CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Impact of lossless compression algorithms on in-memory database synchronization

Analysis of the Redis in-memory database

Master's thesis in Computer science and engineering

ION ANDREI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Impact of lossless compression algorithms on in-memory database synchronization

Analysis of the Redis in-memory database

ION ANDREI

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Impact of lossless compression algorithms on in-memory database synchronization
Analysis of the Redis in-memory database
ION ANDREI

Supervisor: Romaric Duvignau
Advisors: Jesper Geertsen Jonsson, Johan RV Andresson, Ericsson AB
Examiner: Philippas Tsigas

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Impact of lossless compression algorithms on in-memory database synchronization
Analysis of the Redis in-memory database
ION ANDREI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In-memory databases have gained popularity in the last decades due to the increased demand for high-speed access to data. Redis is one such database that provides a sub-millisecond response time for incoming requests. These speedups are of particular interest in the telecommunication industry, where 5G technology needs to provide multi-Gbps network speeds. Ericsson is a global leader in 5G network equipment that benefits from in-memory database improvements. When a fault occurs in the system, methods to prevent data loss are needed. One such method is the data replication on a secondary node. Synchronization after a fault in the replica node puts pressure on the primary node to withstand the incoming requests from clients. Requests are buffered until the replica is ready. The buffering can demand a lot of main memory space, and if the system runs out of memory, the synchronization restarts. The novelty of the thesis work focuses on minimizing the impact over main memory size when synchronization between primary and replica nodes is taking place. Practically, adding multiple types of compression over received data in the Redis network layer. We gather performance metrics related to memory size reduction, requests per second, CPU utilization, relative time spent on CPU, and Maximum Main Memory used.

We show that compression over the random data set, an extreme case, does not provide any memory size reduction, and it has a significant negative impact on performance. Another extreme data set is a single character generated multiple times. Intuitively, this data set is highly compressible and provided unrealistic compression ratios of 71.1. Lastly, we showcase the real data set with a 3.582 compression ratio when using the `ZSTD` algorithm; furthermore, the above data-set showed a higher maximum transfer rate when using compression. The maximum transfer rate shows how much bandwidth can the system support when synchronization is ongoing. Given these data sets, we showcase the positive impact of adopting compression in a 5G network.

Keywords: in-memory database, redis, replication, compression, performance, synchronization

Acknowledgements

I thank my supervisor at Chalmers, Romaric Duvignau, for the support, feedback, and long email conversations he had to endure throughout the thesis work. I would also like to express my appreciation to my Examiner, Philippas Tsigas, for his time and support.
I am grateful to the Ericsson community that helped me throughout the work, especially Jesper Geertsen Jonsson, who provided excellent technical input in the most critical moments. I would also like to thank Johan RV Andresson for his guidance and support in enabling me to continue the work.
Lastly, it is most important to show my gratitude to my former Line Manager at Ericsson, Patrik Nyman, who showed kindness and compassion in my most trying times.

Andrei Ion, Gothenburg, July 2022

Contents

List of Figures

List of Tables

1
Introduction

In today's software-as-a-service paradigm, we rely on databases to persistently store data. Redis is an open source in-memory database that provides storage for key-value pairs. It uses primary nodes and multiple replicas to provide resilience [9]. In-memory databases are used to provide fast response time by removing the disk from processing requests. Data stored in main memory can be lost when a fault occurs in the system. Deploying replicating data over multiple servers is a way to mitigate this drawback.

1.1 Context

In today's interconnected world of 5G technology, the databases need to service millions of Requests Per Second (RPS) and have very short response times. The 5G technology needs to provide low latency rates, reaching response times of 1 millisecond. Using Redis as a service for the 5G technology meets these constraints by having sub-millisecond response times [10]. The most relevant example in the 5G context is storing session information. Updates to these sessions occur all the time and they need to be visible as soon as possible. Thus, it is necessary to analyse how Redis works in the telecommunication industry and how performance can be improved and extensively monitored.

Redis is an open-source in-memory key-value storage project that provides extensive tools for memory analysis and clustering options. It is implemented in the C programming language, but it offers a wide variety of tools and interfaces implemented in many languages such as C++, Java, Go [8]. An example of such tool is the `redis-benchmark`. Such tools are used to collect and analyse the data. Moreover, the Redis benchmark tool gives a quick way to simulate how the database reacts to clients applying commands, gathering performance metrics such as the RPS and communication latency.

1.2 Aim

The main goal of the thesis is to provide an extensive analysis of compression algorithms by implementing them into the Redis `Network Layer` and evaluating performance metrics, including but not limited to resources utilisation (maximum memory, processing time, bandwidth), communication latency, and RPS. We hope to provide better performance for in-memory data structures storage by adding streaming compression algorithms, such as the `LZF`, `LZ4`, and `ZSTD`, but also provide a better understanding of a real-life Consistency Protocol implementation. The analysis will provide new insights into Redis performance in the context of an industry-leading 5G technology company. The primary study case of this thesis is a Redis setup containing one Primary server and one replica server. Using two servers, we will simulate different relevant scenarios such as replica restart, bandwidth limitations, and the addition of compression algorithms to provide an extensive and robust test framework.

We answer the following research questions by adding the `LZF`, `LZ4` and `ZSTD` compression algorithms into the main memory data-base (Redis):

- What overall impact do compression algorithms have on the main memory size, and by how much is the memory footprint reduced?
- Adding a compression step to processing requests from clients could have performance degradation on the Redis server. Determine the performance impact by measuring the request per second supported by Redis.
- What is the maximum bandwidth supported by the server when a total synchronization is ongoing?

1.3 Compression for In-Memory Data

Compression has become more and more critical in present technology stack due to increased use of DRAM by applications that need fast response times [18]. Using in-memory compression also provides a smaller memory footprint and thus reduces the manufacturing and operating cost of a business [19]. There is, of course, a performance trade-off between memory reduction and how fast a compression algorithm does the compress and decompress stages. For example, the `LZ4` and `ZSTD` algorithms [3] [14] provide acceleration parameters to choose the right speedup for the needs of the application.

The algorithms used in the thesis project are based on the well known lossless compression algorithms called Ziv-Lempel [24] [25] from 1977 and 1978 (also known as LZ77 and LZ78, respectively). The LZ77 and LZ78 algorithms work based on the same general principle: finding existing patterns in the stream of bytes and replacing those with an offset relative to the current cursor pointer (a pair of distance-length). There is a balancing act when choosing the longest substring length, and in practice, different algorithms choose different strategies for deciding upon the substring length [4].

The Redis server is asynchronous and single-threaded when processing commands, and it mainly benefits from systems with fast CPUs. Having multiple cores does not really bring any benefit to the processing speed of requests, but multi-thread support is available, and the cores are used mainly for blocking I/O tasks [6]. With this in mind, it is essential to evaluate the performance impact of adding compression, as this would be done on the same core that processes incoming requests.

1.4 Overview

The thesis is structured as follows: The Background chapter dives into the knowledge required to understand the thesis and the following chapters. It will also discuss related topics within the in-memory database research area. The implementation chapter will focus on how compression is added to the Redis code and the design choices for the testing tools. The evaluation chapter shows the testing scenarios and the hardware resources used to conduct the experiments. It also brings to light some of the essential Redis configurations changed for our experiments. The Results chapter will show the impact of compression on the Redis server. We present the collected performance metrics, memory and processor impact and discuss the findings. Lastly, a conclusion is formulated in the final chapter with a possibility of future research based on the open questions resulting from our findings.

2 Background

The following chapter describes concepts relevant to understanding the thesis work. The first section brings information about Redis Replication. The second section discusses the `Hiredis API` used in the implementation. Next, the `reply list` functionality is described, and it is essential because the compression algorithms are added here. Flame Graph testing tool is also presented as it is used to measure important CPU performance metrics. Lastly, we dive into related work and how the thesis work fits the current research field.

2.1 Redis Replication

Redis uses asynchronous replication between the primary node and the replicas. It is non-blocking, and the primary node can still serve requests from clients. In short, the primary node forks a child process that handles the replication and synchronization. The child uses copy-on-write to prevent duplicating memory that has not been modified. One advantage of replication is that the data in the primary database is replicated onto another server, instead of keeping the data as a backup on the disk. This does not only makes for a faster handover if the primary process fails, but also provides resilience in the case of hardware or process failure.

When a transient fault such as a timeout occurs between a primary node and its replicas, the Redis logic tries to do a partial synchronization. When this method fails, the replicas have to fully synchronize. This is done by sending ,via sockets, an Redis DataBase (RDB) file from the primary nodes to the replicas. In Redis terminology, this is called "diskless replication".

Redis uses a replication buffer, called `client-output-buffer` that stores the writes performed recently in the primary database while a partial or total synchronization is ongoing.

A failure case can occur when the `client-output-buffer` is full. The limits for different types of clients can be configured via the `client-output-buffer-limit` parameter from the *redis.conf* file. In Redis terminology, the replica that the primary talks to is generally called a `client`, thus the naming convention described above.

Redis has multiple types of clients. For example, clients can be of the following types:

- primary
- replica
- monitor
- multi

When a client is of type *primary*, it means that the primary node is issuing requests to the current node. A *replica* client means that the current primary node servers its replicas requests. The `client-output-buffer-limit` parameter tracks the memory limit for each client type.

If partial synchronization fails, a total synchronization is triggered, and the replica will try and get the complete copy of the primary database. While the total synchronization is ongoing between the primary and the client (i.e. the replica), all the writes performed by other clients in the primary node are buffered. The primary node is waiting for the total sync to finish to send the buffered modifications to the replica.

Redis offers hard and soft output buffers limits for different types of clients via the *client-output-buffer-limit* configuration parameter. If the maximum capacity is reached, the client is disconnected as soon as possible. The maximum capacity can be reached if the produced data (writes) rate is greater than the consumed data (reads).

If a client disconnect occurs while the total sync is ongoing, the primary node restarts the total sync. If the produced data is still greater than the consumed data, the total sync step will loop, and thus the replica will never finish the sync.

Redis relies on the eventual consistency model, where high availability is the main benefit when it comes to replication. As mentioned before, Redis uses primary and replica nodes where clients can write to both primary and replica nodes. The replicas can be configured as read-only. When a Write to the primary node occurs, that write is stored in the backlog and forwarded to the open connection that the primary node has with its replicas.

