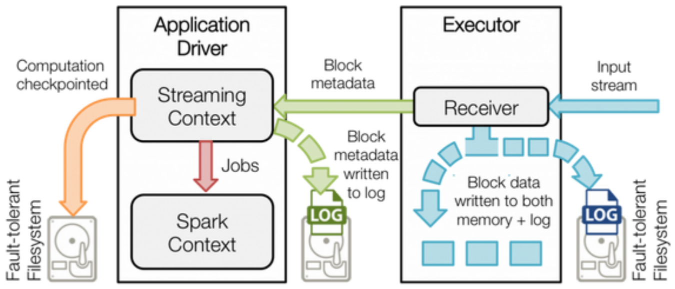


Figure 2.6: Spark Execution [27]

2.1.10 Direct Kafka Integration

The Direct API for Kafka is implemented in Spark 1.3 to achieve exactly-once message delivery semantics when consuming data from Kafka. Compared to earlier Kafka integration with receivers and WALs, offset information is only kept in Spark Streaming rather than simultaneously updated in Zookeeper and Kafka's simple consumer API instead of the high-level consumer API is used to fetch data. Specifically, as shown in Figure 2.7, the driver program decides offset ranges to read from Kafka at the beginning of each batch interval to generate an RDD and launches jobs for each batch using the offset ranges. Then the executor reads data from Kafka based on offset ranges to process. As a result, Kafka direct approach automatically parallelizes data reading across executors without using long-running receivers, thus offering uniform usage of Spark cluster resources. It also achieves a 1:1 correspondence between a Kafka partition and an RDD partition, thus simplifying the parallelism model. Additionally, the offset ranges generated at each batch interval need to be stored in case the driver fails, which can be done by a metadata checkpoint.

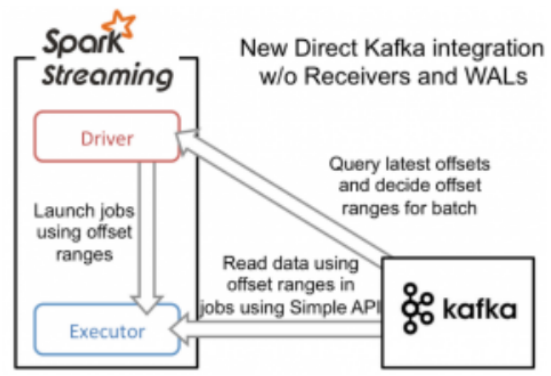

Spark Streaming
New Direct Kafka integration w/o Receivers and WALs
Driver
Launch jobs using offset ranges
Query latest offsets and decide offset ranges for batch
Read data using offset ranges in jobs using Simple API
Executor
kafka

Figure 2.7: Direct Kafka Integration [28]

2.2 Kafka

2.2.1 Stream Data Platform

Stream processing is a generalization of batch processing, which does not require a static snapshot of the data being processed and produces output at a user-controlled frequency. A stream data platform is a central hub for data streams and it enables continuous, real-time processing of data streams. On one hand, the platform offers uniform stream format for various sources, applications that consume the data streams are decoupled from those that publish the data streams. On the other hand, the platform allows applications to tap into the stream and create derived streams as well as supports a close integration with stream processing systems. Apache Kafka is a publish-subscribe stream data platform as shown in Figure 2.8. It allows geographic distribution of data streams and processing while ensuring durability, fault-tolerance, scalability and high-throughput properties [29].

2.2.2 Topic

Topic is the logical name given to a particular data stream, which is implemented as a WAL as shown in Figure 2.9. The log can be partitioned and spread over a cluster of servers, and each partition can be replicated for fault-tolerance. Each message/record in the data stream is a key/value pair and the particular format is flexible. The key is used to assign the record to a log partition. Each record in the partition has an associated entry number called the offset that uniquely identifies each record within the partition. The records are not deleted when they are read but retained for a configurable period of time.

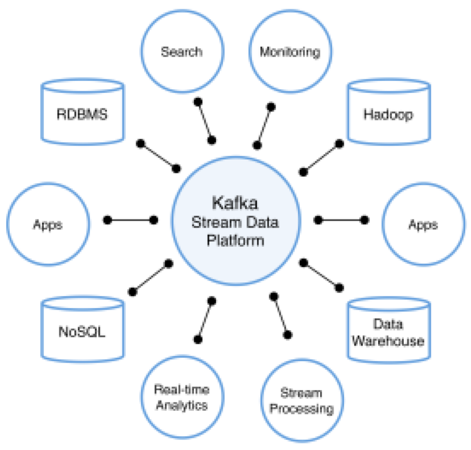


Figure 2.8: Kafka Stream Data Platform [30]

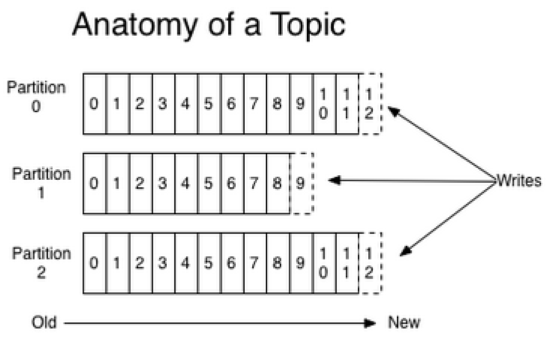


Figure 2.9: Topics and Logs [31]

2.2.3 Cluster Overview

As shown in figure 2.10, each server of a Kafka cluster is called a broker. The processes that write data to brokers are called producers and the processes that read data from brokers are called consumers. The number of partitions can be set per-topic and per-broker. Within a partition, records are stored in order in which they are written by producers and read by consumers. Each partition can be replicated across brokers among which one acts as the leader and the others act as followers. All writes and reads go to the leader of partition and the leader keeps track of the set of "in sync" brokers. The ID of a replica is the same as the ID of the broker that hosts it as shown in Figure 3. Brokers, producers and consumers use Zookeeper [32] to manage and share state. Specifically, Kafka uses Zookeeper for the following reasons. Firstly, Zookeeper elects the controller across brokers which maintains the leader/follower relationship for all the partitions [33]. Secondly, Zookeeper manages cluster membership to acknowledge which brokers are alive. Thirdly, Zookeeper stores topics configuration under brokers. In addition, Zookeeper is used for offset management. Before Kafka 0.8.2, Zookeeper was used to coordinate consumer groups and it store the next offset for a consumer/topic/partition combination. Kafka high-level consumer API connects to Zookeeper to track consumed offsets as well as to handle load balancing. Starting with version 0.8.2, the high-level consumer can use Kafka itself to manage offsets, which relieves the Zookeeper from being a bottleneck given high read write load on it [34].

