Vrije Universiteit Amsterdam

Bachelor Project

Analysing the Kafka Producer and Benchmarking its Performance.

Author: De Vos Meaker (2617529)

1st supervisor: dr. ir. Animesh Trivedi
2nd reader: ir. Jesse Donkervliet

A thesis submitted in fulfilment of the requirements for the VU Bachelor of Science degree in Computer Science

August 14, 2020

Abstract

To determine and improve the performance of distributed systems like Apache Kafka is a complex and necessary undertaking. Trends show that data generation is increasing at an exponential rate, and a large portion of this is becoming real-time data. With mainstream adoption, Apache Kafka is the de facto solution for real-time data streaming problems. Here, real-time data refers to data that is sent or processed as it arrives, such as a feed of weather or transport data. This study aims to understand Kafka and determine how to improve its performance. Extending on previous work, it asks: What is the performance of the Kafka Producer and can we explain and quantify its behaviour? The Kafka Producer is the interface used to write data to Kafka.

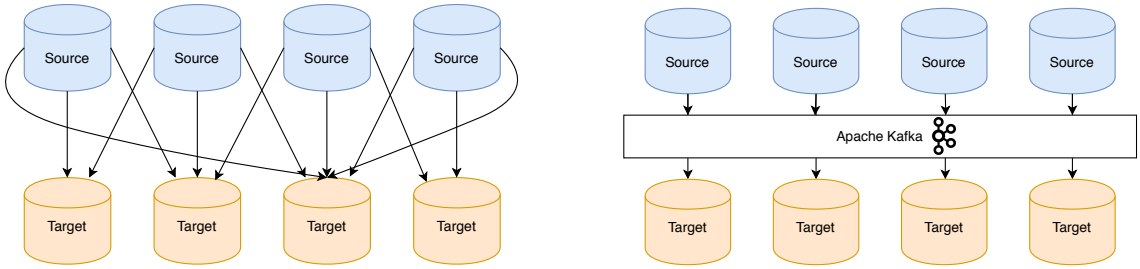
Empirical measurements of a running computer system were made to answer this question. The Producer's "out-of-box" performance is set as the baseline, thereafter incremental configuration changes are made, and their effects on performance are measured in isolation. The source code of the experiments is at this Github repository [12]. Analysis of the measurements showed unexpected behaviour which is caused by the batch.size configuration. Findings indicate that the batching process is central to the Kafka Producer and therefore it is recommended to focus on improving the batch.size configuration due to its influence on this process. Further investigation is required to determine the overall behaviour and performance of the Kafka system and generalise results in production environments.

Contents

1 Introduction

1.1 Context

At the time of writing more than 2.5 quintillion (10^{18}) bytes of data are generated per day [3]. This data generation will only increase due to more than half of the world's population, which still has to connect to the internet [1] coupled with the trend of ubiquitous technology [15]. Furthermore, by 2025 almost 30% of this generated data will be real-time data, up from 15% in 2018 [9]. Real-time data refers to information that is immediately delivered after it becomes available and usually seen as a real-time feed of events due to its timeliness. Examples include weather data or financial trading data. This continuous volume of events, or data streams, poses many challenges to organisations. One of which is organising and moving the data between the data sources, which generates the data, and data targets, which consumes the data. A challenge which can become very complex, as illustrated in Figure 1:



(a) The complexity of the problem.

(b) The simplified solution Kafka provides.

Figure 1: The problem of connecting data sources with data targets and the solution Kafka provides [25].

Apache Kafka is a popular real-time event processing platform to solve this problem. Published as an open-source project [5] in 2011, it has since grown to become the second most visited and downloaded Apache Project [8]. It is being used by over 60 Fortune 100 organisations [17]. These include Uber [13], Netflix [27] and LinkedIn [18], who are each using it to send trillions of messages per day. Furthermore, it is deployed in a wide range of domains ranging from finance [23] to online games [24].

Several trends are driving the adoption of Kafka and other platforms in the real-time data streaming and processing space. A market that is expected to grow from USD 7.08 billion in 2019 USD to 38.53 billion by 2025 [19]. Big Data Analysis and Applications, Microservices and Artificial Intelligence are some of these trends. Therefore the need for Kafka will increase, both in scale and variety and it will still be held to strict performance requirements that real-time data streaming demands.

1.2 Problem Identification

Thus Kafka has mainstream adoption and serves in a wide range of domains. The problem is, therefore, to determine and improve Kafka's performance. Solving this will improve the

quality of services deploying Kafka and save various resources, namely time, computing and energy costs. Due to the complexity involved with measuring a distributed system such as Kafka, this study will focus on the Kafka Producer, which is the interface used to write data to Kafka. It is a very influential component in the Kafka system and can be isolated to a high degree to be adequately investigated. Furthermore, the Producer is involved with many intricate processes, such as ensuring the durability and synchronisation of the data written to Kafka. Therefore, its performance is critical to the Kafka system as a whole.

1.3 Objective of the Research

Therefore this study aims to understand the Kafka Producer's implementation and determine its default "out-of-the-box" performance. Additionally, analyse the influence of various configurations on its performance. These configurations are settings that can be changed to improve Kafka's performance. Lastly, we determine to what extent it can utilise the available networking and computing power.

1.4 Contribution of This Paper

The contributions of this thesis are:

(i) Provide an analysis of the Kafka Producer's internals.

(ii) Thorough performance evaluation and analysis of influential Kafka Producer configurations.

(iii) Confirm if the Kafka Producer behaves as expected under different workloads.

(iv) Indicate the performance limits of the Kafka Producer on the available hardware.

1.5 Approach and Methods

Empirical measurements of a running computer system were made to reach the goal of this study [21]. The performance of the Kafka Producer was measured on a local machine, see Section 4.2 for the environment. The metrics measured to determine performance are throughput and latency, which is defined in Section 5.3. The correctness of these metrics is also ensured by having multiple measurements for the same metrics. The conducted experiments were systematically guided by the research questions. Which begins with Kafka's default performance to establish its baseline and thereafter incrementally changing and testing configurations in isolation to quantify their influence on performance.

1.6 Outline of Paper

First Section 2 names and comprehensively motivates the research questions. Next, Section 3 explains and analyses all the concepts and processes of Kafka and the Kafka Producer necessary for the scope of this study is. Section 4 covers the setup and design of the experiments and explains how the correctness of the measurements is ensured. Section 5 deals with the benchmarks performed and accompanied each with a thorough analysis of

the results, and the best results are summarised in Section 6. Section 7 provides more in-depth analysis of the results, compares the findings with that of previous work performed and propose ideas for potential future work. Lastly, Section 8 contains the conclusion.

2 Research Questions

The research question that is being addressed in this thesis is: **What is the performance of the Kafka Producer and can we explain and quantify its behaviour?**

The following tangible questions and their motivations will serve to answer this central question:

(RQ1) Can we understand the Kafka Producer at a level of depth necessary to explain and quantify its behaviour?
In order to analyse the performance of the Kafka, it is necessary to understand it at a significant level of depth. Which includes acknowledging its initial purpose as a solution for real-time 'website activity tracking data' streaming [7] and identifying the design decisions taken to address this problem. Furthermore, exploring more use cases and the performance requirements that they demand. The central method in the Kafka Producer interface is the send() method, which performs many functions that are abstracted away from the developer and it is necessary to identify and evaluate the underlying processes that occur when it is called. This knowledge will prohibit the developer from treating the Kafka Producer as a black box and enable him or her to make more educated decisions when tweaking configurations to improve the performance of their Kafka Producer and diagnose unexpected behaviour.

(RQ2) How does the Kafka Producer perform with its default configurations?
The Kafka Producer alone has more than 70 configurations, and it also interacts with other components in the Kafka system(see Section 3.1). The other components add many additional configurations that need to be considered. Due to many of the configurations being dependent on each other, they cannot necessarily be evaluated in isolation, but their relationships must also be acknowledged. These configurations make Kafka highly adaptable to many use cases [7]. Thus the most appropriate approach to generalising Kafka is to run it on its default configurations, assess its performance on the default configurations and treat this as its baseline performance.

(RQ3) What are the influences of the individual configurations on the performance of the Kafka Producer?
The previous paragraph highlights the sheer number of configurations that can be configured, which creates a problem of high complexity. Although daunting, it is necessary to understand the influence of configuration on the Producer's performance. Not knowing the influence will lead to little more than random guessing. This testing must be performed systematically, incrementally changing one configuration at a time at different intervals while measuring its performance. The results must also continuously be compared against the baseline to guide further experiments.

(RQ4) What are the minimal latency and maximal throughput of the Kafka producer on the provided hardware?
What is the minimum latency that can be achieved on the available hardware, not considering the trade-offs that will result from this decision? To elaborate, configuring the Producer for minimal latency will be at the expense of many other metrics, namely throughput and durability. Although it will be rare to configure a Kafka Producer to serve such an

extreme use case in production, it is good to know where its limits are. Knowing these limits will provide points of reference of where a configuration's performance lies on the possible spectrum. Moreover, they will aid in assessing if the performance of Kafka Producer is sufficient given the available hardware and necessary trade-offs demanded by the use case.