This approach resembles the Replicated-Write Protocols, more precise the Active Replication Protocol. In the Active Replication Protocol, a client writes to a replica, and that replica propagates the writes to the other replicas. In Redis's case, the primary node propagates the writes to the replicas. Moreover, clients can only connect to primary nodes, not to any replicas as in the Active Replication Protocol.

There is a keep-alive mechanism that checks if there is still a connection between replicas and primary nodes. If there is a timeout on the keep-alive messages, the replicas would have to request a partial synchronization. The primary node keeps a backlog of the most recent modifications of the database. In case of a timeout, this backlog buffer is sent to the replicas to try and fix the out of sync state. However, sometimes a partial sync does not fix the out of sync state. This can occur when the databases did not exchange keep-alive messages for a long period of time, as the replay buffer could be too small to replay all the changes.

When the partial sync does not solve the issue, a total synchronization is needed. The sync mechanism works as follows: The primary node creates a snapshot of what the database looks like at a given time. This snapshot is saved in a file format named RDB (Redis Database Backup file). The RDB file can be configured to use `LZF` compression on the database objects via the `redis.conf` file. The compression is done regardless of the replication type. Next, the primary node opens a connection to the replicas and sends the RDB file. Because this is done while the RDB file is sent, there are still modifications done in the primary database that will be stored in the backlog buffer. The Redis primary node forks a child process called *redis-rdb-to-replicas* that handles the replication. When the sending of the RDB file is finished, the remainder of the backlog messages not captured in the snapshot is sent to the replicas.

As mentioned before, Redis is an in-memory database that provides higher access speeds to data than a regular on-disk database [16]. Moreover, Redis provides built-in replication mechanisms that provide high availability and resilience. The replication mechanism is one of the most complex parts of the Redis database and works by having multiple copies of data from the primary node into replica nodes. When there are no connection errors, the replicas are kept up to date with the primary node updates by receiving a copy of the changes via streams. The received commands are executed on the replicas.

2.2 Hiredis API

Hiredis is a small client for Redis that is implemented in C. It is easy to use, as it has basic function calls that allow the user to interact with a `redis-server` easily. The main functions that one can use are in Listing 2.1. The `redisConnect` function creates a redisContext that will hold the communication to the `redis-server`. In order to send commands to Redis, the `redisCommand` sends the given command to the redisContext. This function is blocking waiting for a reply. The reply structure contains information about the requested command and keeps the information about any error messages. The reply types and more about hiredis are described in more detail in the hiredis repository [7].

```
redisConnect(IP, port)
redisCommand(context, format)
freeReplyObject(reply)
```

Listing 2.1: Hiredis Functions

```
# Commands over string data type
GET key
SET key value
# Command over hash data type
HSET key field value
```

Listing 2.2: Redis Commands Example

2.3 Redis data types

Redis supports multiple data types such as: Strings, Lists, Sets, Hashes, Sorted Sets, Streams etc. Every data type provides a set of commands that manipulate the data. Table 2.1 shows some of the most used data types together with some basic command types that can be applied on these types.

Strings	Lists	Sets	Hashes
GET & MGET	LRANGE	SADD	HGET & HMGET
SET & MSET	LPUSH & RPUSH	SMEMBERS	HSET & HMSET
APPEND	LSET	SINTER	HGETALL

Table 2.1: Redis Data Types and some basic afferent commands.

The most relevant data types for our thesis work are `strings` and `hashes`. Because `strings` are the most basic data type on which other data types are built, it provides accurate measurement for the following analysis. In addition, the thesis work is focusing mostly on commands such as `SET` and `MSET` because they provide larger command sizes compared to `GET` and `MGET`. For example, in Listing 2.2 the `SET` command will have bigger size compared to the `GET` command because the former includes a value parameter that increases the command size. Furthermore, `Hashes` is a data type that maps string fields and values, and it is recommended to be used in the Redis community because they are a versatile data type that can be used to replace other data types. The `HSET` command is used in the thesis work to provide a robust understanding of another data type than the basic one (`string`). By using these command types and data types, the thesis work can provide insights into the differences in results when combining different parameters.

2.4 Reply List

Redis Network layer uses a simple producer/consumer paradigm to send data to clients. As mentioned before, clients can be of multiple types: normal (including monitoring) clients, replica clients, pubsub clients. The producer/consumer mechanism is done in the same thread, asynchronously. The producer side tails in a list the incoming data and the consumer side pops the head from the list. This list is usually allocated to at least 16KB. Figure 2.1 shows that the producer adds node

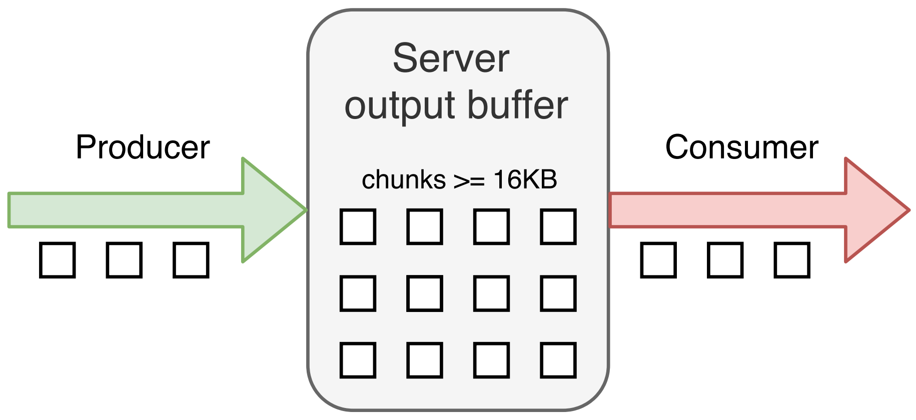

Figure 2.1: Redis Reply List.

elements to the list and checks against the server output-buffer limit.

Interestingly, the zmalloc library used by Redis is interrogated in the producer side to return the exact number of Bytes allocated. This prevents internal fragmentation of the allocator. With other words, while there is space in the allocated memory block, append incoming data to try and fill the block before it is consumed.

2.5 Compression algorithms

The thesis work only considers lossless compression algorithms. Lossless compression is a type of compression that does not modify the original data when it is rebuilt after decompression. Other types of compression are not of interest because the Redis replica should have an exact copy of the primary server.

The `LZF` algorithm is a lightweight lossless compression method integrated into Redis source code but used only for RDB compression. It is simple, as it only has compress and decompress functions.

The `LZ4` algorithm [3] overs tuning options, in contrast to the `LZF` algorithm. It can be configured to accelerate the compression speed, trading CPU time for gains in compression ratio. The `LZ4` compression function can be called using an acceleration parameter that gives about 3% compression speed. The proposed implementation in Redis uses the default acceleration value of 1. `LZ4` also has an API that allows for dictionary compression. This is not pursued in this thesis, as it requires storing the dictionary for the whole lifespan of the database, and it is not data agnostic. According to the benchmarks performed over a series of compression algorithms [3], the `LZ4` compression speed is about the same as the `LZF`. However, a difference can be noticed at the decompression function, `LZ4` being about 4 times faster than `LZF`.

ZStandard [14] is a lossless compression algorithm similar to `gzip` [15] that offers fast decompression speeds and multi-thread support. The algorithm uses frames to refer

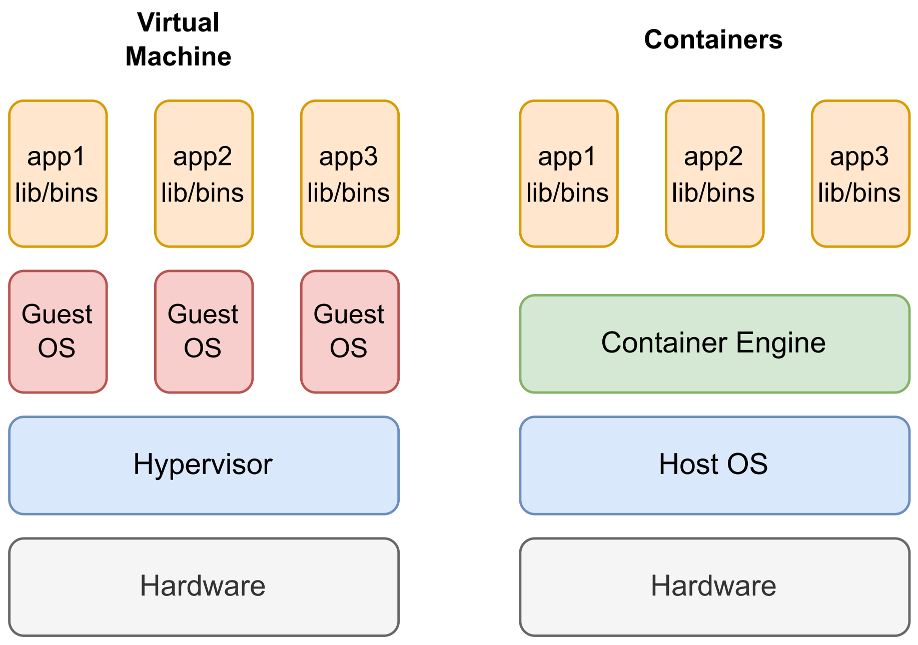

Figure 2.2: Virtual Machine vs Containers.

to compressed data. Each frame is independent and is decompressed individually. Moreover, the frames can be concatenated, and the decompression of a list of frames results in each frame being decompressed and appended next to each other. Same as with `LZ4` compression method, `ZSTD` offers tunable compression levels, trading compression speed for compression ratio.

2.6 Docker and VMs

Docker is a platform that allows the users to deploy applications into a virtual environment called `containers`. The advantage of using containers is that an application can be deployed in a lighter environment compared to virtual machines. Virtual machines need to deploy an entirely new OS to run the demanded applications. To clarify, to run two separate applications, one must deploy two virtual machines with their own operating systems and their own library configurations.

Docker is OS agnostic, as applications can be deployed in a virtual container with all the dependencies met. Multiple containers can be deployed within one host OS. Figure 2.2 shows a high-level difference between containers and virtual machines when it comes to three applications that are deployed with their own specific dependencies and binaries.