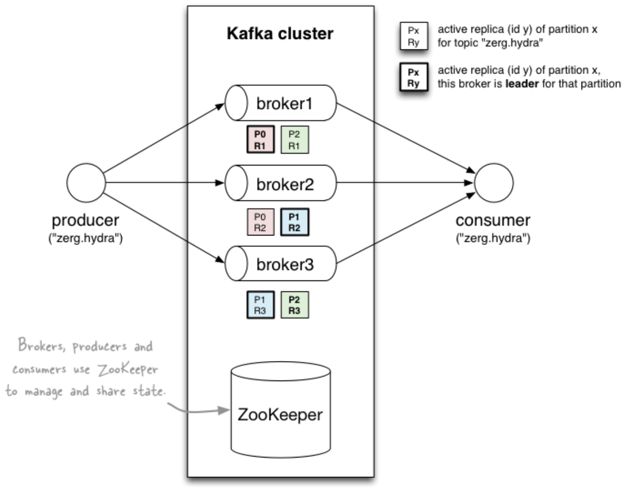

Kafka cluster
Px Ry active replica (id y) of partition x for topic "zerg.hydra"
Px Ry active replica (id y) of partition x, this broker is leader for that partition
broker1
P0 R1
P2 R1
broker2
P0 R2
P1 R2
broker3
P1 R3
P2 R3
producer ("zerg.hydra")
consumer ("zerg.hydra")
Brokers, producers and consumers use ZooKeeper to manage and share state.
ZooKeeper

Figure 2.10: Kafka Cluster Overview [35]

2.2.4 Producer

The producer is in charge of choosing the partition within the topic to write to for a given record. Records with the same key will be routed to the same partition and the partition function can be customized. There are two types of producers called the synchronous and the asynchronous producer. In asynchronous mode, producer manages a buffer of records in memory and sends records in batch in a background thread which guarantees no blocking of sending in producer client. In Kafka, a record is considered committed when the required number of in-sync replicas have written it to their log. The producer is in charge of configuring the number of required replicas and will get an acknowledgement from the leader once the record is committed. However, the consumer will only fetch messages that have been written to all active replicas. In addition, Kafka supports compression of a batch of records that will remain compressed in the log and will only be decompressed by the consumer.

2.2.5 Consumer

The consumer is responsible for tracking its sequential offset when fetching data from broker. Kafka offers a cluster-wide identifier for the consumer applications called a consumer group. Consumer threads belonging to the same consumer group share the burden of reading from a given Kafka topic, practically through assigning the partitions in the topic to the consumer threads in the consumer group. Multiple consumer groups can run in parallel against the same Kafka topic and each with different level of read parallelism. The number of consumer threads in a consumer group cannot exceed the number of partitions of a given topic, otherwise the extra consumers will be idle. Before Kafka 0.9, there are two levels of consumer APIs provided by Kafka for JVM users which are the high-level consumer API and the simple consumer API. The Kafka high-level consumer API stores offset in Zookeeper which manages the offsets per partition per topic for users. In contrast, the Kafka simple consumer API requires the user to specify offset ranges on a partition basis in a fetch request.

2.3 Serialization Methods

2.3.1 JSON

JavaScript Object Notation (JSON) is a text format for the serialisation of structured data [36]. It derives from JavaScript and it is a language-independent data format. Code to generate and parse JSON-format data is available in many programming languages [37]. JSON has number, strings, booleans and null as primitive data types. It also supports objects and arrays as structured data types. JSON is widely used to exchange data between web services and applications and to pack or unpack data to/from a text format. Encoding mechanism should be specified when sending JSON data over the network or storing it in a file.

2.3.2 Avro

Avro is a cross-language serialization library that requires the data structure to be formally defined by some sort of schema [38]. It provides APIs for a wide range of languages including Java, Python, Ruby, C and C++ etc. Avro schema is represented in JSON and the data itself is stored in a binary format. The serialization schema should always be stored together with the Avro data in the same file [39]. Avro supports a rich set of primitive data types including: numeric, binary data and strings; and a number of complex types including arrays, maps, enumerations and records [40]. One interesting feature of Avro is schema demand during both data serialization and data deserialization, which relieves Avro data to be tagged with type information, thus making the binary encoding of Avro data compact and efficient. Another key feature of Avro is a robust support for schema evolution including both backward compatibility and forward compatibility, which allows the data encoded with the old schema to be read with the new one and vice versa.

2.4 Communication Protocols

2.4.1 HTTP

Hypertext Transfer Protocol (HTTP) is an application layer protocol exchanging or transferring content, such as hypertext, in request/response pattern [41]. HTTP communication is usually established over TCP/IP connections with the default port TCP 80. HTTP has a limitation on handling connections. For every request of the client, a port/socket is opened and then closed after transferring the data. Generally, a TCP connection is expensive to create, and therefore the opening and closing of TCP connections creates large overhead. Another limitation with HTTP is its uni-directional communication mechanism. HTTP is based on a "pull" paradigm and the server cannot actively push data to the client. Several approaches are created to help the client asynchronously retrieve data from the server which are polling, long polling and HTTP streaming [36]. Figure 2.11 illustrates a sequence diagram where a client communicates with a server using HTTP.

2.4.2 WebSocket

The WebSocket Protocol enables full-duplex communication through a single socket between a client and a server [36]. It supports data frames including text and binary. Due to its singe socket design, WebSocket improves the efficiency on both the client and the server compared to the communication over HTTP. On the server side, it decreases the amount of opening ports, thus allowing fewer servers to handle requests. On the client side, it saves computing, storage and networking resources for polling and making requests. In addition, the socket that

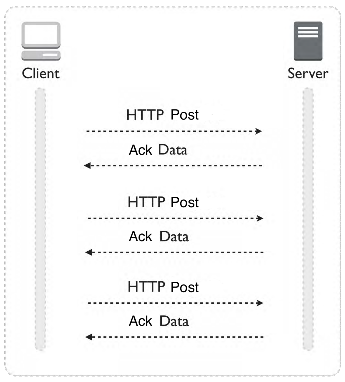


Figure 2.11: HTTP Post between a client and a server [36]

is connected between the client and the server stays “open” for the whole communication period, therefore the data can be “pushed” from the server to the client in real time on demand. As a result, WebSocket is especially suitable for certain applications that require real time interactions or display streams of data. Figure 2.12 illustrates connection details where a client communicates with a server using WebSocket.

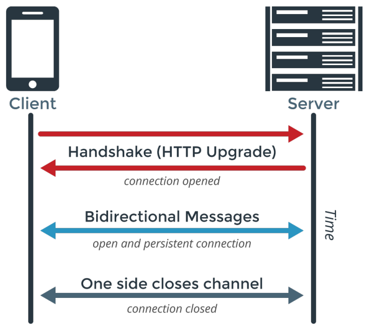


Figure 2.12: WebSocket Post between a client and a server [42]

3 Experiment Design

In this section, we will start with data processing platform components introduction. Then we will describe our client workload and corresponding analytics tasks in the experiment. Next, we will present the design of the benchmark experiments. Finally, we will illustrate the implementation of the benchmark experiments.

3.1 Platform Components

Our data processing platform consists of the following components as shown in figure 3.1: clients, the ingest server, Kafka cluster, Spark cluster and Cassandra cluster. They are constructed together to offer a coherent processing pipeline. The clients refer to the MongoDB which stores the continuous stream of usage data and statistics sent from home gateways. the ingest server is a HTTP/WebSocket server that accepts binary messages and forwards them to Kafka. It acts as the Kafka producer for Kafka cluster. Kafka cluster handles massive data load and integrates data between the ingest server and Spark Streaming. It is treated as a durable store of messages. Spark Streaming cluster reads data from Kafka, run Spark jobs and writes the result to Cassandra. Cassandra is a NoSQL database platform [43], which is widely used to manage dynamic data in real-time and is an excellent fit for handling time series data in sequence regardless of data type or size [11]. In this thesis, interaction between Spark and Cassandra is not covered.