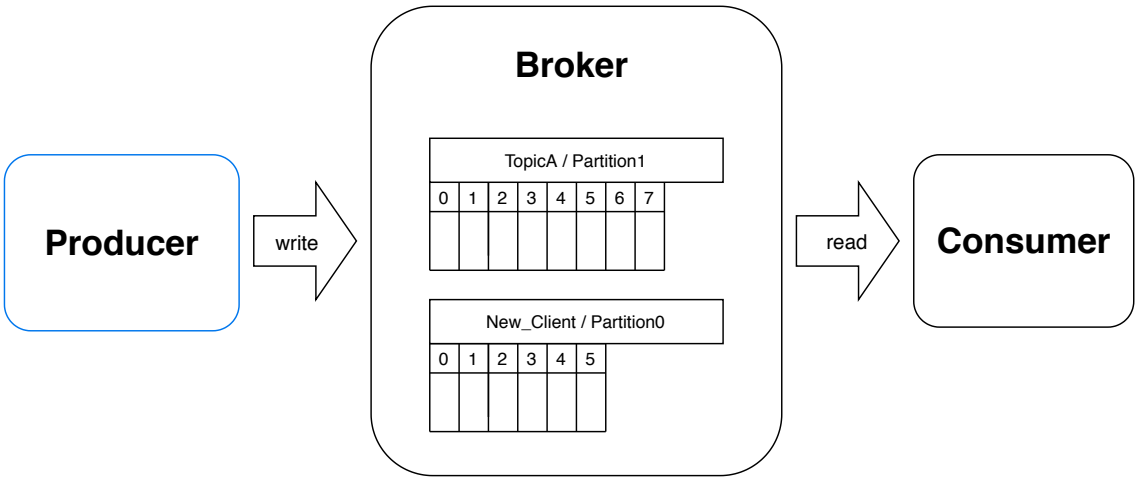


Figure 2: Kafka Overview

3 Background: A Quick Dive Into Kafka

3.1 What Is Kafka?

Kafka is a distributed message streaming solution. The project began in LinkedIn as a solution to the need for real-time event processing. An 'event' can, in the case of LinkedIn, be a user connecting to another user. This event needs to trigger several processes such as the updating of the users' network, sending notifications and updating friend recommendation systems. The process of connecting this data source event to all of the individual data targets creates complexity. Each integration has a list of complexities such as data format, protocol, data scheme and handling increased load. Kafka simplifies all this integration with a central commit log. This problem and solution are illustrated in Figure 1.

3.2 Design Overview

The main design decision of Kafka is the central commit log, a simple append-only data structure containing the data written to Kafka. This log is persistent, immutable and ordered. Another critical decision was to enable Kafka to scale horizontally to serve large scale applications. To accomplish this, it is necessary to build distributed platform concepts such as partitioning, replication and fault-tolerance into its design [11]. Thus, a 'topic', which is an abstraction of the commit log, can be partitioned to scale and replicated to increase fault-tolerance [10]. A topic can be compared to a table in a relational database. Here it will contain a stream of events that are relevant to it.

The core components of the Kafka system [11] are:

(i) the Producer, which is the interface used to interact with the Broker and writes records to the topic via high-level API operations.

(ii) the Broker that contains and maintains the topics and

(iii) the Consumer, which is the interface used to interact with the Broker and read records from the topic via high-level API operations.

3.3 A Deep Dive Into the Kafka Producer's Send() Method:

The purpose of this Section is to partially answer **RQ1**. The send() method is the method in the Kafka Producer API used to write records to and interact with the Kafka cluster. To better understand Kafka's behaviour and its performance, it is necessary to know what happens internally when send() is called. Send() is a member method of a KafkaProducer class, it requires a ProducerRecord as a parameter and to create one, a topic and data are required. Further optional metadata can be added, but these will not be discussed for brevity. The series of steps that occurs after the send() method is called asynchronously are illustrated in Figure 3 and are as follows:

1. The record is sent as a parameter of the send() method, by the application.
2. The record value will be serialised in the desired format by the serialiser.
3. The record will then be assigned a partition number that belongs to the record's topic. This partition number determines the record's destination and is assigned by the partitioner in a round-robin fashion.
4. The record will be compressed and batched together with other records that are designated for the same topic and partition combination.
5. The batches destined for the same topic partition are queued together in the record accumulator.
6. The sender thread polls the queues for new batches.
7. The batches available for sending are sent to the sender thread.
8. The sender thread groups batches together that are ready to be sent and destined to the same leader broker. The records are ready to be sent when **`batch.size`** has been reached or **`linger.ms`** has expired. These are two configurations that determine the amount of batching performed and is discussed in Section 3.4. The leader broker contains the topic partition and receives new data destined for this partition from the producer. It also distributes it to other brokers, called follower brokers, who contains the replicas of the topic partitions and keeps this in-sync with the leader's original.
9. The sender thread sends the grouped batches to the leader broker.
10. The leader broker adds the batches to the relevant topic partitions, by writing it to its relevant local logs.
11. The follower broker/s sends a fetch request.
12. The leader broker sends the group of batches as one to all the follower brokers that contains a replica of the topic partition.
13. The follower broker/s adds the batches to the relevant topic partitions, by writing it to its relevant local logs.

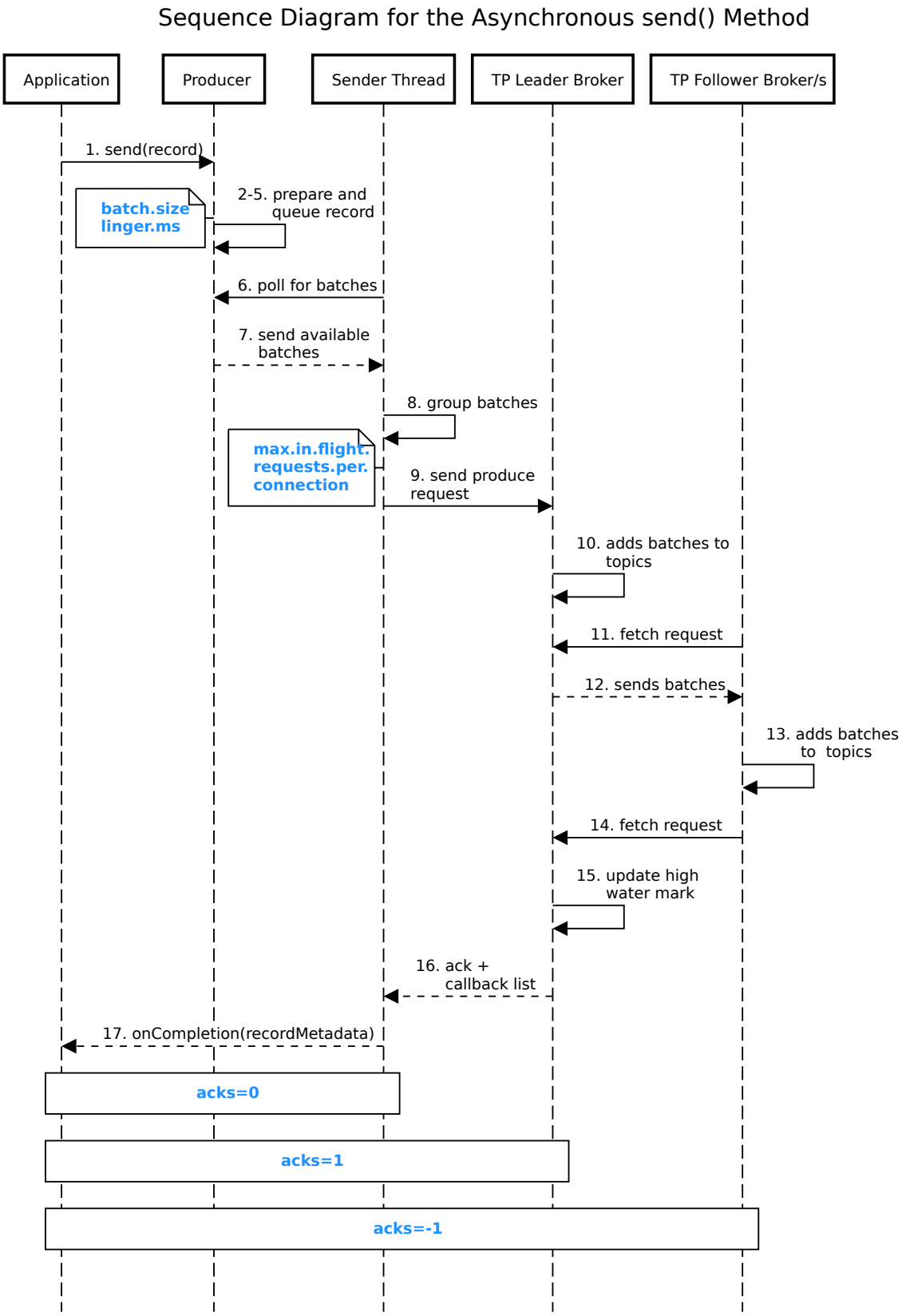


Figure 3: Kafka Producer Overview [5, 22].

Table 1: acks Configuration Performance Trade-Offs [22]

acks	Throughput	Latency	Durability
0	high	low	no guarantee
1	medium	medium	leader
all	low	high	in-sync replica

14. The follower broker/s sends another fetch request for a batch with an offset higher than the previously sent batch, which confirms that these previous batches are successfully written to its local partition.
15. The leader broker updates the high watermarks for each partition. High watermarks are the offsets which have been successfully replicated on the follower brokers.
16. The acknowledgement of the original produce request and list of callbacks containing metadata is sent from the leader broker to the sender thread.
17. The application receives a confirmation that the record has been either successfully or unsuccessfully produced. Metadata also accompanies this confirmation which is accessed through the onCompletion() method of the CallBack class.

This analysis helps to understand the Producer and identify the configurations which have a significant influence on its processes and subsequently its performance. The next section below discusses these configurations in detail.