2.7 Flame Graphs

Flame Graphs are a performance visualisation tool, invented by Brendan Gregg, that plots the stack traces collected by profilers such as `perf` or `DTrace` [17]. On the x-axis, the stack trace is alphabetically sorted. Sorting choice is important to mention as, when one reads a Flame Graphs, it does not mean that the furthest to the left stack call is executed first. The Flame Graphs do not show the order of execution. The width of the stack call is a good metric for how much CPU time that specific function used. In addition, the tool offers a search option that outputs the percentage of CPU spent time, relative to the other stack calls. The y-axis keeps track of stack depth (e.g. function A called function B).

Comparing different Flame Graphs is a difficult task, in the sense that it can become a tedious task to do by hand. The `FLAMEGRAPHDIFF` tool was created to ease the comparison between Flame Graphs and plot the difference between them [11].

Flame Graphs can also be used to profile memory leaks by tracing the function calls of the allocator. Flame Graphs also offers the option to investigate Off-CPU time [1]. Off-CPU time is the time spent by a process and threads in blocking I/O, paging, waits, and locks. Off-CPU is a relevant metric as this can impact the performance of an application that needs to do On-CPU tasks.

2.8 Related Work

A modification of the Redis source code was proposed in 2019 by Q. Liu, and H Yuan [20] that provides higher performance for the Redis database. The paper shows cache performance improvements by modifying the hashtable used in Redis. The authors also propose a segmentation memory management method that reduces the internal and external fragmentation of the memory (in-memory database). The results shown in the paper mainly focus on the string data structure, while the other data structures, such as hashes and sets, do not show any improvements from the previously mentioned method.

From a security point of view, there is ongoing research in the data recovery area [13]. An attacker might try to delete all the records from the database. The newly published paper by R. Chopade and V. Pachghare [13] presents a method that can recover lost data by modifying the free memory option in the Redis source code. When a record is deleted, memory will not be freed, and the pointers to the records are stored in a linked list. The authors conducted a list of experiments, and the outcome was a data recovery with accuracy ranging from 65% to 95%.

Shansan Chen et al. [12] published a paper on the synchronization topic, that proposes a method called Semi synchronization. Instead of doing a full synchronization that waits for an *OK* message to be sent from the replicas to the primary node, the semi sync is based on the fact that the data is sent via the TCP protocol (which is reliable). Their results showed a performance increase in the number of queries per second compared to the partial synchronization already present in Redis. However, the baseline comparison of this method is Redis version 3.0.3, while at the time of writing the thesis, Redis stable version is at 6.2.0.

An Adaptive Logging method for Distributed In-memory Databases was proposed by Chang Yao et al. [23]. ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) logging is the backbone choice in the paper. The ARIES logging uses the disk to store the logs for the commands. Conversely, the thesis project uses only in-memory storage and focuses on the total recovery time when the primary node buffers incoming commands. Still related to the Log-based recovery, Gang Wu et al. [22] are showing in their paper how different algorithms work on the log compression. They modify the ARIES logging and propose two types of logging: physical and logical. The paper shows that the compression rate is 95.5% (4.5% reduction in size) with the dictionary compression.

3 Implementation

This chapter presents how is compression added in Redis. We also discuss the tools used to analyze our implementation. Moreover, we present the environment setup that simulates two scenarios for extracting metrics. We discuss the performance differences of the testing tool and show how we reduced the execution time of injecting test data to our System Under Test (SUT).

3.1 Adding Compression in Redis

The compression algorithms are added in Redis Network framework. This framework handles the connections and data transfer between the server and its clients. We add compression inside the reply layer. The reply layer is the part that handles data without knowing anything about it. More precisely, data is seen as a series of bytes. The reply layer is an asynchronous single-threaded producer/consumer procedure that stores incoming and outgoing data in a linked list of 16KB nodes. The nodes are allocated accordingly if the incoming data comes in bigger chunks than 16KB.

When a client is in the `wait` state, the reply layer logic waits until the state switches to `online` to send data. The buffered data is then compared against the `client-output-buffer-limit` that closes the producers when the limit is reached (hard limit). The output buffer limit parameter also offers a soft limit option that can be applied to prevent memory usage above a certain threshold. The soft limit parameter looks at the rate with which data is sent for a given amount of time. If the rate is higher than the chosen time and size thresholds, the client is disconnected as soon as possible. The implementation focuses on compressing the incoming data and decompressing it when ready to be sent to the clients. The compression is applied only when the replica is in `waiting` state. The Redis parameter showing this state is called `REPLICA_STATE_WAIT_BGSAVE_END`.
In Listing 3.1 it is described how the compression is done in the `addReplyProtoToList` function. This function is used for all client data communication, and it always tries to append data to the tail of the list. If the tail is full or has no tail node yet, this function requests memory allocation by calling the `zmalloc` function. Compression is only applied when the tail buffer is full. The tail node has two variables that track the actual allocated size and the used size. When these two are equal (lines 10, 11, 12), the node is full, and we can apply compression on the buffer. In most cases, the reallocation will return significantly less memory than before (line 15).

```
1  def addReplyProtoToList(client, data, data_size):
2      tail = listLast(client->reply)
3      if (tail):
4          if (tail->flag == BLOCK_COMPRESSED):
5              goto create_new_node
6  
7          copy_data(tail->buf, data, data_size)
8  
9          if (client->replstate == REPLICA_STATE_WAIT_BGSAVE_END &&
10             tail->size == tail->used &&
11             tail->flag != BLOCK_COMPRESSED):
12             compression_buffer = zmalloc(tail->size)
13             compress(tail->buf, compression_buffer)
14             zrealloc(tail)
15             tail->flag = BLOCK_COMPRESSED
16 create_new_node:
17     if (client->replstate == REPLICA_STATE_WAIT_BGSAVE_END &&
      data_size >= PROTO_REPLY_CHUNK_BYTES):
18         create_new_node(tail)
19         compression_buffer = zmalloc(data_size)
20         compress(tail->buf, compression_buffer)
21         tail->flag = BLOCK_COMPRESSED
22     else:
23         create_new_node(tail)
24         copy_data(tail->buf, data, data_size)
```

Listing 3.1: Compression of data

When the data is bigger than `PROTO_REPLY_CHUNK_SIZE` (currently defined as 16KB), the algorithm allocates a heap memory block to store the compression. The block is a temporary allocation and is freed at a later point. The compressed data is added to the newly created node (lines 18, 19, 20). If the data size is smaller than `PROTO_REPLY_CHUNK_SIZE`, the algorithm creates the tail node and waits for it to be populated with further data.

It is essential to mention that the consumer always tries to grab data from the list. In other words, consuming data immediately after it becomes available. For example, if there is node in the list that is not yet compressed and the client has gone into the online state, then that data will be consumed and sent to the corresponding clients.

The decompression is done in the `writeToClient` function right before it is sent over the socket. Most of the code from 3.2 has been omitted, and just the decompression logic is highlighted. The consuming of the reply buffer is done from the head (lines 4-8). This guarantees that the data will be consumed in the order it arrived (recall that the producer appends data to the tail).

The compression type for the `client-output-buffer` is configuration is added to the `redis.conf` file. The configuration is only relevant for the primary node as

```
def writeToClient(client):
    [...]
    head = listFirst(client->reply)

    if (client->flags & CLIENT_REPLICA && client->flags
   BLOCK_COMPRESSED):
        decompress_buffer = zmalloc(head->size)
        decompress(head->buf, decompressed)
        connWrite(client->connection, decompressed)
    else:
        connWrite(client->connection, head->buf)
    [...]
```

Listing 3.2: Decompression of data

the current implementation does not apply compression algorithms outside the primary node. The `replica-output-buffer-compression` configuration parameter can take the following values: `no` - meaning that there will be no compression applied on the buffer, `lzf` - meaning that the simple `LZF` (existing in Redis source code) compression will be applied to the data inside the buffer, `lz4` - meaning that the `LZ4` standard compression will be applied, and `zstd` - which applies the `ZSTD` compression algorithm to the data inside the buffer. In Listings 3.2 and 3.2, the compress and decompress functions are wrappers that use the different compression algorithms presented before.

Source code for both `ZSTD` and `LZ4` are included in Redis repository. On one hand, `LZ4` comes in the form of `.c, .h` files that is compiled together with the `redis-server` binary. On the other hand, `ZSTD` comes in the form a library that is included in the `dep` folder that includes all external dependencies necessary for the `redis-server` binary compilation. Lastly, `LZF` source code is already present in the Redis repository, but it is used in other parts such as `compression queue` and RDB compression.

3.2 Tuning Redis-benchmark

`Redis-benchmark` is a tool used to test the Redis servers. It connects to a specified Redis instance via an IP and Port using the `TCP` Protocol. The tool offers a wide variety of options for testing the Redis instances. It can test how the Redis server responds to command types including, but not limited to `GET`, `SET`, `LPUSH`, `RPUSH`. Randomly generated data is sent via the commands mentioned above. The benchmark tool creates event loops for each connecting client communicating with the Redis server. The request message sent contains the randomly generated data. The number of requests is configurable and can be specified through the `redis-benchmark` command arguments.