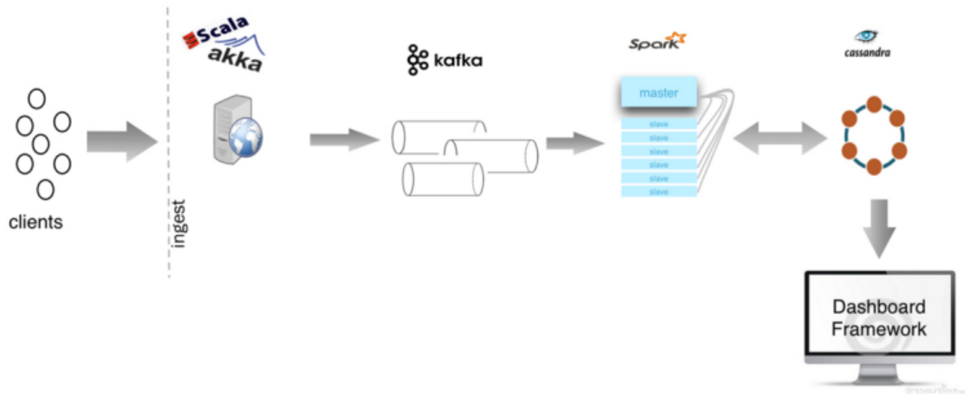


Figure 3.1: Real Time Processing Pipeline

3.2 Client Workload Description

Technicolor has collected home gateway usage data from around 200 home gateways during the last two years. In the MongoDB, we stored the time series dataset which is the real measurement traces generated from home gateway usage data. The time series dataset contains records with specific year, month, day, hour, minute and second timestamp. The records are not exactly collected by time slice, which means the time interval between two records is not stable. The most common interval between two records is around one minute. Each record has the following schema:

- gateway_id: String
- date: Long
- ipTx: Float
- ipRx: Float

The gateway_id field refers to the gateway that sends the record. The date field refers to the timestamp of the record. The ipTx field refers to the amount of sent traffic (represented by bytes or rate) and the ipRx field refers to the amount of received traffic. In the raw dataset, ipTx and ipRx fields are defined by cumulative value of sending and receiving traffic at each timestamp, which corresponds to the "cumulative counter" version in our benchmark experiments. In the MongoDB client, we have implemented other two client pre-processing versions (with different definitions of the ipTx and ipRx fields) on the raw dataset for workload characteristic benchmark called "counter" and "rate", which correspond to some operations that can be done on real gateways before they outputting records to the data processing platform. Each of the three versions corresponds to a Kafka topic in our benchmark experiments. The detailed definitions of ipTx and ipRx fields, in other words, the sent and received traffic in the three client pre-processing versions are listed below:

- Cumulative Counter: Defined by cumulative value of sent and received bytes at each timestamp for a gateway_id. The cumulative value cannot exceed a maximum threshold otherwise it will start from zero again.
- Counter: Defined by sent and received bytes from previous timestamp to current timestamp.
- Rate: Defined by sent and received rate (bytes/sec) from previous timestamp to current timestamp.

In terms of sending data over the network, we consider other two aspects that impact our client workload which are serialization methods and communication protocols. Regarding serialization methods, we have implemented JSON and Avro serialization for our dataset. With respect to communication protocols, we have chosen HTTP and WebSocket for our dataset. As a result, we finally have a bundle of client workload implementations which contain all different combinations of dataset versions, serialization methods and communication protocols. And on the stream data processing side, we have implemented Spark Streaming jobs corresponding to all the combinations we have on the client side.

3.3 Benchmark Design

The job of benchmark is to read massive JSON or Avro records from Kafka, identify the home gateway, calculate the whole traffic for each home gateway in each batch interval, identify the top ten gateways that have the highest cumulative traffic per day, and print the result to the terminal in each batch interval. These steps correspond to some operations performed on data streams.

The flow of operations is as follows:

1. Read a record from Kafka (with a given topic related to the client workload versions and a specific decoder related to the serialization methods used).
2. Deserialize the JSON string of each record into a key-value pair (Key is a two-element tuple of gateway_id and day_id, with day_id generated from date field in record schema. Value is a three-element tuple which includes ipTx, ipRx and date fields in record schema).
3. Group the records by key (with groupByKey or reduceByKey).
4. Calculate the cumulative sent and received traffic for the records with the same key.
5. Aggregates the result per day in each batch interval through updateStateByKey function.
6. Filter the top ten gateways with the highest cumulative sent and received traffic per day by key.
7. Print the result to the terminal in each batch interval.

The details of the Spark Streaming jobs will be discussed along with benchmark experiments in section 4.

Currently, the benchmark workload is relatively simple. We are likely to design and implement more complex benchmark workload in the future, for example, calculating the mean traffic for all gateways in each batch interval.

3.4 Benchmark Implementation

3.4.1 Cluster Deployment

We deployed our cluster with reproducible development environments using VirtualBox [44], Vagrant [45] and Ansible [46]. VirtualBox is used to create and run virtual machines. Vagrant is used to manage virtual machine environment, which allows us to configure and share reproducible configurations of the virtual machines on top of Virtualbox. Ansible is used to provision virtual machine, i.e. automatically install and configure software on virtual machine, which allows us to keep system-level configurations transparent and instantly repeatable from source control [47]. Vagrant and Ansible work together well. They both applied to virtual machines through reproducible SSH sessions. Vagrant lets us define virtual machines in a human-readable format, for instance to set up a development group [48]. Ansible integrates with Vagrant as a provisioner and it uses Ansible playbook to provision groups of hosts to a repeatable state.

Our data processing platform can be deployed both locally and on the cloud providers like AWS and the choice is based on a single configuration parameter. The performance benchmark experiments did in this thesis are based on the local deployment.

3.4.2 Experiment Environment

With respect to the deployment of Spark, our experiments are based on the Spark standalone cluster deploy mode. We use Spark Streaming application web UI to monitor the Spark Streaming metrics, runtime environment and cluster resource usage, etc in real time. We also use Spark History Server to view the UI of a finished application. HDFS is used as the persistent storage for the Spark cluster, which stores Spark events logs that encode the information displayed in the appliation web UI as well as provides checkpoint directory for the application. In addition, we take advantage of Spark master's web UI in the Spark standalone mode to manage the cluster. With respect to the Kafka cluster, we use an open source tool – Kafka-manager – [49] to manage the Kafka cluster.

For the benchmark workloads, we run experiments using a cluster of 5 homogeneously configured virtual machine instances, of which each have 4GB of memory, 18.4GB of disk and two Intel E5-2640 processors running at 2.5GHz, with a total of 2 cores (1 physical, 2 hyper-threading). Our experiments use Apache Spark version 1.6.1 and Apache Kafka version 0.8.2.

Benchmark setup

- 4 Spark worker nodes (version 1.6.1, each with 2GB memory)
- 3 Kafka nodes (version 0.8.2, no replica)
- 1 Kafka producer node (the ingest server)
- 1 Zookeeper node

Currently, there is no bottleneck in our data processing pipeline given the highest MongoDB client sending rate. In terms of the ingest server, we probably need more Kafka producer in production environment for handling large number of gateways in real time. In terms of Kafka cluster, a three node Kafka cluster isn't very big, but since we will only be testing up to 3 partitions without replica, it is all we need. And for the case when the number of partitions exceed the number of brokers or when a number of replicas are added to ensure fault-tolerance, they are likely to be explored in the future. In terms of Spark cluster, the use of 4 workers for a topology as well as the amount of memory configured are more than enough in our benchmark experiments since we will only be testing up to 3 parallel tasks in Spark, which will be explained in the evaluation subsection. Other metrics or configurations that are involved in the benchmark experiments include Spark Streaming batch interval and Kafka partition number are not listed in the setup. The related details will be discussed in the evaluation subsection.

4 Evaluation

In this section, we will evaluate our benchmark experiments results after a brief introduction of the MongoDB client behavior in the experiments. We will start with the client pre-processing benchmark, then move onto the serialization methods, and communication protocols benchmark. Next, we will look into the performance of different parallelism levels. Finally, we will focus on setting the right batch interval to process data as fast as it is being received among different levels of parallelism.