3.4 Important Kafka Configurations

The important configurations [6] identified in the analysis and examined in the benchmarks are provided below with their default values, units and short descriptions.

`acks` : 1
Depending on the acknowledgement configuration the record will be considered successfully sent when each batch in the group has been:

- **`acks`**=0 : sent to the leader broker, with no acknowledgement from the broker.
- **`acks`**=1 : sent and written to the local partition of the leader broker and receives an acknowledgement.
- **`acks`**=-1 : written to the local partition of the leader broker and each follower broker containing a replica of the partition. The leader broker confirms that the replicas are in-sync, of which an acknowledgement is received [6].

Table 1 illustrates the rule of thumb trade-offs involved when selecting an acknowledgement configuration which can be changed to suit the intended use case.

`batch.size` : 16384 bytes
Determines the maximum batch size in bytes. Once this limit is reached the batch will be made available to the sender thread for sending to the broker. This configuration can be considered as batching based on size.

Table 2: Configuration Default Value Summary

Configuration	Default Value	Unit
acks	1	N\A
batch.size	16384	bytes
linger.ms	0	ms
max.in.flight.	5	N\A

linger.ms : 0 ms
Determines the maximum amount of time for a batch to fill up in milliseconds. Once the time has expired the batch will be made available to the sender thread for sending to the broker. This configuration can be considered as batching based on time. Note: The batch will be considered ready once either **batch.size** or **linger.ms** is reached.

max.in.flight.requests.per.connection : 5
The maximum number of produce requests that can have outstanding acknowledgements before the sending of more produce requests are halted. This configuration is only relevant when **acks**=1 or **acks**=-1.

3.5 Kafka Use Cases

The original problem Kafka addressed was website activity tracking which demanded high throughput in real-time [7]. Since then, addressing this performance requirement is central to its design. Today, Kafka is serving many purposes beyond its initial problem, for example, as a traditional message broker. This function requires low end-to-end latency and durability guarantees for which Kafka can be configured to serve. Kafka also has comparatively high throughput to alternatives such as ActiveMQ [4] and RabbitMQ [26] and is suitable for large scale applications due to built-in features that welcome horizontal scaling such as partitioning, replication and fault-tolerance [11]. Another use case is collecting the metrics of distributed applications into a centralised feed. Further applications are log aggregation, stream processing and event sourcing.

3.6 Key Terms

3.6.1 Related to Kafka:

- Record: the data that is written to a Kafka topic.
- Topic: the stream of data of a specific type of event and stored in the format of a commit log. Producers write to the topics, and consumers read from it. It can be compared to a table in a database.
- Batch: a collection of records destined to the same topic/partition combination.
- Produce request: the request sent by the Producer to the Broker to write a group of batches to the specified topics.
- Offset: an index of a record in a topic's partition.

- Broker: a node in a Kafka cluster, which contains and maintains the topic partitions.
- Cluster: one or more Kafka Brokers.
- Replication: duplicating partitions among brokers to prevent loss of data in case of failure.
- Leader Broker: leader of one or more topic partitions. The broker contains the topic partition and receives new data destined for this partition from the producer. It also distributes it to other brokers, called followers, who contains the replicas.
- Follower Broker: contains a replica of a topic partition and keeps this in-sync with the leader's original.
- High watermark: the last log offset of which the follower broker is up to date with.

3.6.2 Related to Benchmarking

- Internal measurement: Kafka's built-in measurement of a metric.
- Check measurement: self-made measurement to validate Kafka's measurement of the same metric.
- Throughput: the amount of data units that are sent from the producer to the broker per second.
- Latency: the amount of time it takes from when a message is sent via the producer interface until an acknowledgement is received from the broker by the producer. This definition differs according to `acks` configuration setting.
- Batch-size-avg: the average size of a batch sent from the Producer to the Broker. This is an internal metric provided by Kafka.
- Records-per-request-avg: the average amount of records sent by the Producer to the Broker per produce request.

Table 3: Environment Specifications

Component	**Local Machine: Dell XPS 13 7390**
CPU	Intel(R) Core(TM) i7-10510U
Memory	16GB LPDDR3, 2133 MHz RAM
Storage	M.2-PCIe-NVMe-SSD, 512GB
Network	10 Gbit/s
Operating System	Ubuntu 18.04 LTS
Kafka	2.12-2.4.1
Java	JDK 1.8

4 Experimental Setup and Design

4.1 Outline

In short, the approach is empirical measurements of a running computer system. Kafka is deployed on a local machine, and several metrics are measured while it runs for a fixed period per benchmark. The environment and metrics are specified and discussed in sections 4.2 and 4.3 below. The total amount of time per benchmark is 6 minutes, of which the first 1 minute accounts for the warm-up period and the remaining 5 minutes are used to determine the performance. The motivation for the duration is to account for the variation present in the measurements; this variation is especially evident for some configurations [16] and might be caused by garbage collection pauses [28].

To prevent the experiments from being a 'hacking' exercise, they are carefully planned and structured to answer the research questions in a systematic manner. Therefore the motivations for the specific experiments reflect the motivations of the research questions which can be found in Section 2. This systematic approach of starting with the default configuration to establish the baseline performance and after that changing and testing the influence of individual configurations in isolated experiments is, therefore, an appropriate approach to measuring the performance of a complex distributed system.

4.2 Environment

The benchmarks are performed on a Dell XPS 13 7390 [2] and a list of the hardware and software used to perform the benchmarks are listed in Table 3. It is good to note that the hardware that the experiments are deployed on is not representative of a typical production environment. Although it is not representative, many of the results from the benchmarks can still be generalised. These are discussed in Section 7.1.

4.3 Metrics Used and Ensuring Their Correctness

During these benchmarks, two separate measurements are made of the same metric. Kafka's internal metrics are accessed and recorded, and concurrent measurements of the same metrics are made to verify that Kafka's internal measurements are indeed correct.

Table 4: Error Between Internal vs Check Measurements

Record Size	**10^1B**		**10^5B**	
Metric	**Throughput (MB/s)**	**Latency (Ms)**	**Throughput (MB/s)**	**Latency (Ms)**
Internal measurement	2.87	0.535	336.44	94.841
Check measurement	2.86	0.531	336.27	96.088
error (%)	0.35%	0.75%	0.05%	-1.30%

Here we refer to the Kafka's measurement as 'internal' and the self-made measurements as 'check'. The source code for the program can be found on this repository [12].

4.3.1 Internal Measurements:

Below are the names and descriptions of the built-in metrics that are used and provided by the Kafka Producer:

Measuring Throughput:
'record-send-rate': The number of records sent per second, which is multiplied with the record size to calculate the data rate.

Measuring Latency:
'record-queue-time-avg': The average amount of time in milliseconds a record waits inside Kafka before it has been sent to the broker.
'request-latency-avg': The average amount of time in milliseconds that it takes for the broker to serve a request of the producer.
Note: the two latency measurements are summed to calculate the total latency.

4.3.2 Check Measurements:

Measuring throughput: Counting the number of records sent with a simple counter and multiplying it with the record size to calculate the throughput.

Measuring Latency: The time duration from sending the record until when the confirmation is received from the broker is calculated and recorded. The HdrHistogram library [14] is used for logging these recorded time durations. The library allows for high dynamic range histograms and simplifies the recording and calculation of the measurements.

4.3.3 Error Between Measurements:

To confirm that the measurements are correct, they are collected and compared during default configuration benchmark of Section 5.1. They are determined as accurate based on the fact that the measurements have an error that under is 3%. Table 4 compares the internal and check measurements when a record of 10^1B and 10^5B are sent from the Producer to Kafka. It also lists the error between the two measurements as a percentage. This error is monitored during all of the performed experiments.

5 Micro Benchmarking The Kafka Producer

Key findings:

- Unexpected behaviour when running at the default configuration. In short the throughput worsened and latency improved when the record size increased from 10^3 B to 10^4B. Analysis indicates that it is likely caused by **batch.size**, which is not close to optimal at its default value for these two record sizes (Section 5.1).
- Results indicate significant gains in latency performance for record sizes 10^1B and 10^2B when **acks** is set to 0 instead of the default 1 (Section 5.2.1).
- The best performing **batch.size** for throughput is different for each record size and a good rule of thumb is a record size : **batch.size** ratio of at least 1:100. Increasing **batch.size** from the default can also be beneficial for latency (Section 5.2.2.1).
- Changing **linger.ms** in conjunction with **batch.size** indicates further performance improvements for certain record sizes. Increasing **linger.ms** from the default 0 to 10 improves upon the throughout of all record sizes and surprisingly the latency of most record sizes (Section 5.2.2.2).
- Results indicate that by changing the configuration **max.in.flight.requests.per.connection** from its default of 5 to 1 and 10 does not deliver improvement except for the latency of records of size 10B and 10^2B (Section 5.2.3).