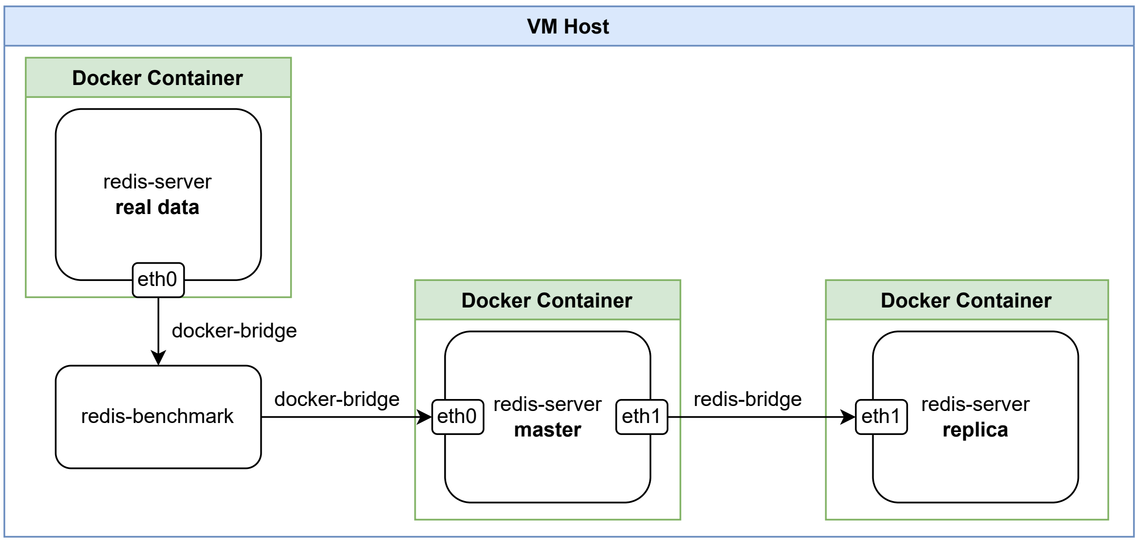

Figure 3.1: Environment Setup.

```
redis-benchmark -h $IP -t set -c 8 -n 2000 -d 1000 --random-data 1
```

Listing 3.3: Redis Benchmark example

As the `redis-benchmark` is open source, we have modified it so that it can satisfy our evaluation and testing needs. For example, the `-random-data` argument has been modified for the purpose of this thesis so that we can generate generate extremely compressible data when set 0 (i.e. 'aaaaaaaa' string). For example, in Listing 3.3 the `redis-benchmark` command creates eight clients that will send 2000 `SET` requests with 1KB of data per request. This will yield about 2MB of data into the specified Redis instance (through *IP* and *Port*) . The tool can output metrics related to the Redis instance that receives the requests. For example, it can output the transfer latency and how may RPS are accepted by the server.

The `redis-benchmark` tool is also modified to fetch a data set from a Redis server instance that contains a loaded `dump` file. The data is injected to the Redis testing environment composed of one primary and one replica. The tool is deployed on the host VM that holds containers for each Redis server deployed. The communication between the Redis server holding the data is done via a different network bridge than the one used in the Redis testing setup. The data is fetched and passed to the primary Redis server node via the `docker-bridge`. The container used to deploy the primary Redis server node is connected to two bridges: the docker-bridge and the `redis-bridge`. It is essential to keep the connections separate, as the `redisbridge` contains traffic control queuing disciplines that could impact test setup. Figure 3.1 shows how the infrastructure for the implementation looks.

Functionality wise, the `redis-benchmark` tool uses the `SCAN` command to parse and find all the data types stored in the database. Each type of data is then fetched and pushed to the primary Redis server node. The data is pushed using commands such as `MSET` and `MGET`. The `MSET` command in Redis adds multiple keys and values via a single command, and by doing this reduces the overhead of sending `SET` commands individually.

command	SET	MSET	tuned SET	tuned MSET
requests	2053920	102696	2053920	102696
exec time (sec)	**91**	**20**	**27.94**	**6.3**
pairs per command	1	20	1	20
RPS	22570.5	5134.8	73511.8	16300.9

Table 3.1: Testing tool Performance Results.

Performance is essential for fetching the data set and feeding it to the testing environment. If the writing speed is limited by the testing tool, then the bottleneck will be the testing tool itself and not the SUT. We believe it is relevant to describe the design choice of the testing environment.

Table 3.1 shows the difference in performance between using the `redis-benchmark` tool and `hiredis` API in testing the Redis deployment. The table only refers to the STRING type in Redis data, which consists of key-value pairs data. The total number of pairs is the same for all the command types described in the table. For references about the data size see Table 3.2.

The first two columns shows the `SET` and `MSET` commands sent through the `hiredis` API. The commands are sent using the `redisCommand()` function. This function blocks until a reply is received from the `redis-server`. In Table 3.2 mentioned above, it is observed that the execution time for the `SET` and `MSET` via the `redisCommand` is significantly slower than when using the `redis-benchmark` tool.

The `redis-benchmark` tool uses event loops to handle writes and reads to and from the `redis-server`. The event loop handlers are writing and reading to the `redis-server`, using very similar commands to the `read` and `write` functions from the `POSIX` standard. This approach is quite fast because the functions work directly with raw buffers that are send through a `redis context`. It can be observed in Table 3.1 that our `redis-benchmark` modification can send more than three times the amount of requests compared to the previous implementation. For example, using our `redis-benchmark` implementation total number of RPS sent is 16300.9, while using the basic `hiredis` API only a total of 5134.8 RPS is sent.

Data Parameters	**Values**
key/value pairs (type string)	2053920
average value size (Bytes)	753
average key size (Bytes)	44

Table 3.2: Data information.

3.3 Redis Configuration

As Redis offers a wide variety of configuration parameters, it is worth describing the parameters chosen for the testing scenarios.

The `save` option enables recovery from a disk snapshot after a restart occurs. We disabled it in both primary and replica nodes because we want to trigger a total synchronization event, which is impossible with the option enabled. Next, the `Rdbcompression` option is used to compress the Redis objects sent between Redis instances over the network. We set `rdbcompression` to `no` because we want more time to do the total synchronization when testing.

Redis also allows disk and socket synchronization configuration. The disk-backed replication works well with partial synchronization, as replication can resume from the last snapshot stored on the disk. However, we have selected to use diskless replication by setting `repl-diskless-sync` to `yes` and thus preventing any partial synchronization. The `repl-diskless-sync-delay` option is set to 0, as this parameter is useful when waiting for more replicas to go into `ONLINE` state in order to feed them the data. In the thesis case, we only use one replica, so thus this is irrelevant.

The `client-output-buffer-limit` for the replica is set to 0, meaning there is no buffer limit. The limitless buffer allows the tests to show the total used memory without forcing the synchronization restart. However, the limits are given the RAM size, but test parameters are selected so that we do not go above the available RAM of the SUT. In the unlikely case of reaching RAM limits, the *Linux Out of Memory killer* will clean the process causing the issue.

We have added a new configuration parameter, `replica-output-buffer-compression`, that enables the user to select which compression type should be applied on the `client-output-buffer`. Currently, the options are: `no`, `lzf`, `lz4`, and `zstd`. Such command was necessary because we can interactively configure compression types, without the need for recompilation or redeployment.

Next, Listing 3.4 presents the essential Redis configuration parameters used for the thesis setup. Moreover, as mentioned before, we have implemented a new parameter to control the compression types without the need to redeploy or recompile the `redis-server` and this is shown below as well.

```
# No supervision (e.g. systemd) or daemonize the Redis process
supervised no

# Process title is the default
proc-title-template "{title} {listen-addr} {server-mode}"

# Save the Redis database to disk is disabled
save ""

# No compression when dumping the database, and LZF compression is
    the only option.
rdbcompression no

# Replicas do not accept writes.
replica-read-only yes

# Use the socket to send data to the replica node directly to the
    process instead of dumping it on disk first.
repl-diskless-sync yes

# Server waiting time before it initiates the sync to replicas.
repl-diskless-sync-delay 0

# Because this option is experimental, it is not enabled in our
    setup.
repl-diskless-load disabled

# Disable the TCP_NODELAY flag on the replica
repl-disable-tcp-nodelay no

# Limits for the buffers that store data in order to be consumed by
     the server
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit replica 0 0 0
client-output-buffer-limit pubsub 32mb 8mb 60

# The client output buffer supports multiple types of compression
    such as 'lzf', 'lz4', and 'zstd'.
replica-output-buffer-compression no
```

Listing 3.4: Contents of the *redis.conf* file.

4

Evaluation

This chapter will discuss the hardware resources and how the tests were conducted. We will present the steps taken for each test scenario and how the `redis-servers` are configured. In addition, we present how the resource utilization metrics are collected together with the tools used for extraction and visualization.

4.1 Environment setup

The experiments are conducted on an Ubuntu 20.04 VM (kernel version 5.4.0-90-generic) having the following specifications: Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz processor (8 cores), 32 GB RAM, 64GB SSD.

The modified Redis code an its configurations are deployed to the Docker containers using Ansible. Ansible uses playbooks to copy files, execute commands and change VM configurations by simply running the playbook from the local machine; furthermore, Ansible allows flexibility in deploying different configurations and automating the testing environment.

Using Docker containers offers options for local traffic shaping, deployment, and monitoring flexibility. The Redis primary and replica nodes are deployed in separate containers and connected via an isolated network bridge. By using a network bridge unique to the two containers, we can apply traffic control rules only on the containers network devices.

4.2 Testing Scenario 1

This test scenario measures how well the compression algorithms work when no buffer limit or client rate limit is applied. During these tests, we extract metrics including, but not limited to, RPS, Online Synchronization Duration, and Used Memory.

- **Setup Containers**. Deploys 3 redis-server instances. Two of the instances are a primary-replica pair with their private network, and the third one is a redis-server that is used by the redis-benchmark to fetch the data set.
- **Populate the Redis DB with random data**. This is done because we want to have data to synchronize between the primary node and the replica after restart.
- **Restart the replica DB**. This will trigger a total bulk sync from the primary Redis instance (as configured).
- **Apply traffic control**. `tc` rules are configured on the network between the primary and the replica to simulate real-life scenarios. The `tc` is needed because we are on the same host, and the network speed is too high.
- **Send requests to Redis primary node**. In this step, all the incoming requests are being appended to the client reply buffer, waiting for the bulk sync from the previous step to finish. Because the `client-output-buffer` limit is removed from the `redis.config` file, we can measure the actual used size of the `client-output-buffer`.
- **primary node syncs the buffered data** The Redis primary node will send the data as a stream to the replica after `wait` state switches to `online`. Every data buffered after the snapshot occurred will be sent. This step also measures the impact decompression has over the total synchronization duration, as the data will be decompressed before being sent to the replica.
- **Cleanup the testing environment**. We flush the data and remove the traffic control rules imposed on the network.