4.1 Client Sending Behavior

As mentioned above, the client reads records from MongoDB and then sends them to the ingest server. Figure 4.1 represents the behavior of the client sending rate before the client sinks data into the ingest server. In our implementation, the client program controls and schedules the speed for sending records based on the speedup parameter. Specifically, the speedup parameter specifies how many times faster to replay the dataset in MongoDB. Since the time interval between two records in MongoDB is not stable, the actual sending rate is not stable either, thus resulting in a normal fluctuation as shown in the upper right corner of Figure 4.1. In the lower left corner of Figure 4.1, there is about one minute interval before the client sending rate increases to a relatively stable value. Because of this reason, we conduct some extra operations at the client side to perform all the benchmark experiments with the relatively stable client sending rate value, therefore ensuring a relatively stable input rate in our data processing pipeline.

Another important feature of the client is that it has a maximum sending rate limit. Therefore, as the speedup parameter increases, the client sending rate will gradually reach a threshold. The threshold comes from the speed the client can read from MongoDB and is also impacted by the client workload we have implemented given a considerable high speedup parameter configuration, especially the communication protocol we choose.

In addition, the ingest server we have implemented follows a reactive programming paradigm which acknowledges to the client only once a message has been acknowledged by Kafka. Therefore, the client sending rate is the same as the input stream rate in Spark Streaming given there is no bottleneck in the Kafka cluster.

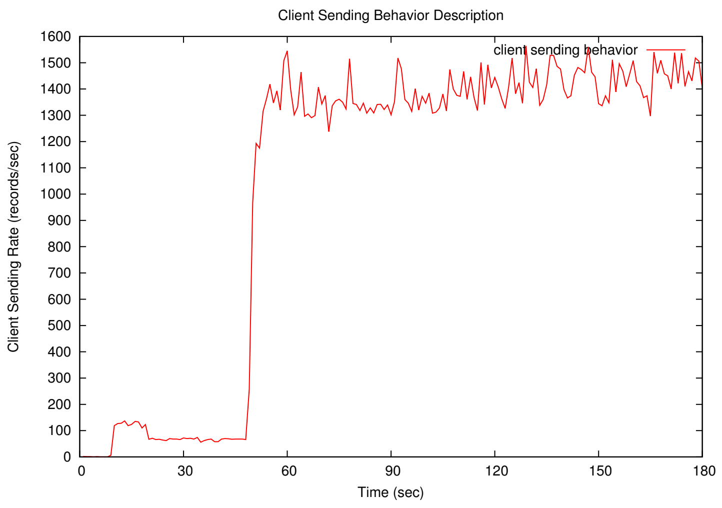


Figure 4.1: Client Workload Description

4.2 Impact of Client Pre-Processing

In this subsection, we compare and evaluate the batch processing time for the three client dataset versions called "cumulative counter", "counter" and "rate" under the same input stream rate in Spark Streaming. The benchmark experiments are performed with the following configurations. The serialization method is JSON and the communication protocol is HTTP. The number of Kafka partition is three, with the gateway_id extracted from the dataset field as the Kafka key. The batch interval in Spark Streaming is 5 seconds. The input stream rate is set with values varying from 450 records per second to 1200 records per second. The benchmark results are shown in Figure 4.2.

It turns out that the batch processing time of the counter version is about 30 percent smaller than the other two versions which have a very similar performance behavior between each other. The performance gap between the counter version and the other two versions grows with the input stream rate. In a word, the counter version we have implemented at the client side can be a good optimized solution compared to the inherent cumulative counter version while the rate version provides little optimization. The rate version is clearly not a wise choice for our dataset in MongoDB.

Obviously, the reason is that we have different implementations in the third and fourth steps of the benchmark experiments among the three client dataset versions, which group the records by key and calculate the cumulative sent and received traffic for the records with the same key. Each of these two steps in the

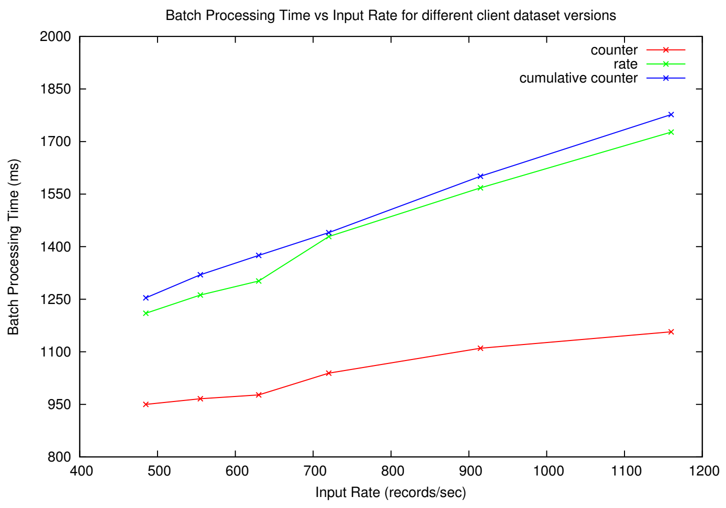


Figure 4.2: Client Dataset Version Benchmark

three client dataset versions is implemented as follows:

- counter
  1. Apply reduceByKey transformation on DStream to map records by Key.
  2. Calculate the aggregate sent and received traffic and the duration of period for each gateway in the DStream.
- cumulative counter
  1. Apply the dropping late messages function by the UpStateByKey transformation for deserialized key-value pairs (records) in the DStream.
  2. Apply groupByKey transformation on each RDD in the DStream to map records by Key.
  3. Sort the records within each RDD.
  4. Calculate the aggregate sent and received traffic and the duration of period for each gateway in the DStream.
- rate
  1. Apply the dropping late messages function by the UpStateByKey transformation for deserialized key-value pairs (records) in the DStream.

2. Apply reduceByKey transformation on DStream to map records by Key.
3. Calculate the aggregate sent and received traffic and the duration of period for each gateway in the DStream.

In the Spark Streaming jobs, as the records sequence read from Kafka is not guaranteed to be in order by the date of the time series dataset which is stored in MongoDB, some records may be associated with dates that are earlier than the latest date received in the previous batch. In the counter version, there is no need to deal with it because we can calculate the traffic during every time interval of the records to get the aggregate traffic value for the gateway in a day. However, in the cumulative counter version and the rate version, we need to drop this group of late messages otherwise we would calculate some traffic twice. Moreover, in terms of the impact of dropping late messages, the cumulative counter version implementation has no impact on the aggregate traffic value in the end unless the late messages that have been dropped owns the traffic value near the maximum threshold before it starts from zero again. However, the rate version implementation impacts the accuracy of the result since we use the mean rate in a batch interval to calculate the whole traffic. As a result, we use the UpdateStateByKey transformation on the input DStream to drop late messages. In consequence, the counter version implementation relieves the need to perform UpdateStateByKey DStream transformation in the phases described above, therefore the data processing performance of the counter version is better than the other two versions. As the input stream rate grows, the number of messages per batch interval increases and dropping late messages takes more time, thus increasing the performance gap between the counter version and the other two versions.

Another implementation detail is that we need to sort the records in the cumulative counter implementation since there is a maximum threshold for the sent and received traffic and we cannot calculate the whole traffic of a gateway based on the minimum and maximum timestamps. As we observed in Figure 4.2, this sort function should be the cause of the small performance gap between the cumulative counter version and the rate version.