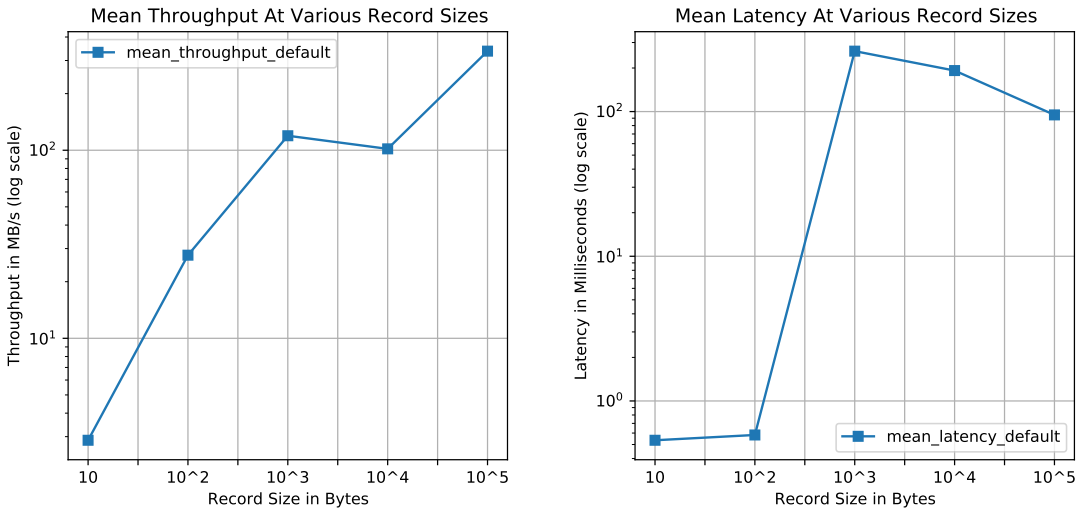
5.1 Default Configuration Benchmark

What: This benchmark explores the performance of the Kafka Producer under its default configurations. Therefore the experiment will be confined to the limitations of Kafka's default configurations [6].

Why: To answer **RQ2**, measure the Kafka Producer's "out-of-the-box" performance on the local system and establish this as the baseline.

Results: Figure 4(a) is the mean throughput, and Figure 4(b) is the mean latency for a range of record sizes. On the y-axis of Figure 4(a) is the mean throughput measured in megabytes per second and on the y-axis of Figure 4(b) is the mean latency measured in milliseconds, both in log scale. For both, the x-axis is the corresponding record size in bytes. The higher the throughput and the lower the latency, the better the performance.

Throughput: A surprising result from this benchmark is that the throughput of record size 10^4B is lower than 10^3B, which bucks the trend of a larger record size having a higher throughput. A possible reason for this is due to the batching process by the Producer. Evidence of this reasoning can be found when looking at the metric "batch-size-avg" and "records-per-request-avg" presented in Table 5. It shows that by having the default **batch.size** of 16384 bytes, a record size of 10^3B on average batches 15.99 records together and send it in one request as opposed to 1 record per batch for record size 10^4B. Thus an average of 16204.96 bytes are sent per request for record size 10^3B versus 10072.0



(a) Default throughput.

(b) Default latency.

Figure 4: Default configuration performance.

Table 5: Comparing Batch Size and Records Per Request

record size (bytes)	**10^3B**	**10^4B**
batch-size-avg (bytes)	16204.96	10072.0
records-per-request-avg (bytes)	15.99	1.0

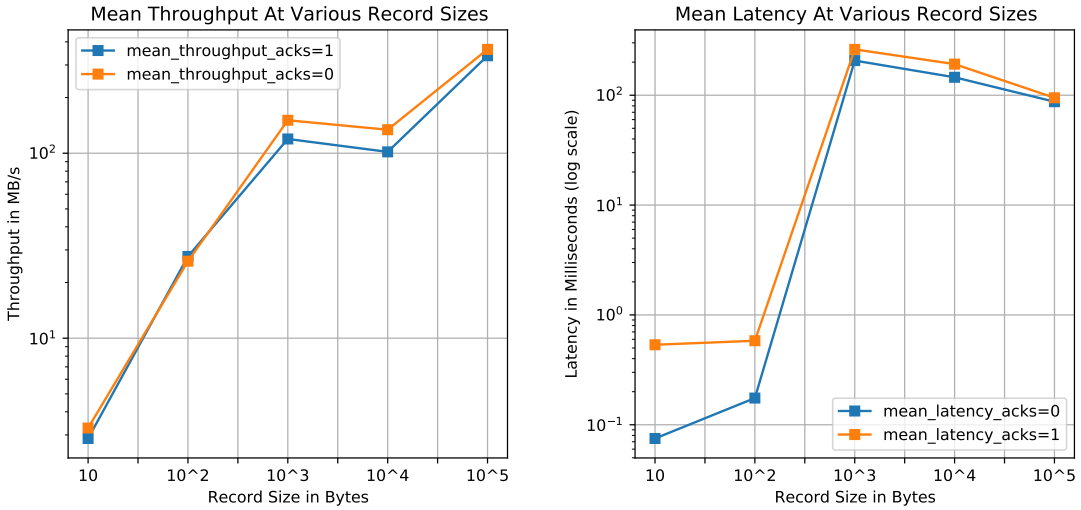
bytes for record size 10^4B.

Latency: An unexpected result is a great increase in mean latency from record size 10^2B at 0.582 to 10^3B at 261.591. Another unexpected result is the decrease in latency from record size 10^3B to 10^4B and again from 10^4B at 192.018 to 10^5B at 94.841. This behaviour of latency decreasing when the record size is increasing is not intuitive, but a reason can be determined when taking into account the throughput analysis above, which proves that there is batching performed with record size 10^3B and minimal batching performed with record size 10^4B. Performing this batching takes time which will add delay and contribute to the total latency. This delay is evident in the data in Table 6 where the record-queue-time-avg of record size 10^3B is at 265.157ms, which is significantly higher than record size 10^4B is at 191.592ms. Note that this table does not contain the average values for the entire benchmark, but the average values at a snapshot during the benchmark.

Summary: Unexpected behaviour was observed, which is a drop off in both throughput and latency from record size 10^3B to 10^4B. When inspecting more granular metrics such as batch-size-avg and records-per-request-avg it is apparent that this is due to the batching of the Kafka producer. Thus **`batch.size`** seems to be a very influential configuration

Table 6: Comparing Queue Time Latency & Request Latency

record size (bytes)	**10^3B**	**10^4B**
record-queue-time-avg (ms)	265.157	191.592
request-latency-avg (ms)	0.638	0.462



(a) Throughput performance.

(b) Latency performance.

Figure 5: acks=1(default) vs acks=0 performance comparison. Note: the colours of the legend are different between the two figures.

concerning the performance of the Producer and needs to be properly investigated. See appendix A Table 14 for raw values.

5.2 Incremental Changes to Default Configuration

5.2.1 Default + `acks`=0 Benchmark

What: This benchmark is identical to 5.1 except for the acknowledgement configuration which is set to `acks`=0 instead of `acks`=1.

Why: To answer **RQ3** and measure the difference in performance with the incremental change to an influential configuration `acks`.

Results: Figure 5(a) is the mean throughput, and Figure 5(b) is the mean latency at different `acks` configurations for a range of record sizes. On the y-axis of Figure 5(a) is the mean throughput measured in megabytes per second and on the y-axis of Figure 5(b) is the mean latency measured in milliseconds, both are log scaled. For both figures, the x-axis is the corresponding record size in bytes. The higher the throughput and the lower the latency, the better the performance. The raw values and percentage difference from the baseline can be found in appendix A.

Table 7: Latency Breakdown At Different Record Sizes and acks Combination

Record size and acks setting	**10^1B & acks=1**	**10^1B & acks=0**	**10^5B & acks=1**	**10^5B & acks=0**
record-queue-time-avg	0.09	0.073	94.014	88.249
request-latency-avg	0.455	0	1.418	0
total latency	0.545	0.073	95.432	88.249
request-latency / total latency %	83.49%	0.00%	1.49%	0.00%

Throughput: Throughput increases from the baseline for all record sizes except 10^2B. This observation of the throughput decreasing is counter-intuitive, because the overhead that is incurred when waiting for an acknowledgement from the server is removed when **`acks`**=0. Thus removing the overhead should intuitively lead to higher throughput.

Latency: The decrease in latency was greater for smaller record sizes than for the larger record sizes: -85.98% for 10^1B compared to -7.93% for 10^5B. This is due to the sender thread not waiting upon the request confirmation from the broker when **`acks`**=0. Out of the total latency the request latency is proportionally much larger for a 10^1B record than a 10^5B record, 83.49% vs 1.49% respectively, therefore when this request latency is removed the 10^1B's total latency is reduced by -85.98% vs -7.93% for 10^5B. This observation is illustrated in Table 7 below and Table 16 in appendix A.

Note: Table 7 does not contain the average values for the entire benchmark, but the average values at a snapshot during the benchmark. Also, in this benchmark, there is a significant difference between the internal measurement and the check measurement for latency. This is due to onCompletion() method behaving differently when **`acks`**=0. Thus only the internal measurements have been used to evaluate this benchmark.