The data extraction from the testing environment is done in a `JSON` format and it has the following design:

```
{
"compression-type": "no",
        "items": [
    {
        "start-test-time": "Sun May 22 22:25:14 CEST 2022",
        "data-type": "real",
        "command-type": "set",
        "rate-limit": "250mbit",
        "bulk-sync-duration": "52",
        "used-mem": "1699272136",
        "online-sync-duration-msec": "11139",
        "page-faults": "4722432",
        "end-test-time": "Sun May 22 22:27:08 CEST 2022",
        "latency-report": {
            "exec_time_sec": "31.04",
            "rps": "66176.50",
            "avg": "0.489",
            "min": "0.176",
            "p50": "0.423",
            "p95": "0.871",
            "p99": "1.175",
            "max": "16.375"
        }
    },
    ...
    ]
},
...
```

The `data-type` in the JSON format can be `real`, `compressible`, or `random`. We test the compression algorithms for each data-type. The `rate-limit` is the maximum rate limit on the network interface between the Primary Redis server and the replica Redis Server. This limit is applied so that the synchronization will not finish too fast and end up in the case where we cannot fully pass all the data to the primary `redis-server`. `Used-mem`, expressed in Bytes, keeps track of how much memory was used by the `client-output-buffer` during the synchronization. The `latency-report` will output a multitude of metrics, from which the most relevant is the RPS. RPS are an important metric for the CPU pressure that the compression ads to the Redis Server.

Every element presented above is incorporated into an `items` list. The `items` list mainly keeps track of the different data types and command types executed for each `compression-type`.

4.3 Testing Scenario 2

The next testing scenario will export the maximum transfer rate supported by the `redis-server`. The problem statement is as follows: For a given output buffer size, find the maximum transfer rate that the server can support. The transfer rate represents the bandwidth supported by the server that has multiple parallel or asynchronously clients doing requests. The output buffer size is tested incrementally starting from 200MB to 1GB with a step of 50MB. The Maximum transfer rate starts from 600Mbps and decrements to 75Mbps in steps of 25Mbps.

- **Setup Containers**. Deploys 3 redis-server instances. Two of the instances are a primary-replica pair with their private network, and the third one is a redis-server that is used by the redis-benchmark to fetch the data set.
- **Populate the Redis DB with random data**. This is done because we want to have data to synchronize between the primary and the replica after restart.
- **Restart the replica DB**. This will trigger a total bulk sync from the Redis primary instance.
- **Apply traffic control** rules on the network between the primary node and the replica to simulate real-life scenarios. The `tc` is needed because we are on the same host, and the network speed is too high.
- **Send requests to the Redis primary node**. All the requests are being appended to the client reply buffer, waiting for the bulk sync from the previous step to finish.
- **Add buffer and rate limits** Add `client-output-buffer-limit` in order to show maximum possible rate limit that the Redis primary server can support during a total synchronization.
- **Cleanup the testing environment**. We flush the data and remove the traffic control rules imposed on the network in this step.

Data extraction format for the second scenario is presented below. For each command type, data set type and command type we extract the `client-rate-limit` together with the `client-output-buffer-limit`.

```
{
        "data-type": "real",
        "command-type": "set",
        "rate-limit": "250Mbps",
        "client-rate-limit": "375Mbps",
        "client-output-buffer-limit": "1050MB",
        "buffer-filled-time": "26"
}

{
        "data-type": "compressible",
        "command-type": "mset",
        "rate-limit": "250Mbps",
        "client-rate-limit": "600Mbps",
        "client-output-buffer-limit": "250MB",
        "used-mem": "38181920",
        "buffer-filled-time": "0"
}
```

4.4 Test framework

Testing is done by deploying Redis servers into docker containers and monitoring them via the `redis-cli` command. Before restarting the replica `redis-server`, we check the status of the ongoing synchronization. The *replica* can be in an ONLINE state but transfer of data can still occur. In order to check if the transfer is finished, we can inspect the primary node offset against the replica offset. The `offset` is a random number that shows the current version of that in the server. Inspection of the offset is done by a simple bash script that inspects the output of the INFO REPLICATION command given in the `redis-cli` tool. The test waits via a while loop until the offsets are the same. When they are in sync, we can proceed to the restart step. The replica restart is done by restarting the Docker container that deployed it.

After the restart, we apply traffic control rules to simulate a more real-life scenario where the bandwidth is not unlimited but fixed. The traffic control is applied using the `tc` command. In Listing 4.1 we present how the rate limit is applied on a Redis instance. The `tc` command applies a token-bucket-filter on the specified network device. The token-bucket filter limits the transfer speed only on the interface between the Redis primary node and its replica. We do not want to apply network limits on any other network devices in our testing as we want to prevent gathering data to be a bottleneck. The setup was described in Figure 3.1.

```
def add_rate_limits():
    docker exec $container_ID tc qdisc add dev eth1 root tbf rate
   $rate_limit burst  $burst_byes limit  $rate_bytes
```

Listing 4.1: Traffic Control Command

While the synchronization is ongoing (after restart), the replica is in `waiting` state. The test tool inspects the current state by parsing the output of the INFO REPLICATION command `redis-cli` output again. The synchronization transfer speed was reduced by using the `tc` command to give us more time to populate the `client-output-buffer` with data.

A time pattern is followed by the Test execution for the no limits buffer scenario. To better understand the upcoming plots, we present approximate time moments (expressed in seconds) for when the test steps are executed. To be precise, these time frames provide a better understanding of resource utilization plots presented in Chapter 5.

From $t = 0$ to about $t = 5$ the Redis nodes are redeployed, and rate limits are set. Continuing from $t = 5$ to about $t = 40$, the SUT is populated with random data to use as dummy data to send when total synchronization is triggered between primary and replica nodes. Starting from $t = 40$ to about $t = 50$ the replica node is restarted, total synchronization starts and the population of different data types is sent to the primary node by a selected few command types. $t = 50$ marks the beginning of data transfer. Some data type and command type combinations could end the execution as fast as the $t = 60$ mark. However, most tests yield different transfer finish times, some ending at the $t = 100$ mark. The total synchronization is running from when the transfer is done to the end of the test. The buffered data is transferred via streams to the replica when the total sync is done, starting from about $t = 120$.

The relevant time interval to evaluate is the data transfer time, from about $t = 40$ to around $t = 100$ (depending on the test parameters and data). The interval mentioned before is the interval that applies compression over the incoming blocks of data from clients.

Extracting Synchronization Time

In order to measure the impact that decompression has over the SUT, we need to extract the time it takes for the nodes to be in `ONLINE` state after a total synchronization event. The test script starts logging the time right after the connection between the primary and replica nodes changed from `WAIT` to `ONLINE` state. The rate limits are removed from container network devices to precisely measure the fastest possible configuration. `Redis-cli` offers the possibility to inspect replication status by using *info replication* command. Below is shown an output example for the command. The `primary_repl_offset` is compared against the connected replica's offset. If the offsets are equal, then the synchronization has finished. The test framework busy-waits for the offsets to be in sync and logs the time difference between the transition to `ONLINE` state (t_1) and the time the offsets are equal (t_2).

```
> redis-cli -a redis -h 172.17.0.2 info replication

# Replication
role:primary
connected_replicas:1
replica0:ip=172.18.0.3,port=6380,state=online,offset=3851898030,lag=0
max_mem_used=27504432
bgsave_close_time=0
primary_failover_state:no-failover
primary_replid:fe3b9669ad6501773655e033474d125e13b0f720
primary_replid2:0000000000000000000000000000000000000000
primary_repl_offset:3851898030
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:3850849455
repl_backlog_histlen:1048576
```

Page Fault Extraction

Page Faults are an important performance metric for Redis DB because being mainly single threaded, requesting memory is slower than processing data on the CPU and could become a bottleneck ending up not utilizing the maximum capacity of the processor. The test framework deploys a `perf stat record` in the container holding the primary Redis node. We extract the page-faults for all threads and processes that the primary node has created. By doing this for all tests, there is no difference between the metric and different compression types could be compared and yield valid results.

When the test execution is completed, `perf` output binary data is converted to text by using the `perf script` command. This file contains some of the following fields: `CPU, THREAD, VAL, TIME, EVENT`. For example, in the case of page-faults, all threads show the page-fault event, but only some of them have the value counter incremented. The test script sums up the page-fault event counters and outputs them into the JSON file that gathers the data for plotting.

4.5 Random and compressible data set

The random data is generated by the `redis-benchmark` tool, and it is created every time a new request is sent. This string is appended to the `SET` command as the value. In order to have different `(key, value)` pairs, the keys are randomized by generating a 12 digit number that is appended to each key. This approach populates the database with different keys containing different random data.

The first option we tried to generate random data was the `rand()` function in C.

This is not really desirable as the `rand()` function is slow and could produce a bottleneck in the benchmark tool. Moreover, using the rand() function to generate a number for the hashing function did not provide a good enough random data set.

The random data set was finally generated by reading random data directly from `/dev/urandom` special file. The data set is created before the execution of the tests because it is not desirable to wait for data generation while trying to test the system.

We kept the random data format as close as possible to the real data set with respect to size, providing a constant in our testing. For example, the size of each request is about 1KB, whereas for the real data set, the average size is about 700Bytes.

Table 4.1 shows the data generation parameters and how tests are conducted. The testing tool used a single thread and 4 asynchronous clients. The clients were sending requests to the `redis-server` while measuring the response time.

Parameters	**Values**
clients	4
requests	1600000
data size	1 KB
MSET size	10
Threads	1

Table 4.1: Test Framework parameters for Random and Compressible data.

Moreover, the random data set is generated before the test starts, so that there is no time spent on data generation when the `redis-benchmark` tool starts sending requests and measuring RPS. The tool waits for a `SIGURS1` to start the testing.

The compressible data is something we have also added to test the performance of the compression algorithms, and it only contains one repeated character. Using this type of data shows how the added compression algorithms behave in an extreme case. The running of the benchmark tool is similar to the random case. However, there is no need to wait for a signal, as one repeated character is fast enough to not impact our testing results. We still append the 12 digit number to the keys in order to prevent duplicates in the database. Key duplicates are not desired because they are not applied to the database and do not help with analyzing the compression ratios.

We explore how the compression works on random and compressible data with different types of Redis commands.