In addition, we have tested the three client dataset versions in a single Kafka partition setup in order to observe whether there are different behaviors with respect to the performance gap among them compared to the three Kafka partition setup. And we particularly focus on the gap between the cumulative counter version and the rate version since the sort function is implemented in the cumulative counter version but not in the rate version. The result shows that the whole performance of the three client dataset versions in a single Kafka partition setup is quite similar to that in the three Kafka partitions setup. The reason is that the

Kafka message key is the gateway_id for more than one partition case. As we use the default partitioner in the Kafka producer, the records with the same key will be sent to the same Kafka partition for the given topic and therefore belong to the same RDD partition in Spark. Moreover, the sort function in the cumulative counter version is performed after the groupByKey transformation in which the key is a tuple of gateway_id and day_id in the deserialized key-value pairs. As a result, the sort function implemented in the DStream transformation process is within one RDD partition rather than across RDD partitions. The performance gap between the cumulative counter version and the rate version therefore stays quite small for different numbers of Kafka partitions.

Key takeaways: The counter pre-processing dataset version has lower data processing latency compared to the other two versions because its Spark job relies on commutative and associative functions, thus relieving the records to be processed in the same order as they are sent from the client. Specifically, dropping late messages as applied by the UpStateByKey DStream transformation in the cumulative counter version and the rate version results in a significant performance gap between them and the counter version. Moreover, the performance gap becomes larger as the input stream rate grows, up to about 40 percent under the current maximum client sending rate (around 1200 records per second). In addition, the sort function implemented in the cumulative counter version contributes to a small performance gap between it and the rate version and the gap stays stable for different levels of parallelism given the sort function is performed within one RDD partitions rather than across RDD partitions.

4.3 Impact of Serialization Methods

In this subsection, we compare and evaluate the batch processing time for two serialization methods – JSON and Avro – under the same input stream rate in Spark Streaming. The benchmark experiments are performed with the following configurations. The client dataset version is "cumulative counter" and the communication protocol is HTTP. The number of Kafka partitions is one. The batch interval in Spark Streaming is 5 seconds. The experiment variable – input stream rate – is set with values varying from 200 records per second to 1200 records per second. The benchmark results are shown in Figure 4.3.

It turns out that the processing time of Avro records is nearly the same as the processing time of JSON records, which means that the deserialization step in the Spark Streaming job does not bring much performance difference between the two serialization methods. In our experiments, the size of a JSON record sent from Kafka producer is about 180 bytes including record metadata. The size of

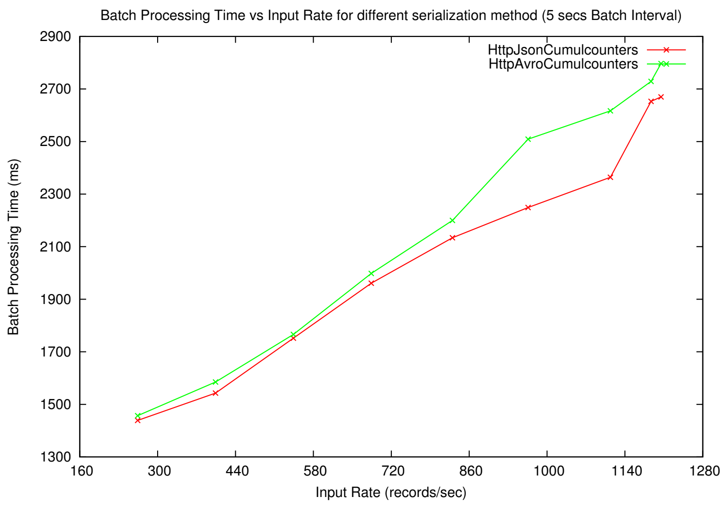


Figure 4.3: Serialization Method Benchmark

an Avro record is about 90 bytes, which is half the size of a JSON record. As the Avro records are much more compact than the JSON records, it costs much fewer network resources to transfer them. Therefore, Avro should be a better serialization method compared to JSON.

In addition, we have observed in our experiments that the throughputs in Kafka of JSON and Avro records are almost the same in terms of the number of records per second under the same sending speed scheduled in the client. However, according to the benchmark results from [50], the throughput regarding records per second will decrease as the record size grows under a considerable high input stream rate. Therefore, Avro serialization should have a better throughput over JSON serialization when we push to a significant client speed, which cannot be achieved currently for the records in MongoDB. However, the effect of message size on throughput needs to be taken into consideration for real time stream processing with larger number of home gateways in the production environment.

Key takeaways: Deserialization in Spark job does not bring much performance difference between the Avro and the JSON version. The throughput in terms of records per second in Kafka is the same between Avro and JSON serialization methods under current client sending rate, but Avro is likely to achieve higher value than JSON version under a high client sending rate. The record size in the Avro version is half of the size in the JSON version, which costs much fewer network resources. In conclusion, Avro is a wiser serialization method choice compared with JSON.

4.4 Impact of Communication Protocols

In this subsection, we compare and evaluate the processing pipeline throughput for two communication protocols – HTTP and WebSocket. The benchmark experiments are performed with the same configuration as the above serialization methods benchmark experiments and the serialization method is JSON. As the difference of HTTP and WebSocket lies in the sink process from the client to the ingest server, we actually only need to compare and analyze the client sending rate behaviour for HTTP and the WebSocket versions under different sending speedup parameters. The corresponding results for the relationship between the scheduled and real client sending rates are shown in Figure 4.4.

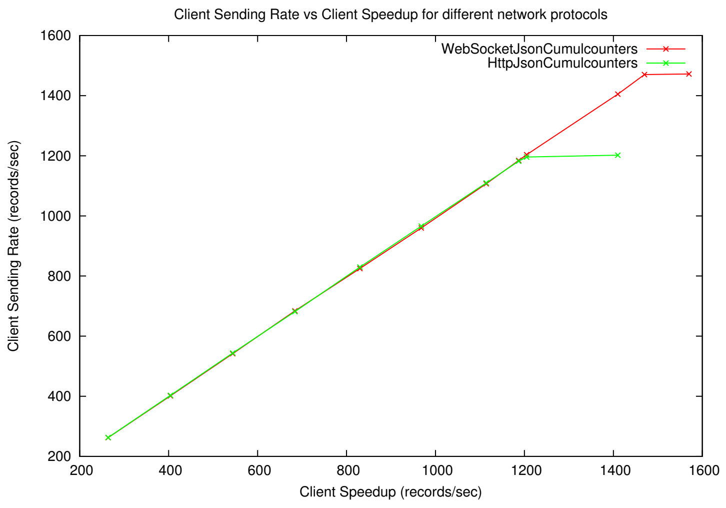


Figure 4.4: Communication Protocol Benchmark

It turns out that the WebSocket version has a higher maximum limit of sending rate compared to the HTTP version. The reason is that in the HTTP version, a new TCP connection is established for every record while in the WebSocket version, there is only one socket open for each gateway. As a result, the HTTP version creates large overhead for opening and closing of TCP connections while the WebSocket version saves computing, storage and networking resources for polling and making requests in the client side. Therefore, the HTTP version reaches a bottleneck earlier than the WebSocket version as the speedup parameter grows. In conclusion, WebSocket protocol is a wiser communication protocol choice compared to HTTP.

Key takeaways: WebSocket enables full-duplex communication through a single socket between a client and a server which saving computing, storage and networking resources for polling and making requests. In a real deployment, WebSocket relieves the ingest server to be burdened with unnecessary work, thus allowing the ingest server to handle a greater input rate than HTTP. In a word, WebSocket is a wiser communication protocol choice compared with HTTP.

4.5 Parallelism Benchmark

In this subsection, we compare and evaluate the input stream throughput and data processing latency in Spark Streaming for different numbers of Kafka partitions for the same topic. The benchmark experiment is performed with the following configurations. The client dataset version is "cumulative counter", the serialization method is JSON and the communication protocol is HTTP. The batch interval in Spark Streaming is 5 seconds. The experiment variable – the number of Kafka partition for each topic is set from one to three. The gateway_id extracted from the dataset field is used as the Kafka key. The input stream of Spark Streaming is consumed from a single topic in every experiment. We analyze both the input stream throughput and batch processing time in Spark Streaming. The result for batch processing time is shown in Figure 4.5.