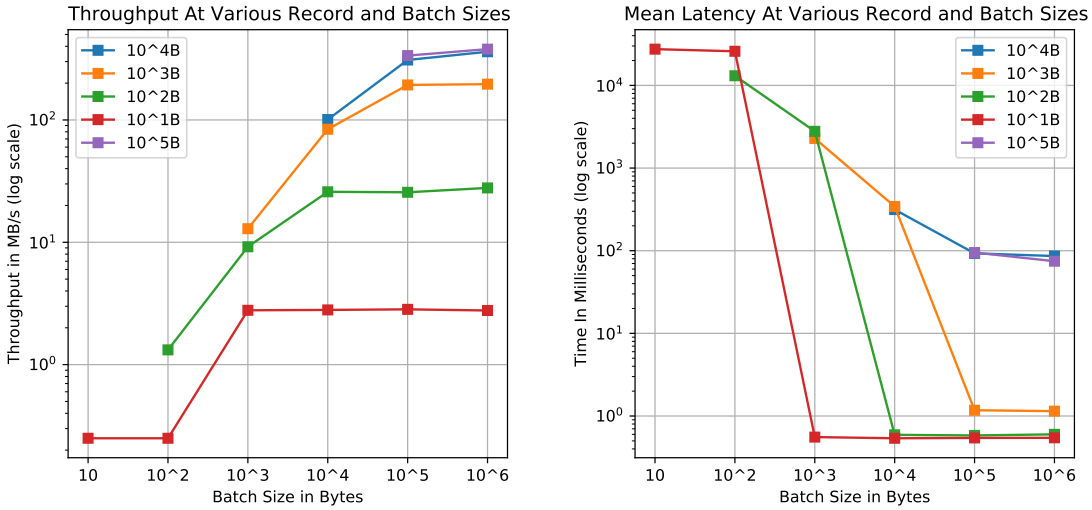
Summary: The shape of the lines closely mirrors the baseline and as expected the throughput is higher and latency lower for **`acks`**=0 than the default **`acks`**=1. For record sizes 10^1B and 10^2B there are significant increases in latency performance, decreasing latency by -85.98% and -69.93% respectively. For all record sizes the increase in throughput performance is relatively small considering the durability sacrificed when **`acks`**=0, which provides no guarantee that the record will be written to the desired topic.

5.2.2 Default + `batch.size` + `linger.ms` Benchmark

In this section, two benchmarks are performed. First, the **`batch.size`** changes to a range of values, while keeping all configurations at their default values. Thereafter the same benchmark is repeated, but **`linger.ms`** is incrementally changed with each iteration of the benchmark. This is to determine **`linger.ms`**'s and **`batch.size`**'s combined influence on performance.

5.2.2.1 `linger.ms`=0 (default)

What: This benchmark measures the default configurations at different values for **`batch.size`** and record sizes.



(a) Throughput performance.

(b) Latency performance.

Figure 6: batch.size configuration performance comparison. Note: the colours of the legend are not ordered.

Why: To answer **RQ3** and determine the influence of **`batch.size`** on the performance of the Kafka Producer.

Results: Figure 6(a) is the mean throughput and Figure 6(b) is the mean latency for a range of record and batch sizes. On the y-axis of Figure 6(a), the mean throughput is measured in megabytes per second and on the y-axis of Figure 6(b) the mean latency is measured in milliseconds, both in log scale. On the x-axis is the corresponding **`batch.size`** in bytes. Each coloured line represents a different record size in bytes and its corresponding throughput or latency given a specific **`batch.size`**.

Throughput: For the record sizes 10^1B, 10^2B and 10^3B selecting a **`batch.size`** of two orders of magnitude larger than itself proves to provide the greatest increase in throughput. Choosing a **`batch.size`** that is two orders of magnitude larger than the record size seems to provide diminishing returns for record size 10^1B, 10^2B and 10^3B. Further analysis is required to determine if the marginal increase in throughput is worth the additional resources such as memory which is incurred when **`batch.size`** is further increased by these orders of magnitude. Also, it is good to notice that the unexpected behaviour that occurred during the default benchmark in 5.1, which was that the record size 10^4B has a smaller throughput than record size of 10^3B did not occur in this benchmark. This point further substantiates the argument made in 5.1, which reasons that **`batch.size`** caused it.

Latency: The latency metric is displaying similar behaviour for record sizes 10^1B, 10^2B and 10^3B, in that, the greatest decrease in latency is at a **`batch.size`** that is two orders of magnitude larger than itself. From there onwards, the latency mostly flattens out with

Table 8: Difference Between the Mean Throughput Performance of Various Record Size/**batch.size** Combinations vs Baseline. **linger.ms**=0 (expressed in %).

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	-91.29	-	-	-	-
	10^2	-91.29	-95.23	-	-	-
	10^3	-3.14	-66.79	-89.19	-	-
	10^4	-2.44	-6.54	-29.69	-0.94	
	10^5	-1.39	-7.37	61.88	204.00	-0.18
	10^6	-3.48	0.58	64.26	255.10	12.72

Table 9: Difference Between the Mean Latency Performance of Various Record Size/**batch.size** Combinations vs Baseline. **linger.ms**=0 (expressed in %)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	5135075.33	-	-	-	-
	10^2	4837371.03	2248192.44	-	-	-
	10^3	3.93	478110.65	772.46	-	-
	10^4	0.56	1.72	31.11	63.96	-
	10^5	1.50	0.00	-99.55	-51.62	0.19
	10^6	1.87	3.09	-99.56	-55.10	-21.29

marginal increases and decreases. It is worth noting that the latency is extremely high when the **batch.size** is equal or one order of magnitude larger than the record size. This observation is valid for record sizes 10^1B, 10^2B and 10^3B and a reason for this have not been found.

Compared to the Default Baseline: In Table 8, the mean throughput measured per **batch.size** and record size combination is compared with the default configuration's throughput per respective record size. The difference vs the baseline is expressed in percentage and improvements upon the baseline are highlighted in green. Table 8 is the same information except comparing latency. The full tables with raw values are included in the appendix B. The values highlighted in green indicates the performance is higher than that of the default configuration.

Tables 8 and 9, therefore, indicates that the range of **batch.size** values and keeping **linger.ms** at 0 provides minimal performance improvements for record sizes 10^1B and 10^2B. In contrast, significant improvements in throughput and latency can be achieved for record sizes 10^3B, 10^4B and 10^5B if **batch.size** is increased. Another interesting observation from these tables are that **batch.size** 10^6B produces the best throughput and latency for record sizes 10^3B, 10^4B and 10^5B. This result is not intuitive due to latency and throughput usually being at odds with each other. These values are marked in brown in tables 8 and 9.

Summary: For record size 10^1B and 10^2B varying the **batch.size** in the benchmark

Table 10: Difference Between the Mean Throughput Performance of Various Record Size/**batch.size** Combinations vs Baseline. **linger.ms**=10 (expressed in %)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	-92.33	-	-	-	-
	10^2	-92.33	-94.51	-	-	-
	10^3	-3.14	-62.13	-87.63	-	-
	10^4	48.43	27.90	-20.44	11.68	
	10^5	43.90	56.09	114.90	240.31	10.74
	10^6	44.95	48.28	125.53	298.62	26.38

Table 11: Difference Between the Mean Latency Performance of Various Record Size/**batch.size** Combinations vs Baseline. **linger.ms**=10 (expressed in %)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	5779158.69	-	-	-	-
	10^2	5386760.75	1944002.41	-	-	-
	10^3	-36.07	419494.33	663.03	-	-
	10^4	206.17	-31.10	15.85	45.40	-
	10^5	1985.05	362.89	-99.63	-56.79	-9.69
	10^6	1992.52	2019.07	-96.98	-60.01	-29.81

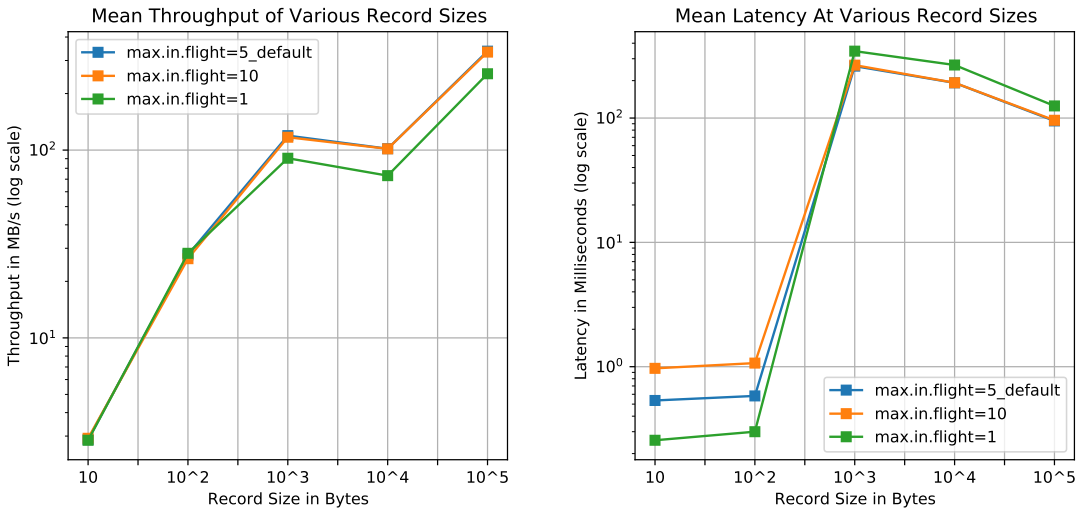
did not significantly improve performance. An interesting result is **batch.size**=10^6B providing the best latency and throughput for record sizes 10^3B and 10^4B. It seems intuitive that you need to increase the performance of one metric at the expense of the other, but both performing better at the same **batch.size** is interesting.

5.2.2.2 linger.ms=10

What: This benchmark is identical to the one performed in 5.2.2.1, with the exception of changing the **linger.ms** configuration from the default value of zero to a range of other values. Note: **linger.ms** was tested at values 5, 10 and 20, but 10 yields the best performance and is thus evaluated.

Why: To answer **RQ3** and determine the influence of **linger.ms** on the performance of the Kafka Producer.

Compared to the Default Baseline: Tables 10 and 11 have identical structures to tables 8 and 9 respectively, but the values are from the benchmark with **linger.ms** set to 10 instead of the default 0 in 8 and 9. Looking at Table 10, as expected this additional delay provided by **linger.ms** increases the throughput for most record size : **batch.size** combinations. This increase was especially evident for record sizes 10^3B, 10^4B and 10^6B in combination with **batch.size** 10^6B. The throughput did not only improve on the baseline from Section 5.1, but also against the performance in 5.2.2.1. This is illustrated by the higher percentages in Table 8 vs Table 10. Thus optimising **batch.size** and



(a) Throughput performance.