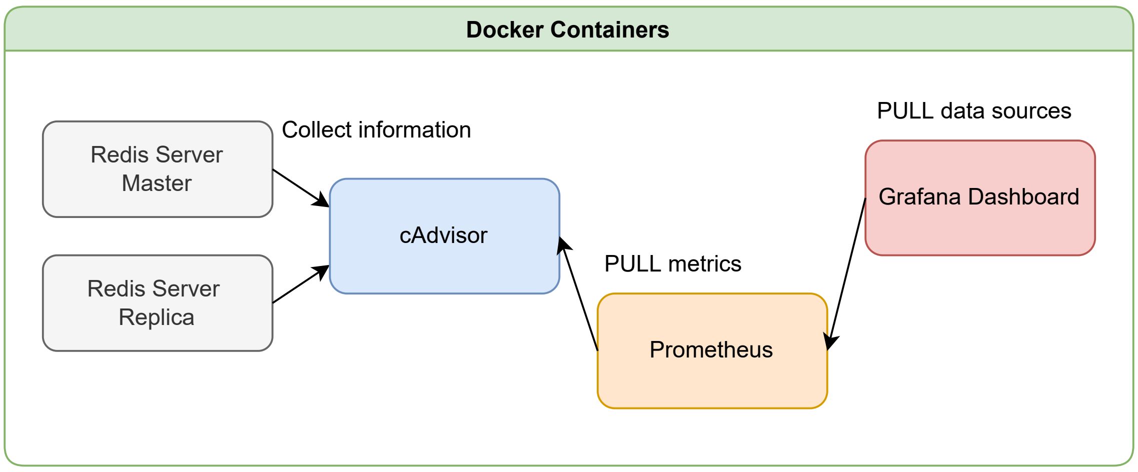

Figure 4.1: Resource monitoring using cAdvisor, Grafana, and Prometheus containers.

4.6 Resource Utilization Metrics Collection

Monitoring of Docker containers is done to extract resource utilization such as: CPU Usage, Memory Usage, Network Bandwidth. These metrics are monitored using cAdvisor (Container Advisor) [2].

A Container with Prometheus Server is deployed to pull metrics from the `redis-server` applications launched in two separate containers. Prometheus [5] is a TSDB (Time Series Data Bases) [21] that enables a HTTP port to allow queries by applications. More precisely, metrics are extracted by Grafana using PromQL (Prometheus Query Language). Figure 4.1 shows how the resource monitoring is setup in the thesis project.

The CPU metric represents the total amount of time spend in the user space by the Container holding the primary Redis Server node. The Prometheus counter `container_cpu_user_seconds_total` is used together with `rate()` function to form an average CPU usage over the last 5 seconds. The memory usage is extracted through the `container_memory_usage_bytes` counter. As the name suggests, it extracts the memory usage for the container. In our case, every Redis node is within a separate container. By using these counters we can collect metrics for each Redis node. Listing 4.2 shows an example of how these queries look like.

The last metric presented in 4.2 extracts the maximum CPU usage at both User Space and Kernel Space levels, for each CPU. After this, only the CPU that had the highest usage is chosen. Finally, all absent data points are replaced with zeros. This is a relevant metric to showcase because the Redis server utilizes only one core (the same core) to process commands and to apply compression on data.

```
> rate(container_cpu_user_seconds_total{image!=""}[5s]) * 100
> container_memory_usage_bytes{image!=""}
> irate(container_network_transmit_bytes_total{image!=""}[1m])
> irate(container_network_receive_bytes_total{image!=""}[1m])

> max(rate(container_cpu_usage_seconds_total{name="redis_6379",
   image!=""}[5s])) * 100 or absent(rate(
   container_cpu_usage_seconds_total{name="redis_6379",image!=""}[5
   s])) * 0
```

Listing 4.2: Prometheus Queries

5

Results

In this chapter we present the findings of the thesis work. We show how the memory is reduced by about 50% when we use the real data set. We present a comparison of the CPU usage in case of no compression and in the case of `LZF`, `LZ4`, `ZSTD` compression algorithms. Throughout this chapter, we will refer to the baseline case as the no compression case. Next, we present the RPS and Online sync duration as an average over 20 executions, together with the corresponding standard deviation. Lastly, the results for memory impact section is presented in MB sizes, while the other following sections presents results in percentages relative to baseline.

5.1 Memory impact and Compression results

We see a significant reduction in memory footprint of the `client-output-buffer` when compression is used. The Compression ratio is presented in Table 5.1. The random data set is not compressible, thus the ratio is 1.0 for all command types and compression types. The Real data set has a compression ratio of about 2.0. Memory footprint of the buffer is not significantly different when comparing the `LZF` and `LZ4` algorithms; however, `ZSTD` outperformed the previously mentioned algorithms in terms of compression ratio. The results for memory usage of the compression algorithms is not averaged over the course of multiple test execution because the compression size is always the same. The performance impact of these algorithms is discussed later in this chapter.

Random data set

Figure 5.1 a) shows that the tested compression algorithms do not compress random data. As an implementation choice, data compression is skipped if the compressed data result is bigger than the decompressed size (which can happen with lossless compression over random data). Data shows no difference between command types regarding the buffer size.

The algorithm tries to apply compression to all incoming chunks of data. In the case of `SET` and `HSET` command, the chunk size is 1KB and in the case of `MSET` command, the data chunk size of 10KB. Next, memory is allocated for the upcoming compression of received data. All compression algorithms tested return an error or a pointer

SET			
Compression	Random data set	Compressible data set	Real data set
LZF	1.0	30.868	2.141
LZ4	1.0	46.044	2.185
ZSTD	1.0	73.163	3.582

MSET			
Compression	Random data set	Compressible data set	Real data set
LZF	1.0	29.992	2.128
LZ4	1.0	43.425	2.173
ZSTD	1.0	71.173	3.555

HSET			
Compression	Random data set	Compressible data set	Real data set
LZF	1.0	30.868	N/A
LZ4	1.0	43.418	N/A
ZSTD	1.0	61.998	N/A

Table 5.1: Compression ratios for each compression algorithm on three different data sets.

to a compressed size bigger than the decompressed size due to the random nature of the data set. Next, as compression was unsuccessful, the temporary allocated buffer is freed, keeping only the uncompressed buffer. Lastly, to keep track of how much memory is used by the `client-output-buffer`, we increment the size counter.

Compressible data set

The compressible data set has a compression ratio of about 73.0 when using the `ZSTD` algorithm when using the `SET` command. Figure 5.1 b) shows significant differences in compression ratio between the no compression case and all other compression algorithms. However, the compression data set is an extreme case, and the experiments are conducted to showcase the compression algorithms' potential. A comparison between the real data set and the compressible data set is of interest to better understand the impact of data types over compression.

A closer look at the differences between compression algorithms is presented in Figure 5.2. Looking at the `SET` command, it is observed that the buffer size is reduced to 54MB when using the `LZF` algorithm, 36MB for the `LZ4` algorithm, and 22MB when using `ZSTD` compression.

We observe a slight difference in compression ratios when comparing the command types. For example, in the case of the `ZSTD` algorithm, `SET` command used 22.9MB, `MSET` command used 23.2MB, and `HSET` command used 27.4MB. The way command types are constructed yields negligible differences when comparing small data sizes. For example, when comparing `SET` and `MSET` commands, the first one sends more data corresponding to command construction. In detail, data such as the command

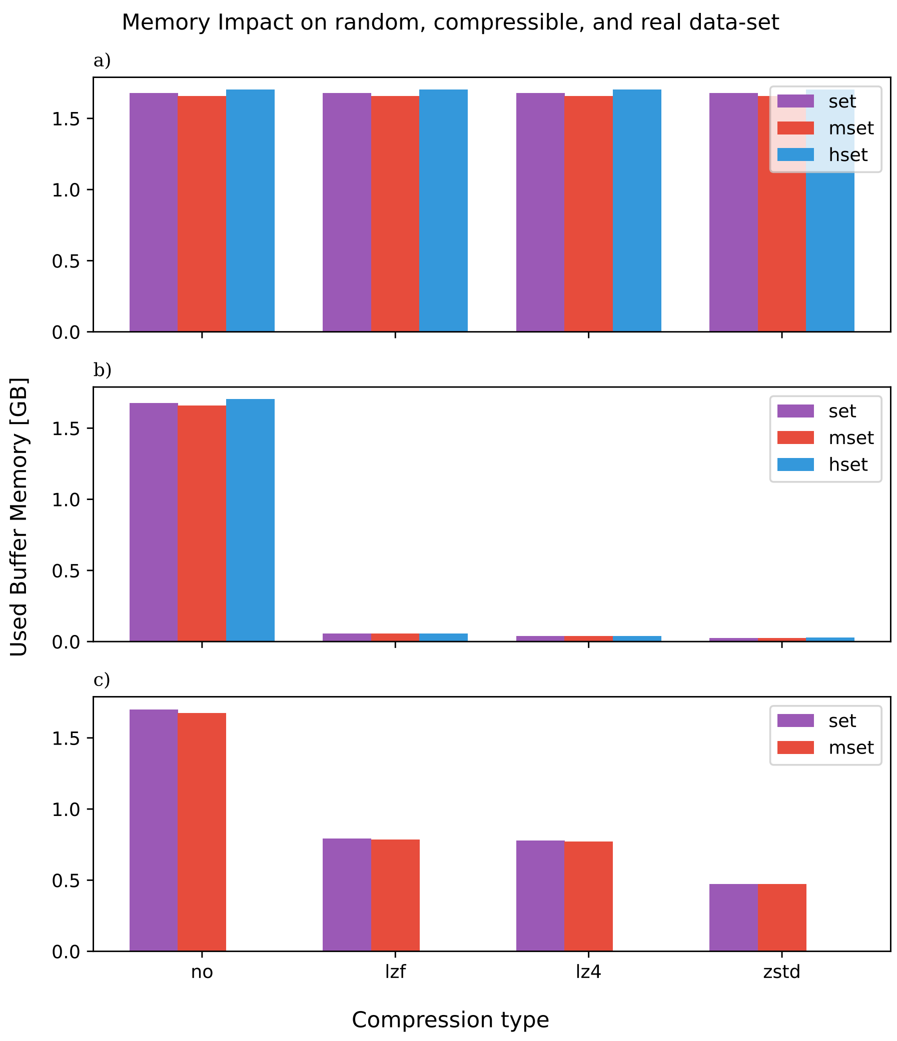

Figure 5.1: Compression results on all data sets. Figure a) shows impact on Random data, b) shows memory impact on compressible data, and c) shows impact on real data.

string *SET* repeats and can be better compressed. Conversely, the `MSET` is a bulk version that replaces ten commands, and the data from the command construction is smaller compared to sending ten `SET` commands. The difference in command data is revealed in the slight difference in compression ratio.