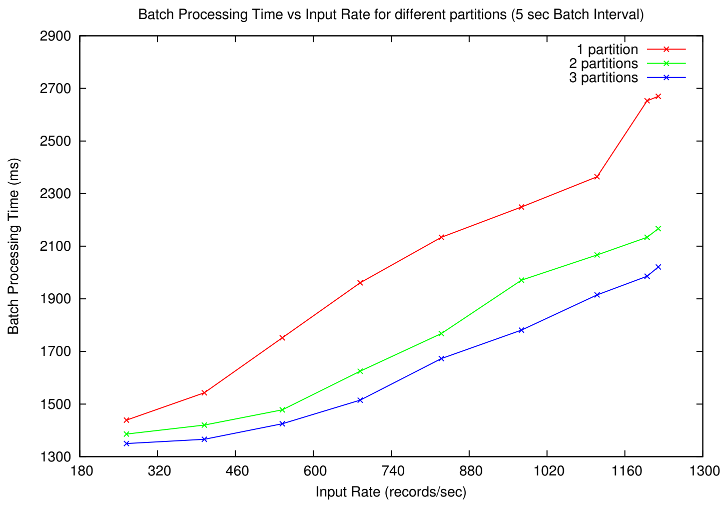


Figure 4.5: Parallelism Benchmark

The input stream throughput analysis refers to the impact of level of parallelism in data receiving. In our experiments, results are essentially the same for different number of Kafka topic partitions under the same client sending rate. The reason behind is explained as follows. As shown by the Kafka Manager monitoring tool [49], for different number of partitions, the upstream throughput is consistent with the downstream throughput, which means there is no bottleneck in Kafka. The total throughput in Kafka is the same for different number of partitions from one to three, which means that the level of Kafka parallelism has no benefit on Kafka throughput even under our maximum client sending rate. When it comes to the integration of Kafka with Spark Streaming, we use the Direct API for Kafka implemented in Spark 1.3 to ingest data from Kafka. This new API achieves a 1:1 correspondence between Kafka partitions and RDD partitions. In other words, it directly ties the degree of Spark parallelism to the degree of Kafka parallelism at least for reading messages. Therefore, the input stream rate in Spark Streaming is the same for different number of kafka topic partitions.

Our batch processing time evaluation refers to the impact of level of parallelism in data processing. As shown in Figure 4.5, the batch processing time gradually decreases as the number of RDD partitions in the direct Kafka input stream grows under the same input stream rate. The reason behind is explained as follows. When running the Spark Streaming job, the one-to-one mapping between RDD partitions and Kafka partition does not remain after the reduceByKey DStream transformation, which is a distributed shuffle method implemented in our Spark Streaming job. In Spark cluster mode, the default number of parallel tasks for shuffle transformations like reduceByKey is the largest number of partitions in a parent RDD. In our case, we followed this default configuration, therefore the number of parallel tasks for reduceByKey DStream transformation is related to the RDD partitions in the direct Kafka input stream. As a result, in our experiments, the level of parallelism in data processing increases when increasing the number of kafka topic partitions. Therefore, we attain a gradually declining batch processing time when increasing the number of kafka topic partitions.

However, whether to choose Kafka direct consumer for Kafka integration requires further benchmark with respect to data processing latency. On the one hand, Kafka direct consumer uses the Kafka simple consumer API while the old receiver-based Kafka integration approach uses Kafka high-level consumer API. Since Kafka simple consumer API does not need to connect to Zookeeper to retrieve offsets information while Kafka high-level API relies on Zookeeper to manage offsets, hence Kafka direct consumer is much faster than receiver-based consumer.

On the other hand, Kafka direct consumer may contribute to a lower data processing parallelism compared to the old Kafka integration approach [51]. As mentioned before, it directly ties the number of RDD partitions with the number of kafka topic partitions, which will be the maximum processing parallelism level. When it comes to the old Kafka integration approach which relies on the receiver, the number of RDD partitions in the input DStream corresponds to the number of blocks in each batch, which is controlled by the receiver's block interval. As a result, the number of parallel tasks for a shuffle DStream transformation per receiver per batch will be the result of batch interval divided by block interval [12]. The receiver's block interval can be configured manually, thus allowing the number of RDD partitions to be larger than the number of kafka topic partitions. Therefore, unless the Kafka topics have larger partitions than RDD partitions in the old Kafka integration approach, Kafka direct consumer will have a low level of parallelism for RDD processing.

Furthermore, regarding the effect of the level of Kafka parallelism, it actually impacts the Kafka throughput under considerable high ingest rate according to the benchmark results in [52]. Other metrics such as the number of producer and replica also have effect on the maximum throughput of Kafka cluster. In conclusion, the above mentioned factors as well as other related Kafka parameters should be taken into consideration for performance tuning when deploying our coherent data processing pipeline on cloud providers like AWS with larger number of home gateways in the production environment.

In addition, one important aspect in the production environment is to adapt to dynamic data source output rates. In the Spark Streaming configuration, we are allowed to set a maximum read rate for each Kafka partition in Kafka direct consumer. However, since it cannot enforce a dynamic maximum rate, we should rely on the new backpressure feature introduced in Spark 1.5. As shown in [53], Spark 1.5 starts to support the backpressure feature to facilitate the construction of end-to-end reactive applications. In our case, this new feature will probably allow us to deal with dynamic data source output rate and relieve the need to buffer with Akka stream in the ingest server. However, enabling the backpressure feature sometimes brings negative effect. Spark may still not be able to keep up with the throughput after the backpressure feature changes the batch length and results in longer latency [54]. Therefore, whether to use the backpressure feature should be explored in the production environment with large and dynamic client throughput.

Key takeaways: The Kafka throughput is the same for different levels of parallelism even under our maximum client sending rate. The Spark input stream rate is directly tied with Kafka throughput using Kafka direct consumer, and therefore the same for different levels of Kafka parallelism. Furthermore, the level of parallelism in data processing is directly tied with the level of Kafka parallelism using Kafka direct consumer with our benchmark Spark Streaming job, resulting in lower data processing latency for higher level of Kafka parallelism. Moreover, Kafka direct consumer is likely to have lower level of parallelism in data processing compared with receiver-based Kafka integration approach when the number of Kafka topics is very small and requires further benchmarking regarding data processing latency. In addition, the backpressure feature introduced in Spark 1.5 facilitates the construction of end-to-end reactive applications. Specifically, it handles dynamic data source output rate and relieves the need to buffer with Akka stream in the ingest server. Since it may sometimes result in longer data processing latency, the backpressure feature needs to be explored in the production environment with large and dynamic client throughput. Kafka metrics such as the number of producers and replicas also need to be explored in the production environment in terms of Kafka throughput.

4.6 Batch Interval Benchmark

In this subsection, we explore the minimal batch interval for a stable Spark Streaming application under different number of partitions. The benchmark experiments are performed with the following configurations. The client dataset version is "cumulative counter", the serialization method is JSON and the communication protocol is HTTP. The number of Kafka partition varies from one to three. The batch interval is set from one second to four seconds. The client sending rate is 1200 records per second. In our experiments, there is no bottleneck in our coherent data processing pipeline, and the input stream rate for Spark Streaming is stable and copes well with client sending rate in every experiment. The benchmark experiment result for setting the right batch interval for our stable Spark Streaming application is shown in Figure 4.6.