(b) Latency performance.

Figure 7: max.in.flight.requests.per.connection performance comparison.

linger.ms together seem to bring additional performance gains for throughput. See appendix C for raw values and % difference. Moreover, as expected, increasing **linger.ms** increases the latency of most record size : **batch.size** combinations. It is interesting to note that the latency also decreases with some of the combinations, this is counter-intuitive as it is expected that the **linger.ms** adds delay to the batching process in the Kafka Producer and subsequently adds to the total latency of sending a record.

Summary: By increasing the **linger.ms** configuration the throughput increased for most record size : **batch.size** combinations. The latency did however improve for some **batch.size** configurations which are counter-intuitive. Further investigation is required to determine its cause. As in Section 5.2.2.1, there are unexpected results were the same record size : **batch.size** combinations provide the best performance for both throughput and latency. In this benchmark, it is record sizes 10^4B and 10^5B providing the best throughput and latency at the same **batch.size** of 10^6B.

5.2.3 Default + max.in.flight.requests.per.connection Benchmark

What: This benchmark is identical to the one performed in 5.1, except for changing the **max.in.flight.requests.per.connection** configuration to 1 and 10 and measuring the throughput and latency of both.

Why: To answer **RQ3** and determine the influence of **max.in.flight.requests.per.connection** on the performance of the Kafka Producer.

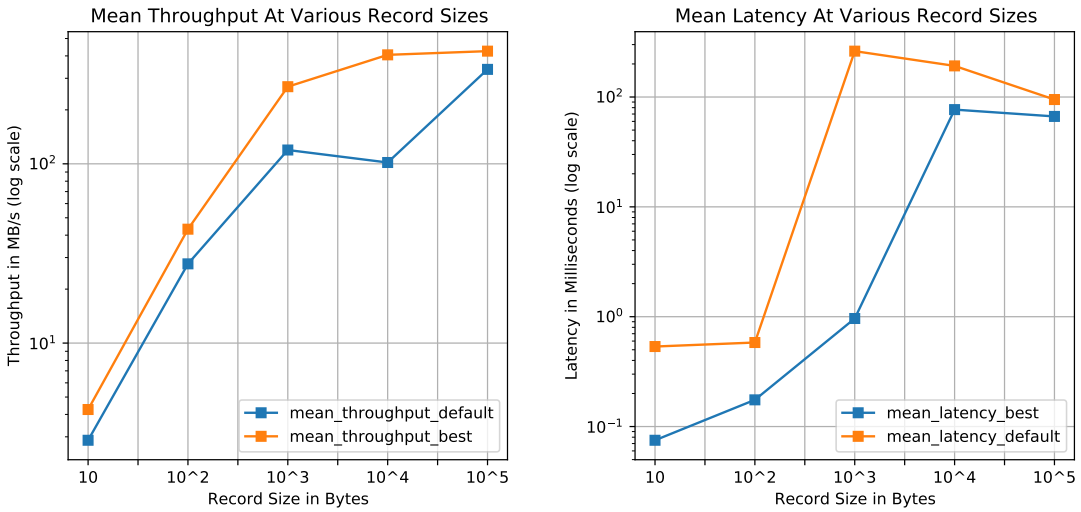
Results: Figure 7(a) illustrates the mean throughput and Figure 7(b) illustrates the mean latency at various **max.in.flight.requests.per.connection** and record sizes combinations. The y-axis represents the corresponding throughput in megabytes per

second for 8(a) and corresponding latency in milliseconds for 8(b). Both y-axes are log scaled and x-axes represents the record size in bytes. The coloured lines represent the various values tested for the **max.in.flight.requests.per.connection** configuration, which is 1 and 10 and is indicated by the legend. These results are also compared to the results obtained in the default benchmark of Section 5.1.

Throughput: Looking at Figure 7(a) it is evident that there is very little difference between the throughput performance of the default value 5 and the measured 10 in this benchmark. Moreover, an expected result is that the throughput performance is generally lower when the **max.in.flight.requests.per.connection** is set at 1 vs the default at 5.

Latency: Changing the **max.in.flight.requests.per.connection** from the default of 5 to 1 resulted in a significant improvement on latency performance for the record sizes 10B, 10^2B. However, it decreases the latency performance for record sizes 10^3B, 10^4B and 10^5B.

Summary: The throughput performance over the different record sizes have mostly declined when the **max.in.flight.requests.per.connection** is changed from its default value of 5 to 1 or 10. There are however significant improvements made in latency performance for record size 10^1B and 10^2B when **max.in.flight.requests.per.connection** is 1. See appendix D for raw values and % difference from baseline.



(a) Throughput performance.

(b) Latency performance.

Figure 8: Best vs default performance

6 Maximising Performance

Aided by the results and insight in the benchmarks of Section 5 this section aims to maximise the performance of the Kafka producer to answer **RQ4**. An analysis is done on each record size's performance, and the configuration values which yielded the best results will be identified and analysed. Figure 8 is the same as Figure 4, but with the added best performance configurations for each respective record size.

6.1 Maximising Throughput

Table 12 is the configurations found in Section 5 that provides the highest throughput per record size. Evidently **`batch.size`** and **`linger.ms`** are the most important configurations for throughput as they were the only two that had to be changed of the four tested configurations. Furthermore, the best performing **`batch.size`** for record sizes 10^1B, 10^2B, 10^3B is three orders of magnitude larger than the record size. For 10^4B and 10^5B the maximum **`batch.size`** is 10^6B and it seems like the throughput is plateauing from record size 10^4B to 10^5B, which is 405.44 to 425.20 megabytes per second respectively and illustrated in Figure 8(b). A possible reason for this is that their **`batch.size`** is relatively small compared to their record size, but the **`batch.size`** has to be limited. If the **`batch.size`** is increased beyond 10^6B then many other configurations have to be increased beyond their default values and this would greatly increase the complexity of the benchmarks. It would, therefore, be interesting to see if the throughput can be increased by increasing these relevant configurations beyond their default values to allow sending batches larger than 10^6B from the Producer to the Broker.

Table 12: Configuration Values Providing the Best Throughput Per Record Size

record size (bytes)	configuration				default MB/s	best MB/s	% increase def. to best
	acks	b.s.	l.ms	m.i.f.			
$\mathbf{10^1}$	default	10^4	10	default	2.87	4.26	48.43
$\mathbf{10^2}$	default	10^5	10	default	27.67	43.19	56.09
$\mathbf{10^3}$	default	10^6	10	default	119.42	269.33	125.53
$\mathbf{10^4}$	default	10^6	10	default	101.71	405.44	298.62
$\mathbf{10^5}$	default	10^6	10	default	336.44	425.2	26.38

Table 13: Configuration Values Providing the Best Latency Per Record Size

record size (bytes)	configuration				default MB/s	best MB/s	% decrease def. to best
	acks	b.s.	l.ms	m.i.f.			
$\mathbf{10^1}$	0	default	default	default	0.535	0.075	-85.98
$\mathbf{10^2}$	0	default	default	default	0.582	0.175	-69.93
$\mathbf{10^3}$	default	10^5	10	default	261.591	0.963	-99.63
$\mathbf{10^4}$	default	10^6	10	default	192.018	76.791	-60.01
$\mathbf{10^5}$	default	10^6	10	default	94.841	66.572	-29.18

6.2 Minimising Latency

Table 13 is the configurations found in Section 5 that provides the lowest latency, per record size. For the lowest latency the best results are caused by a wider variety of configurations. `acks` was the most influential configuration for the smaller record sizes of 10B and 10^2B. For record sizes 10^3B, 10^4B and 10^5B `batch.size` and `linger.ms` were the most influential configurations. A couple of interesting observations are the relatively low latency up until record size 10^3B and how much lower the latency is for record size 10^3B when its configurations are tuned vs its default configurations, a drop from 261.591 ms to 0.963 ms. Also, there is a significant increase in the best latency between record size 10^3B to 10^4B, which is two orders of magnitude although the increase in record size is one.

7 Findings

7.1 Discussion of Results

The analysis in Section 3.3 on the Kafka Producer's internals provides many insights into which processes it performs and how it implements these processes namely batching records together, queuing the batches for the appropriate partitions and coordinating with Brokers. This analysis provided sufficient depth of understanding to reason about most of the observations found in the benchmark results. It especially comes to aid when dealing with unexpected behaviour found in Section 5.1 on default configurations. The two significant unexpected behaviours in this section are the throughput worsening and latency improving as the record size is increasing between 10^3B and 10^4B. This behaviour was resolved with the best configurations in Section 6, this is illustrated by the blue lines in figures 8(a) and (b) which has a trend that is consistently increasing, instead of the increase and decrease by the orange lines. There is still however a decline in latency between 10^4B and 10^5B which cannot be explained. Through the use of the analysis in Section 3.3 and investigating the granular internal metrics provided by Kafka, it becomes apparent that the Producer's batching process is most likely the cause. Control over the batching process is limited, but it can be influenced by the **`batch.size`** and **`linger.ms`** configurations and it must be prioritised when benchmarking for performance improvements.

Furthermore, in Section 5.2 it is apparent that the smaller records gain significant latency performance when **`acks`** is switched to 0 instead of default 1. Although the gains are significant, it must still be determined how it translates to a decrease in end-to-end latency, which is the time duration it takes to send a record from the Producer via the Broker to the Consumer. End-to-end latency might be more important for low latency applications than the isolated Producer latency measured here.