Real data set

The real data set compression performance is shown in Figure 5.1 c). The `ZSTD` algorithm achieved the best results regarding the used buffer size. The data size is reduced from about 1.6GB to 474MB when using the `SET` command and to 470MB when using the `MSET` command. The small difference in compression of about 4MB is because `MSET` does ten `SET` in bulk, thus omitting the string command strings used in the *SET* command.

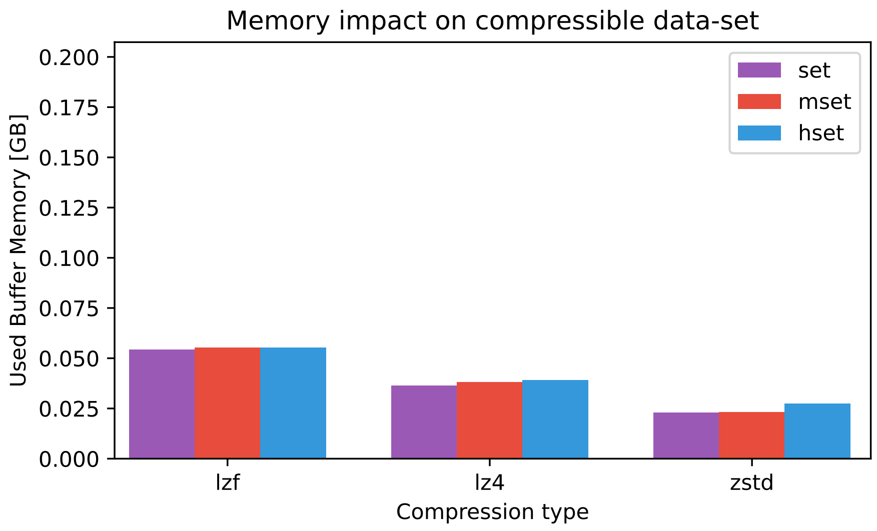

Figure 5.2: Results on compressible data zoomed.

The second best result is achieved using the `LZ4` algorithm, having reduced the data size from 1.6GB to about 777MB. However, comparing the `LZF` algorithm with `LZ4`, there was no big difference in compression ratio, the former achieving a compressed buffer size of about 793MB. Lastly, the same small compression difference, related to `SET` and `MSET` command types (present in the `ZSTD` case) is observed with `LZ4` compression.

5.2 Requests per Second

This section discusses the maximum RPS that the Redis Server supports, given different data types, compression algorithms, and command types. This metric is important because it shows the impact of compression on performance. Intuitively, the performance when adding compression should be lower than without compression, as the Redis server will spend more CPU time compressing incoming data. Compression adds one more step to the server logic after parsing the commands because the Redis server process commands in a single-threaded mode. Figure 5.3 shows the RPS on all data sets.

The RPS metric only analyzes the performance impact of compression functions because decompression takes place at a later stage in the `redis-server`. In the test scenario, the clients send requests to the primary node while the connection between the primary and replica nodes is in `wait` state. The total synchronization between the nodes finishes when the connection switches from `wait` to `online` state. Decompression is applied right after the state transition when data is sent via streams to the replica node. The performance impact of decompression is analyzed in Section 5.3, which looks at the time it takes for the nodes to be synchronized based on *offset* values.

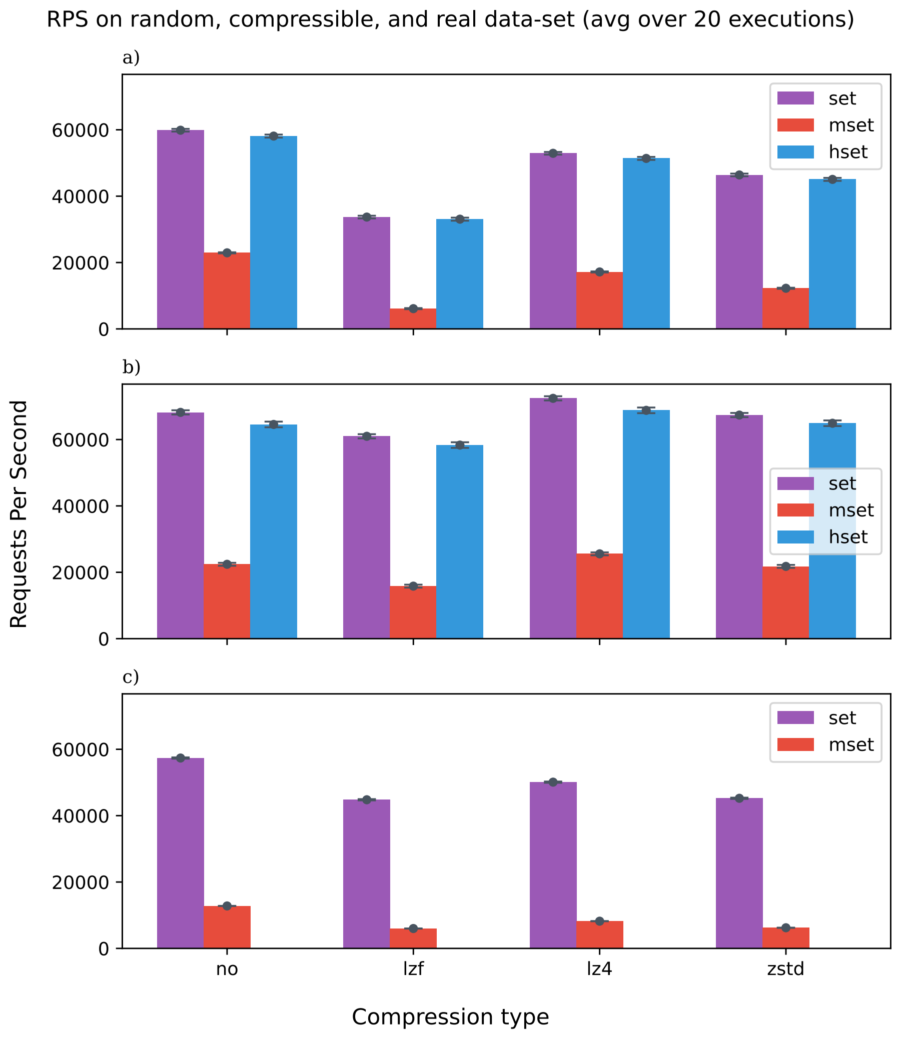

Figure 5.3: RPS on random, compressible, and real data set and the standard deviation of the results.

Random data set

Figure 5.3 a) shows the results for the RPS metric on random data set given different command types such as `SET`, `MSET`, and `HSET`.

The data over the `LZF` algorithm shows a 43% performance degradation of RPS compared to the baseline when testing the `SET` command. Furthermore, `LZF` performed the worst on the `MSET` command, compared to all other tested compression algorithms, having an RPS degradation of about 73%. One reason behind poorer performance when testing the `MSET` command is that data is ten times bigger than when testing the `SET` command, reaching 10KB per chunk. This chunk is received by the `redis-server`, and it tries to apply compression, but because the data chunk is bigger than when using `SET` commands, the compression takes a longer time, preventing the `redis-server` from processing incoming requests. Data shows that `LZF` compression has the biggest performance degradation of the RPS metric compared to all the other compression algorithms.

The `ZSTD` compression algorithm tests, show better performance compared to `LZF`. Further more, the number of RPS processed by the `redis-server` is reduced with about 22%, when testing the `SET` command, compared to baseline case.

Best RPS results, are achieved when using the `LZ4` compression. When comparing the results against the baseline, data shows a 11.62% and 11.58% RPS reduction when using the `SET` and `HSET` command, respectively. `LZ4` performance is about four times better than the one of the `LZF` compression and about two times better performance compared to resulted RPS when using `ZSTD` compression. Overall, the data over `HSET` command shows a better performance by about 1.92% compared to `SET` command. This small percentage difference between the previously mentioned commands shows no significant difference between them when testing the random data set.

As an overall trend on the random data set results, using `MSET` command ended in having the highest RPS reduction when comparing all command types and compression algorithms. This trend is because the data chunks received by the Redis Server are bigger and require more CPU time to apply the compression, thus preventing other tasks from executing. In addition, over the course of our testing of the `MSET` command, data shows around 73% RPS reduction when using `LZF`, 46% RPS reduction when using `ZSTD`, and approximately 25% RPS reduction when `LZ4` when compared against the baseline. Lastly, analyzing only the `MSET` command tests for all compression algorithms, the best RPS values are achieved with `LZ4` compression (only 25% degradation).

Compressible data set

Figure 5.3 b) shows the RPS results of the compressible data set given different the command types. The outcome of using compression over this data set is that the `client-output-buffer` of the Primary Redis server is significantly smaller; moreover, the result is confirmed by analyzing compression ratios from Section 5.1.

The results of testing the `LZF` algorithm have about 10% reduction in performance compared to the baseline, using the `SET` command. When using the `MSET`, the performance degradation was around 29%. In the latter case, the explanation for performance degradation is the significant increase in data size of each command (ten times bigger than with `SET`). As mentioned above, the chunk of data needs more processing time in the `redis-server` process.

Comparing the `ZSTD` compression with the baseline, we notice no big differences in

RPS performance. For example, when testing the `SET` command, the performance impact was only 1.19% compared to the baseline. Results of `MSET` command show a 2.81% RPS performance improvement, while `HSET` command suffered 2% degradation when compared to baseline RPS.

Real data set

Figure 5.3 c) shows the RPS results of testing the compression algorithms on real data set given different only two commands types: `SET` and `MSET`.

The data shows a reduction in RPS of about 13% when using `LZ4` compression over the `SET` command and also around 37% performance impact with `MSET` command. `LZ4` compression provided the best results among all the compression types. Conversely, the `LZF` test results show the biggest performance degradation compared to baseline, resulting in 21% reduction of RPS when using `SET` command and 52% reduction of RPS when using `MSET`. Lastly, the `ZSTD` compression results of RPS are by about 20% lower on `SET` command, while using `MSET`, RPS is reduced with around 52% when compared against the baseline.

5.3 Synchronization Duration

Decompression could potentially affect the synchronization duration between the Redis primary and replica servers. Therefore, we chose to analyze synchronization duration with no limits over bandwidth. When the primary node and the replica connection change from `wait` to `ONLINE` state, the Redis primary node decompresses the buffered data, stored in the reply list afferent to the replica, and sends it as a stream. In this section, we present our findings by analyzing the three data sets. Figure 5.4 shows differences in synchronization when using different compression algorithms.