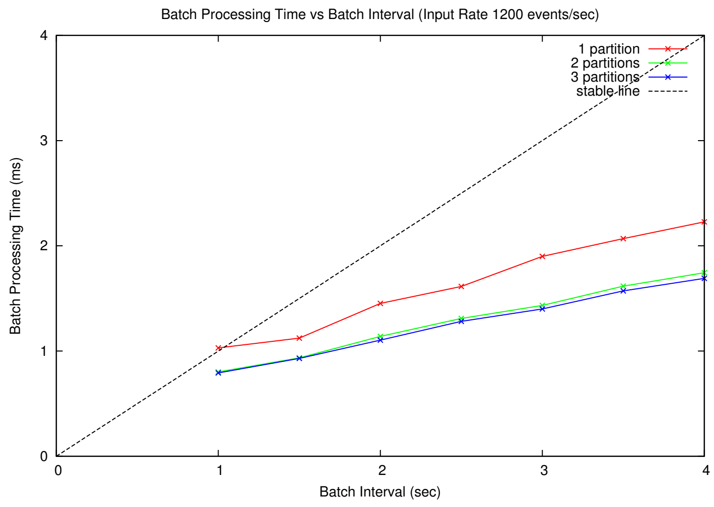


Figure 4.6: Batch Interval Benchmark

It turns out that our Spark Streaming application has a roughly-linear relationship between batch intervals and processing times. This behavior is similar to that of reduce transformation in the batch interval performance test experiments [1]. Since our main transformation in Spark Streaming job is groupByKey, the roughly-linear relationship between the intervals and the processing times is reasonable for our Spark Streaming application.

Moreover, the stable line (the dotted line) in Figure 4.6 refers to the state when the batch processing time is equal to the batch interval, which identifies the stable operating zone. Any batch interval whose processing time is below this line will be stable and vice versa [1]. And for stable batch interval, the whole processing time decreases as the batch interval decreases given the same amount of ingested records. Therefore, we explore the minimal batch interval in terms of minimizing latency.

In the one Kafka partition case, i.e., one RDD partitions case, the minimal batch interval for a stable Spark Streaming application is about one second as shown in Figure 4.6. We observed that the scheduling delay becomes very high in the one-second batch interval scenario, whereas it is zero for stable applications. Meanwhile, the input stream rate is decreased resulting from the backpressure feature in Spark Streaming. Therefore, for batch interval below one second, our Spark Streaming application will be unstable. With respect to the two and three RDD partitions case, the minimal batch interval is below one second which benefits from the level of parallelism in data processing as illustrated in Subsection 4.5. And the batch processing time gap between different number of RDD partitions increases as the batch interval increases which lies in that the impact of level of parallelism increases as the number of records increases in a batch.

In addition, in every batch interval, the batch processing time grows as the input stream rate increases. Therefore, a maximum input stream rate exists for a stable Spark Streaming application with respect to batch intervals. It has been shown that the maximum input stream rate gradually increases for Spark Streaming jobs that have reduce transformations [1]. Therefore, the maximum input stream rate with respect to batch intervals is likely to have a similar behaviour in our application. Regarding our current input stream rate which is around 1200 records per second, it is the maximum input stream rate for one second batch interval in one Kafka partition case. Because the client has a maximum sending limit around 1200 records per second in our experiment, we can focus on reducing batch interval to minimize latency. However, in the production environment with large and dynamic throughput, it will be a more challenging process to find out the optimal batch duration which minimizes latency while allowing Spark Streaming to handle the throughput.

Key takeaways: Our Spark Streaming application has a roughly linear relationship between batch intervals and batch processing times which is likely results from the groupByKey transformation in the Spark Streaming job. The minimal stable batch interval decreases when increasing the level of parallelism in data processing given the same input stream rate. Furthermore, the scheduling delay grows rapidly and the input stream rate decreases because of the backpressure feature in the unstable operation zone. Based on the current relatively low stable maximum input stream rate, we can focus on reducing the batch interval to minimize latency. However, we need to strike a balance between minimizing latency and handling dynamic high throughput in the production environment.

5 Conclusion

In this thesis, we mainly conducted a series of benchmark experiments for processing home gateway traces with our data processing pipeline. The goal of the benchmark experiments was to identify the impact of various client data workloads as well as to get the best performance out of our Spark Streaming applications. We compared and evaluated the pipeline throughput and Spark Streaming batch processing latency in benchmark experiments.

In the beginning, we have introduced big data processing technologies, especially real time stream processing based on our data processing pipeline. We focused on the design, abstraction, cluster architecture and integration of Spark Streaming and Kafka. We highlighted the lambda architecture which is similar with the architecture of our data processing pipeline. The lambda architecture seamlessly integrates batch and streaming processing and enables fast, streaming computations on time series data in asynchronous Akka event-driven environments.

Moving onto the part of experiment design, we first illustrated our home gateway dataset versions along with serialization methods and communication protocols that are designed and implemented in our client. Consequently, a bundle of client workload implementations is ready for benchmark experiments. Then we outline our benchmark design with a flow of operations in our Spark Streaming applications. Next, we introduce two cluster deployment tools used for our data processing pipeline, namely Vagrant and Ansible. Finally, we present monitoring tool used for our benchmark experiments, including Spark application web UI, Spark history server, Spark cluster master web UI and Kafka-manager followed by the explanation on how to setup the benchmark.

Furthermore, we have implemented end-to-end reactive Spark Streaming applications for all client workload implementations. It turns out that there is no bottleneck in our data processing pipeline given the highest client sending rate for all client data workloads. In other words, there is currently no need to look into Kafka throughput. Therefore, we mainly focused on the batch processing time performance of Spark Streaming applications for analyzing the impact of various client data workloads. The benchmark results show that the counter client preprocessing version which relieves home gateway records from being processed in the same order as they are sent from client has the least DStream transformations in Spark jobs, thus leading to the lowest batch processing time. In addition, Avro and WebSocket are the better performing serialization method and communication protocol choice compared with JSON and HTTP, respectively.

Moreover, we have focused on two aspects for Spark Streaming performance tuning which are increasing levels of parallelism and setting the right batch interval in order to achieve stable processing with low data processing latency. In terms of increasing the level of parallelism (in our experiments, the level of parallelism in data receiving and data processing simultaneously), the benchmark results show that the higher level of parallelism leads to lower data processing latency but has no throughput benefit for current highest client sending rate. With respect to setting the right batch interval, the benchmark results show that the Spark Streaming application had a roughly linear relationship between batch intervals and batch processing times. The minimal stable batch interval decreases when increasing the level of parallelism given the same input stream rate.

At the current stage, we have a relatively low client sending rate maximum threshold as well as a stable client sending rate. As a result, we did not experience any bottleneck with a single Kafka producer and increasing Kafka topic partitions brought no benefit to Kafka throughput. We were also allowed to focus on reducing batch interval to minimize latency rather than figuring out an optimal batch interval to handle both throughput and latency. However, a number of Kafka and Spark Streaming parameters need to be taken into consideration in the production environment to adapt to higher dynamic data source output rate in real time.

On the Kafka side, we are likely to need more producer threads to handle the high throughput of the data source. We also need to add replicas for Kafka partitions to ensure the fault-tolerance of the Kafka cluster. Furthermore, we are likely to benchmark Kafka performance with added producers and replicas as shown in [50]. Moreover, even though we will add replicas in the future, Kafka may still lose messages if the leader of a partition is offline with some committed messages that have not been synchronized with other replicas as described in [55]. Therefore we should look into details about replica related parameters configuration in the deployment.