Thorough testing of **`batch.size`** is performed in Section 5.3 to investigate if the reasoning about the extent of this configuration influence on the performance can be substantiated. The results indicate that increasing the **`batch.size`** 2-3 orders of magnitude larger than the record size not only improves throughput but also latency performance. Furthermore, is interesting that latency improves in conjunction with throughput and can also signal that the throughput is not close to optimal for record sizes 10^4B and 10^5B due to the limitation of **`batch.size`** at 10^6B in this study. Motivations for this limitation can be found in Section 6.1. It is worth noting that **`max.in.flight.requests.per.connection`** delivers mixed results and must be changed with caution and careful testing as it can easily decrease performance.

When looking at the relative improvements from the default to the best performance in Section 6, significant improvements in performance can be made by only tuning the configurations **`batch.size`**, **`linger.ms`** and **`acks`**. They are, therefore, a good starting point in the tuning process. This difference in relative performance between default and best configurations is overall higher for latency. In other words, the tuning of the configurations from the default to the best performance shows larger gains in performance for latency compared to throughput measurements. This point is illustrated in Figure 8. Thus the default settings are leaning towards throughput, which ties in with Kafka prioritising high throughput workloads [7].

The high throughput is in large part due to Kafka Producer's batching process, which adds a bit of delay to batch records together. Furthermore, the batching functionality is built-in and cannot be by-passed by setting a configuration. In contrast, true low latency requires the delivery of a record to be attempted immediately after it arrives, and the delay of the batching functionality prevents this from occurring. From this, it is a reasonable assumption that Kafka prioritises throughput over latency. This should be considered when deploying Kafka in a domain where latency is the priority.

To conclude, it is good to note that the testing environment doesn't reflect a typical production environment, but the results are still useful and can be generalised to an extent. For example, the unexpected behaviour during the default benchmark, the degree of influence that the configurations have on performance and the set of configurations that produced the best performance can be generalised. The trends, comparisons and percentage differences between the benchmarks are also useful, although the raw performance values are not. For example, knowing that the best-obtained throughput for a record size of 10^3B on this local machine is 4.26MB/s doesn't have general worth, although knowing which configurations produced this best throughput and the margin it improved by is 48.43% from the baseline does have value.

7.2 Relevant Related Work And Comparing Results With Own Findings

Several studies have made interesting findings on Kafka's performance. Here are a few of them and highlights of important takeaways that are most relevant to the scope of this study. Where appropriate, their results are also compared to that of this paper.

A particularly relevant study was conducted by Hesse et al. [16]. They aimed to find the ingestion limits of Kafka; in other words, how fast one can write to Kafka brokers. The study was however mainly limited to configuring the **acks**, read-in-ram and **batch.size** configurations while measuring the number of incoming messages to the broker. The remainder of configurations were kept at the default levels. A notable finding is that **acks**=0 performed worse at times than **acks**=1, the reason for which was not explained in the paper. A similar benchmark is performed in Section 5.2, which compares the performance of different **acks** configurations. However, results from that benchmark indicate an overall increase in performance in both throughput and latency. Another finding was by keeping **batch.size** at the default of 16384B, the throughput remained very steady over time and when the **batch.size** was increased to 32768B and 65536B there was great inconsistency within the rate of throughput especially with the latter **batch.size**. This greater variability is also observed in benchmarks performed in Section 5.3. When the record size and **batch.size** is set to 10^3B the throughput's variance is 0.00712, compared against when the **batch.size** is set at 10^4B the variance is 0.51890. Therefore on this particular point, the results of the benchmarks agree with the paper's findings.

Wiatr et al. [28] focused on Kafka's consumption of system's resources, specifically examining the effects of increased memory consumption of central processing unit(CPU) usage due to garbage collection(GC) pauses. The amount of memory which Kafka allocates is largely dependent upon the message size, buffer size and **batch.size** parameters. They found that larger buffers increased the latency, moreover larger memory consumption did

increase the GC pauses, which they stated was caused by inefficient memory management of the KafkaProducer. Another finding was that a buffer size of 512KB was optimal for CPU consumption. In Section 5.2.2.1, a similar question was asked, namely if the marginal increase in throughput caused by increasing the `batch.size` configuration is worth the additional delay. This delay is in part due to the additional memory allocation and accompanied garbage collection pauses. The answer depends on the requirements of the application that will be using Kafka and is worthy of further investigation.

Le Noach et al.'s [20] study focused on the impact that the `batch.size` configuration has on the performance of the consumer's and producer's latency and throughput performance. Their main finding in their own words is a major decline in performance that occurs when the number of nodes in the broker cluster is increased; they suspect this is due to internal Kafka synchronisations and bottlenecks in the network. The charts included in the poster indicate a greater throughput for a cluster with a smaller number of nodes as opposed to clusters with a greater number of nodes. This phenomenon was especially evident when the `batch.size` configuration was increased. Increasing the `batch.size` from 30KB to 50KB decreased the throughput of the producers in Kafka clusters with 2 to 6 nodes. Although different broker topologies are not benchmarked in this study, it is interesting to see other studies validating the great influence of the `batch.size` on the performance.

Although the literature on benchmarking Kafka is fairly limited, most of their results agree with those found in this study's benchmarks. These include greater variability in throughput occurring with an increase in `batch.size`, an increase in latency due to increase in memory usage and `batch.size` playing a large role in the performance of Kafka. However, there were conflicting results with `acks`=0 performing worse than the default `acks`=1 in Hesse et al.'s study. Furthermore, the work gave insight into more areas that are deserving of future work, such as benchmarking memory usage and more expansive broker topologies.

7.3 Experience

This thesis was challenging and very rewarding. In the beginning, I was almost completely green to the field of distributed systems, and it was, therefore, a steep learning curve to understand the concepts around distributed systems and why Kafka requires them. Furthermore, apart from creating a measurement to ensure the correctness of Kafka's internal latency measurements, there were no major technical problems which severely hindered the study's progress. It was more an iterative process, with the guidance of the supervisor to improve the efficiency and accuracy of the measurements made in the benchmarks. This includes writing multithreaded code to reduce the overhead incurred when recording the relevant measurements.

7.4 Limitations

Kafka is a distributed system and therefore very complex with many moving components. This makes it especially difficult to measure its performance. Besides, the amount of configurations and combinations of configurations that can be evaluated is vast. Thus only a few can be properly examined considering the time frame of this study. Moreover performing the benchmarks requires a significant amount of time and also the full dedicated use

of the local Dell machine. Thus having a dedicated machine for running the benchmarks and another for writing benchmarks and visualising results will greatly boost productivity when a similar study is made in the future. Also, running the benchmarks on the local machine is not a good representation of a general production environment. Thus it must be considered if the aim is to obtain results that can be generalised.

7.5 Future Work

Following up on the tests performed on the Kafka Producer the equivalent benchmarks can be done on the remaining components of Kafka, namely the Broker and Consumer. This will establish the end-to-end performance and overall influence of the configurations on the Kafka system. Also worthy of further exploration are more production-like cluster topologies, as Kafka is intended to be used as a distributed system and testing it in that domain on realistic use cases would be valuable. Furthermore, more configurations can be investigated, and their influence determined. Lastly creating an automated testing system that tests an array of configuration combinations and provides the best performing combination would be extremely useful. As doing the testing manually can be a tedious process. An equivalent approach can be found in the Machine Learning domain when optimising hyper-parameters, namely the Grid Search process.

8 Conclusion

This study began by obtaining an overview of how Kafka as a system works, understanding its use cases and the major design decisions that influence its performance. Moreover, the Kafka Producer was investigated and analysed at a level that proved necessary in understanding and explaining its behaviour during the latter performed benchmarks. **RQ1** can, therefore, be answered as yes, the Kafka Producer can be understood at a sufficient level to explain and quantify its behaviour. This was valid for the majority of unexpected cases where they can be explained after investigating the finer metrics available and matching it with what is known about its internal processes.

Furthermore, the Kafka Producer's performance was measured at its default configurations to determine its "out-of-the-box" performance. This resulted in interesting and some unexpected behaviour. After some investigation, it quickly became apparent that the `batch.size` configuration is very influential in the behaviour and performance of the Kafka Producer. These results also served as a good baseline to compare the subsequent benchmarks against. **RQ2** was therefore sufficiently answered, knowing what the performance of the default configurations are and thorough investigation was made to explain the reason for the results.

To answer **RQ3**, some configurations deemed of significant influence were further tested. First of them were `acks` that delivered good overall performance improvement and an especially significant reduction in latency for record sizes in the 10B to 100B range. Second were combinations of `batch.size` and `linger.ms`, which is a challenging benchmark to design and perform due to the sheer number of combinations that have to be tested. Results indicate that to obtain optimal throughput the `batch.size` has to be several orders of magnitude larger than the record size. Thirdly `max.in.flight.requests.per.connection` was tested and except for improving the latency of small records, it proved not to be a very influential configuration.

The best results from all the benchmarks are listed together with the configuration combination that produced the respective result. Providing a good indication of the performance limits available in an attempt to answer **RQ4**. Furthermore, the results indicated that a trade-off between latency and throughput don't always have to be made and that both metrics can improve at the same time. Lastly, significant improvements have been made from the baseline performance set by the default configurations. Thus to conclude, it is vital to plan and execute careful testing of the Kafka Producer as the default configurations are not close to its optimal performance. Doing the testing correctly would significantly increase the performance of the service deploying Kafka and decrease the usage of computing resources. This paper provides a few rules of thumb that can aid with this testing process.