On the Spark Streaming side, we are likely to explore the optimal batch interval to minimize latency while handling a higher dynamic data source output rate in the production environment. We also need to choose whether to enable the backpressure feature for dynamic data source output rate with its impact on data processing latency. Furthermore, we are likely to tune the memory usage and GC behavior of the Spark applications based on our transformation in Spark Streaming job to reduce memory usage, GC overhead and data processing latency [12]. In addition, we will look into more complex analytic tasks for Spark Streaming applications, for example, calculating the mean traffic for all gateways or other machine learning analytics tasks [56].

References

[1] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[2] Nitin Sawant and Himanshu Shah. *Big Data Application Architecture Q & A*. Springer, 2013.

[3] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[4] Jay Kreps. Putting Apache Kafka To Use: A Practical Guide to Building a Stream Data Platform (Part 1). `http://www.confluent.io/blog/stream-data-platform-1/`. Referred: 2016-6-29.

[5] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[6] Rajiv Ranjan. Streaming big data processing in datacenter clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014.

[7] Michael Walker. batch-vs-real-time-data-processing. `http://www.datasciencecentral.com/profiles/blogs/batch-vs-real-time-data-processing`. Referred: 2016-6-29.

[8] Helena Edelson. Lambda Architecture with Spark Streaming, Kafka, Cassandra, Akka, Scala. `http://www.slideshare.net/helenaedelson/lambda-architecture-with-spark-streaming-kafka-cassandra-akka-scala/4`. Referred: 2016-6-29.

[9] Matthew Franklin. The berkeley data analytics stack: Present and future. In *Big Data, 2013 IEEE International Conference on*, pages 2–3. IEEE, 2013.

[10] Christopher Batey. Spark with Cassandra. `http://www.slideshare.net/SparkSummit/spark-with-cassandra-by-christopher-batey`. Referred: 2016-6-29.

[11] Patrick McFadin. killrweather – a reference application. `https://github.com/killrweather/killrweather`. Referred: 2016-6-29.

[12] Spark Docs. Spark Streaming Programming Guide – Performance Tuning. `http://spark.apache.org/docs/latest/streaming-programming-guide.html#performance-tuning`. Referred: 2016-6-29.

[13] Stephan Brosinski. Tuning Spark Streaming Applications. http://blog.smaato.com/tuning-spark-streaming-applications/. Referred: 2016-6-29.

[14] Jeroen Van Wilgenburg. Spark Streaming Back Pressure – finding the optimal rate is now done automatically. https://vanwilgenburg.wordpress.com/2015/10/06/spark-streaming-backpressure/. Referred: 2016-6-29.

[15] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 295–308, 2012.

[16] Alexander Rasmussen, Michael Conley, George Porter, Rishi Kapoor, Amin Vahdat, et al. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 13. ACM, 2012.

[17] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[18] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[19] Databricks. What is Apache Spark. https://databricks.com/spark/about. Referred: 2016-6-29.

[20] Paco Nathan. Spark Overview: Spark Components. http://ictlabs-summer-school.sics.se/2015/slides/spark.pdf. Referred: 2016-6-29.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[22] Reynold Xin. Spark Summit Training – Advanced Spark. `https://databricks-training.s3.amazonaws.com/slides/advanced-spark-training.pdf`. Referred: 2016-6-29.

[23] Spark Docs. Cluster Mode Overview. `http://spark.apache.org/docs/latest/cluster-overview.html#components`. Referred: 2016-6-29.

[24] Tathagata Das. Recipes for running Spark Streaming Applications. `http://www.slideshare.net/SparkSummit/recipes-for-running-spark-streaming-apploications-in-production-tathagata-daspptx`. Referred: 2016-6-29.

[25] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.

[26] Tathagata Das. Dive Deep with Spark Streaming. `http://www.slideshare.net/spark-project/deep-divewithsparkstreaming-tathagatadassparkmeetup20130617`. Referred: 2016-6-29.

[27] Tathagata Das. Spark Streaming Implementation Details. `https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html`. Referred: 2016-6-29.

[28] Cody Koeninger, Davies Liu, and Tathagata Das. Direct API for Kafka. `https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html`. Referred: 2016-6-29.

[29] Khin Me Me Thein. Apache Kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483, 2014.

[30] Andrew Brust. Kafka Stream Data Platform. `http://www.zdnet.com/article/kafka-0-9-and-mapr-streams-put-streaming-data-in-the-spotlight/`. Referred: 2016-6-29.

[31] Kafka Docs. Topics and Logs. `http://kafka.apache.org/documentation.html#intro_topics`. Referred: 2016-6-29.

[32] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[33] Gwen Shapira. What is the actual role of ZooKeeper in Kafka. https://www.quora.com/What-is-the-actual-role-of-ZooKeeper-in-Kafka. Referred: 2016-6-29.

[34] Anand Iyer and Gwen Shapira. Deploying Apache Kafka: A Practical FAQ. http://blog.cloudera.com/blog/2015/07/deploying-apache-kafka-a-practical-faq/. Referred: 2016-6-29.

[35] Michael G. Noll. Kafka Cluster Overview. http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node/. Referred: 2016-6-29.

[36] Fuguo HUANG. Web Technologies for the Internet of Things, Master's Thesis, Aalto University, 2013. https://into.aalto.fi/download/attachments/12324178/Huang_Fuguo_thesis_2.pdf.

[37] Wikipedia. JSON. https://en.wikipedia.org/wiki/JSON. Referred: 2016-6-29.

[38] Confluent. Data Serialization and Evolution. http://docs.confluent.io/1.0/avro.html. Referred: 2016-6-29.

[39] Apache. Apache Avro^{TM} 1.8.1 Specification. http://avro.apache.org/docs/current/spec.html#Data+Serialization. Referred: 2016-6-29.

[40] IBM. What is Avro? https://www-01.ibm.com/software/data/infosphere/hadoop/avro/. Referred: 2016-6-29.

[41] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[42] Jasdeep Jaitla. WebSockets vs REST: Understanding the Difference. https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/. Referred: 2016-6-29.

[43] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[44] VirtualBox. https://www.virtualbox.org/. Referred: 2016-6-29.

[45] Vagrant. https://www.vagrantup.com/. Referred: 2016-6-29.

[46] Ansible. https://www.ansible.com/. Referred: 2016-6-29.

[47] RJ Zaworski. Using Ansible with Vagrant. https://rjzaworski.com/2015/07/using-ansible-with-vagrant. Referred: 2016-6-29.

[48] RJ Zaworski. Bring your production environment home with Vagrant and Ansible. https://rjzaworski.com/2015/07/bring-your-production-environment-home-with-vagrant-and-ansible. Referred: 2016-6-29.

[49] Yahoo. A tool for managing Apache Kafka. https://github.com/yahoo/kafka-manager. Referred: 2016-6-29.

[50] Jay Kreps. Benchmarking Apache Kafka. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines. Referred: 2016-6-29.

[51] Dibyendu Bhattacharya. High Performance Kafka Consumer for Spark Streaming. https://github.com/dibbhatt/kafka-spark-consumer. Referred: 2016-6-29.

[52] Jun Guo. Apache Kafka Benchmark. http://www.jasongj.com/2015/12/31/KafkaColumn5_kafka_benchmark/. Referred: 2016-6-29.

[53] Tathagata Das. Spark Streaming Back Pressure – finding the optimal rate is now done automatically. http://www.slideshare.net/Typesafe_Inc/four-things-to-know-about-reliable-spark-streaming-with-typesafe-and-databricks. Referred: 2016-6-29.

[54] Sanket Chintapalli. Benchmarking Streaming Computation Engines at Yahoo. https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at. Referred: 2016-6-29.

[55] Gwen Shapira. Kafka Reliability - When it absolutely, positively has to be there. http://www.slideshare.net/gwenshap/kafka-reliability-when-it-absolutely-positively-has-to-be-there/6. Referred: 2016-6-29.

[56] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.