References

[1] Individuals using the internet (% of population). *The World Bank*. [Online; accessed May 10, 2020] https://data.worldbank.org/indicator/IT.NET.USER.ZS.

[2] Xps 13 inch 10e generatie 4k-laptop. [Online; accessed July 15, 2020] https://www.dell.com/nl-nl/shop/cty/pdp/spd/xps-13-7390-laptop?view=configurations.

[3] I. Ahmad. How much data is generated every minute? *Social Media Today*, Jun 2018. [Online; accessed May 10, 2020] https://www.socialmediatoday.com/news/how-much-data-is-generated-every-minute-infographic-1/525692/.

[4] Apache Software Foundation. Active mq. [Online; accessed July 3, 2020] ://activemq.apache.org/.

[5] Apache Software Foundation. Apache kafka. [Online; accessed April 25, 2020] https://github.com/apache/kafka.

[6] Apache Software Foundation. Documentation. [Online; accessed May 5, 2020] https://kafka.apache.org/documentation/#producerconfigs.

[7] Apache Software Foundation. Use cases. [Online; accessed July 25, 2020] https://kafka.apache.org/uses.

[8] Apache Software Foundation. Apache software foundation annual report fy2019, 2019. [Online; accessed May 10, 2020] https://files-dist.s3.amazonaws.com/AnnualReports/FY2019+Annual+Report.pdf.

[9] S. Condon. By 2025, nearly 30 percent of data generated will be real-time, idc says. *ZDNet*, Nov 2018. [Online; accessed May 10, 2020] https://www.socialmediatoday.com/news/how-much-data-is-generated-every-minute-infographic-1/525692/.

[10] Confluent. Introduction to kafka. [Online; accessed April 22, 2020] https://docs.confluent.io/current/kafka/introduction.html.

[11] Confluent. Kafka design. [Online; accessed April 10, 2020] https://docs.confluent.io/current/kafka/design.html.

[12] De Vos Meaker. Project code. [Online; accessed August 4, 2020] https://github.com/De-Vos/Kafka_Benchmarks.

[13] Y. Fu. Disaster recovery for multi-region kafka at uber, May 2019. [Online; accessed July 15, 2020] https://www.youtube.com/watch?v=SqusdVrUTWM.

[14] Gill Tene. Hidrhistogram. [Online; accessed May 21, 2020] ://hdrhistogram.org/.

[15] B. Heslop. By 2030, each person will own 15 connected devices — here's what that means for your business and content. *Martech Advisor*, Mar 2019. [Online;

accessed May 10, 2020] https://www.martechadvisor.com/articles/iot/by-2030-each-person-will-own-15-connected-devices-heres-what-\\that-means-for-your-business-and-content/.

[16] G. Hesse, C. Matthies, T. Rabl, and M. Uflacker. How fast can we insert? a performance study of apache kafka. *arxiv.org*, Mar 2020. [Online; accessed April 5, 2020] https://arxiv.org/abs/2003.06452.

[17] J. Kreps. Every company is becoming a software company. Tech blog, Sep 2019. [Online; accessed April 18, 2020] https://www.confluent.io/blog/every-company-is-becoming-software/.

[18] J. Lee and W. Wu. How linkedin customizes apache kafka for 7 trillion messages per day. Tech blog, Oct 2019. [Online; accessed July 15, 2020] https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages.

[19] Markets and Markets. Streaming analytics market, Jun 2020. [Online; accessed Mar 10, 2020] https://www.marketsandmarkets.com/Market-Reports/streaming-analytics-market-64196229.html#:~:text=%5B289%20Pages%20Report%5D%20The%20global,25.2%25%20during%20the%20forecast%20period.

[20] P. L. Noach, A. Costan, and L. Bouge. A performance evaluation of apache kafka in support of big data streaming applications. *2017 IEEE International Conference on Big Data (Big Data)*, 2017.

[21] J. Ousterhout. Always measure one level deeper. *Communications of the ACM*, 61(7):74–83, June 2018.

[22] J. Qin. Producer performance tuning for apache kafka, Jun 2015. [Online; accessed May 21, 2020] https://www.slideshare.net/JiangjieQin/producer-performance-tuning-for-apache-kafka-63147600.

[23] B. Stopford. Is event streaming the new big thing for finance? Tech blog, Oct 2018. [Online; accessed July 15, 2020] https://www.confluent.io/blog/event-streaming-new-big-thing-finance.

[24] Y. Tkachenko. Building scalable and extendable data pipeline for call of duty games. [Online; accessed July 15, 2020] https://www.slideshare.net/ConfluentInc/building-scalable-and-extendable-data-pipeline-for-call-of\\-duty-games-yaroslav-tkachenko-activision-kafka-summit-nyc-\\2019.

[25] N. Viswanathan. Apache kafka — an introduction. Tech blog, Mar 2020. [Online; accessed May 8, 2020] https://medium.com/analytics-vidhya/apache-kafka-an-introduction-beaaaedb6cfa.

[26] VMware. Rabbitmq. [Online; accessed July 3, 2020] ://activemq.apache.org/.

[27] A. Wang. Inca — message tracing and loss detection for streaming data @netflix. Tech blog, Sep 2019. [Online; ac-

cessed July 15, 2020] https://medium.com/@NetflixTechBlog/inca-message-tracing-and-loss-detection-for-streaming-data-\\netflix-de4836fc38c9.

[28] R. Wiatr, R. Słota, and J. Kitowski. Optimising kafka for stream processing in latency sensitive systems. *Procedia Computer Science*, 136:99–108, 2018.

A Performance Comparison: `acks`=0 vs Default

Table 14: Default Performance Per Record Size (`acks`=1). (megabytes per second)

Record Size (bytes)	10^1	10^2	10^3	10^4	10^5
Throughput (MB/s)	2.87	27.67	119.42	101.71	336.44
Latency (ms)	0.535	0.582	261.591	192.018	94.841

Table 15: Default Performance Per Record Size (`acks`=0). (milliseconds)

Record Size (bytes)	10^1	10^2	10^3	10^4	10^5
Throughput (MB/s)	3.27	26.08	150.74	134	364.31
Latency (ms)	0.075	0.175	206.786	145.684	87.317

Table 16: Difference Between the Performance of `acks`=1(default) vs `acks`=0. (%)

Record Size (bytes)	10^1	10^2	10^3	10^4	10^5
Throughput (MB/s)	13.94	-5.75	26.23	31.75	8.28
Latency (ms)	-85.98	-69.93	-20.95	-24.13	-7.93

B Performance Comparison: Range of `batch.size` vs Default

Table 17: Raw Values of Throughput of `batch.size`/Record Size Combinations (megabytes per second)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	0.25	-	-	-	-
	10^2	0.25	1.32	-	-	-
	10^3	2.78	9.19	12.91	-	-
	10^4	2.8	25.86	83.96	100.75	-
	10^5	2.83	25.63	193.32	309.2	335.83
	10^6	2.77	27.83	196.16	361.17	379.23
	default	2.87	27.67	119.42	101.71	336.44

Table 18: Raw Values of Latency of `batch.size` / Record Size Combinations (milliseconds)

		record.size (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	27473.188	-	-	-	-
	10^2	25880.47	13085.062	-	-	-
	10^3	0.556	2783.186	2282.284	-	-
	10^4	0.538	0.592	342.965	314.824	-
	10^5	0.543	0.582	1.174	92.897	95.021
	10^6	0.545	0.6	1.145	86.208	74.654
	default	0.535	0.582	261.591	192.018	94.841

C Performance Comparison: Range of `batch.size` + `linger.ms`=10 vs Default

Table 19: Raw Values of Throughput of `batch.size` / Record Size Combinations and `linger.ms`=10 (megabytes per second)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	0.22	-	-	-	-
	10^2	0.22	1.52	-	-	-
	10^3	2.78	10.48	14.77	-	-
	10^4	4.26	35.39	95.01	113.59	-
	10^5	4.13	43.19	256.63	346.13	372.57
	10^6	4.16	41.03	269.33	405.44	425.2
	default	2.87	27.67	119.42	101.71	336.44

Table 20: Raw Values of Latency of `batch.size` / Record Size Combinations and `linger.ms`=10 (milliseconds)

		record sizes (bytes)				
		10^1	10^2	10^3	10^4	10^5
batch.size (bytes)	10^1	30919.034	-	-	-	-
	10^2	28819.705	11314.676	-	-	-
	10^3	0.342	2442.039	1996.011	-	-
	10^4	1.638	0.401	303.052	279.195	-
	10^5	11.155	2.694	0.963	82.977	85.654
	10^6	11.195	12.333	7.903	76.791	66.572
	default	0.535	0.582	261.591	192.018	94.841

D Performance Comparison: `max.in.flight.connections.per.request` vs Default

Table 21: Raw Values of Throughput of `max.in.flight.connections.per.request` (megabytes per second)

		Record Size (bytes)				
		10^1	10^2	10^3	10^4	10^5
max.in. flight.requests. per.connection	**1**	2.85	28.21	90.5	73.1	254.97
	10	2.92	26.41	117.08	101.55	332.55
	default	2.87	27.67	119.42	101.71	336.44

Table 22: Raw Values of Latency of `max.in.flight.connections.per.request` (milliseconds)

		Record Size (bytes)				
		10^1	10^2	10^3	10^4	10^5
max.in. flight.requests. per.connection	**1**	0.256	0.3	345.363	267.164	125.199
	10	0.97	1.069	266.49	192.304	95.972
	default	0.535	0.582	261.591	192.018	94.841