Random data set

Synchronization duration is quite similar for all compression types when the `SET` command is used. The longest sync duration result is when using `LZ4` compression, lasting 11.450 seconds. The highest difference in sync duration between all compression types is only 0.432 seconds, resulting from the comparison between the `LZ4` compression case and `ZSTD` compression case. Moreover, data shows 0.216 seconds difference in sync duration when comparing `LZ4` compression to the baseline, in the testing of the `MSET` command. Moreover, there is a difference of 0.195 between `LZF` compression (11.326 seconds) and `LZ4` compression (11.521 seconds) when tests are executed using `HSET` command.

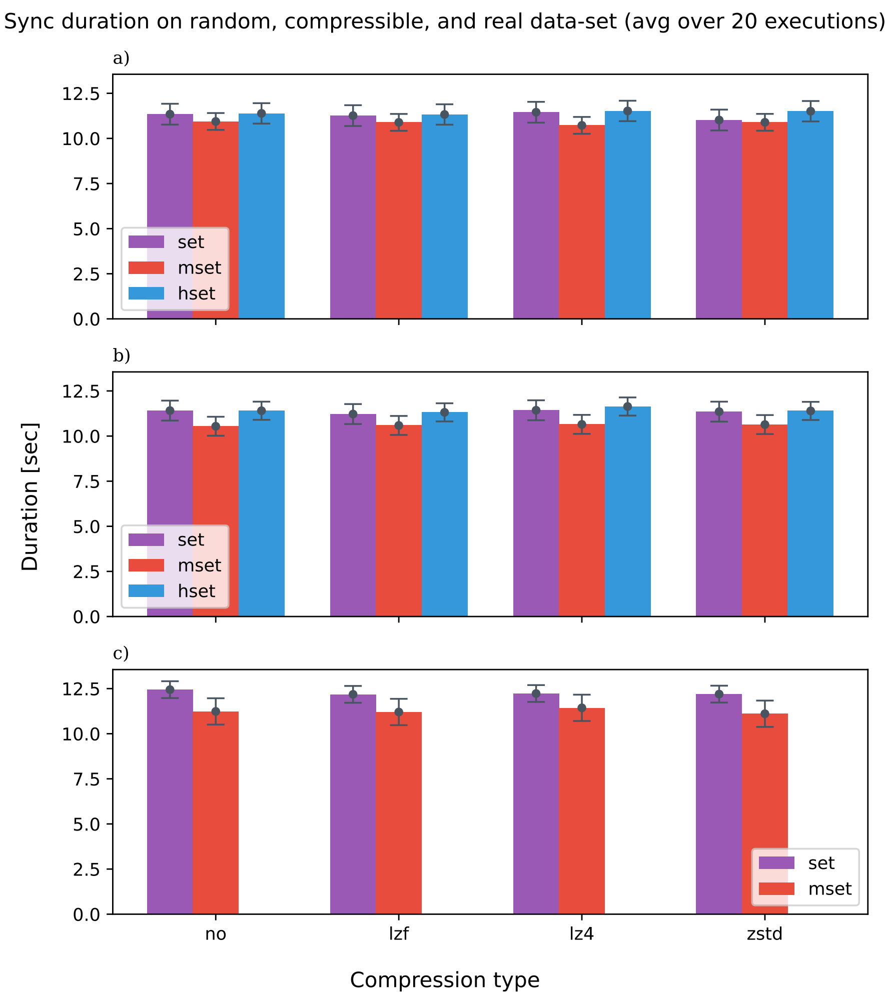

Figure 5.4: Sync duration when primary and replica nodes are in online state, including the standard deviation of results.

Compressible data set

The biggest difference between sync time is 0.324 seconds, by comparing tests over theHSET command for LZ4 compression (11.638 seconds) and LZF compression (11.314 seconds). In this particular case, the sync finished faster when using LZF compression. When MET command is tested, data shows the smallest difference between sync times for all compression types, having a maximum difference of 0.103 seconds (less than 1% increase in sync duration from the slowest case).

Real data set

The data shows a difference of 2.11% in synchronization time between the maximum and minimum duration. These results are achieved when SET command is used. What is more, the maximum sync duration results from the case when no compression is used and by testing the SET command (12.442 seconds). As for the minimum sync duration, is achieved when the ZSTD algorithm is used over the testing on MSET commands (11.106 seconds).

Duration diff over min/max time			
Command type	Random	Compressible	Real
SET	3.77%	1.84%	2.11%
MSET	1.97%	0.97%	2.86%
HSET	1.69%	2.78%	N/A

Table 5.2: Difference between maximum and minimum sync time duration concerning command types and data set.

5.4 Find the maximum transfer rate

This section shows the different results for each compression type in relation to the maximum transfer rate supported by the Redis server. The transfer rate refers to the possible maximum transfer data guaranteeing that the `client-output-buffer` will not fill. If the buffer fills, the connection with the client will be closed as soon as possible along with a restart of the synchronization task between the primary node and its replica. Moreover, if the incoming changes to the primary node keep the same transfer rate as before the synchronization restart, then synchronization restart loops. Looping is not a desirable state, and thus it is relevant to show what would be the maximum transfer rate supported by the Redis Server. Figure 5.5 shows the maximum transfer rate and its afferent `client-output-buffer-size`.

Before discussing the results, we present some clarifications to guide the reader when analyzing the figures in this section. The plots show the maximum transfer rate achieved when the output buffer size varies. Extracted data points on the x-axis start from $x = 75$ (Mbps), the smallest transfer rate tested in our system. The lower bound of the transfer rate (not present in the figure) shows that the maximum transfer rate can be much higher with the given output buffer size and represents a limitation of the data sets and our testing scenarios. For example, with a buffer size of 550MB the max transfer rate is around 200Mbps. Even though it is not visible in the plots, tests are conducted till the 600Mbps Maximum Transfer Rate. All data sets are of limited size, and if compression can provide a smaller size than the Output Buffer Size for a given test, then that test will be able to support the 600Mbps transfer rate. We chose only to show the points that have been tested with compressed data size smaller than the Output Buffer Size.

For clarity, we only present the maximum transfer rate results only for the data that looks at the SET command type usage. Results for MSET and HSET command types can be found in Appendix A, Figure A.7 and Figure A.6.

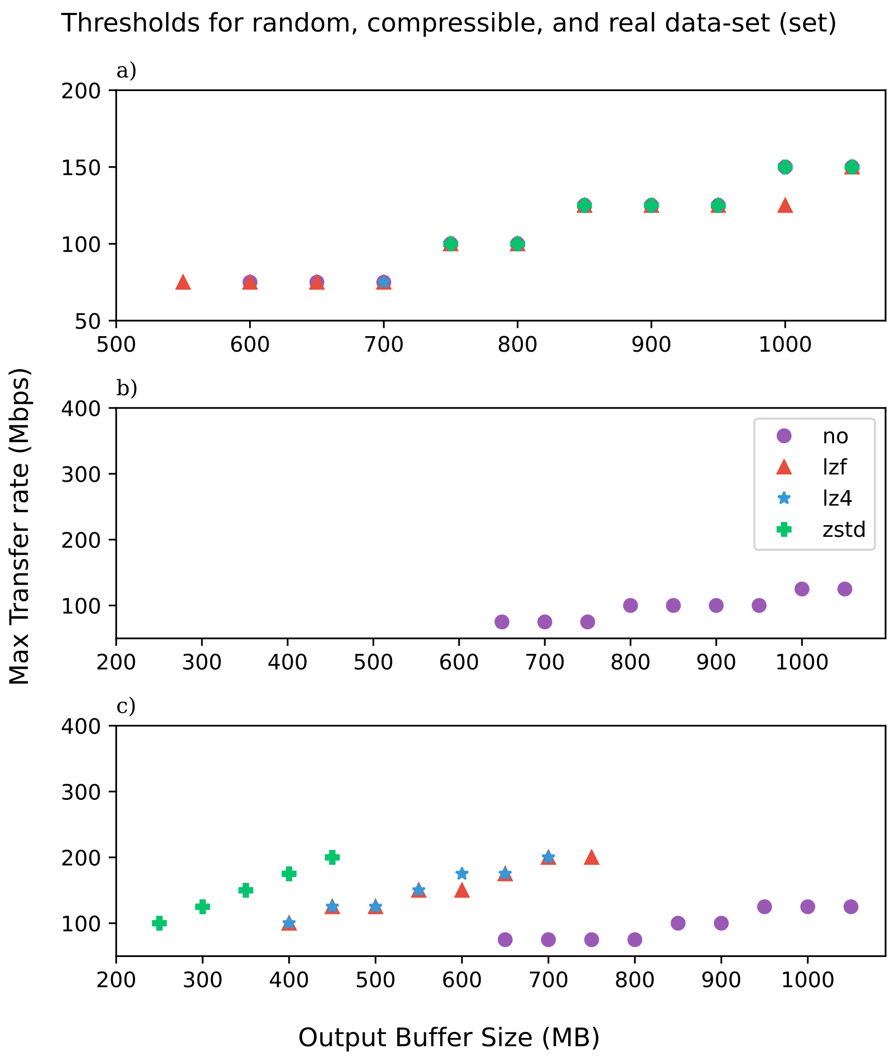

Figure 5.5: Max Thresholds for different data types and the corresponding client-output-buffer size.

Random data set

Tests results for the random data set are shown in Figure 5.5 a). At first glance, there is no difference between compression types and the baseline when it comes to the maximum transfer rate. Throughout most of the tests, the addition of compression behaved similarly to the baseline. However, there is a small transfer rate difference of 25Mbps when the output buffer size equals 1GB, and the lower value is achieved when using `LZF` compression. Additionally, the `ZSTD` algorithm results for output buffer sizes lower than 750MB are not visible in the plot because the transfer rate is somewhere below 50Mbps (which is the minimum point tested in our scenarios).

Compressible data set

The experiment results for the compressible data set are presented in Figure 5.5 b). On one hand, the baseline had a maximum transfer rate of 150Mbps, starting from 75Mbps with no noticeable improvement of the maximum transfer rate. On the other hand, when applying compression (`LZF`, `LZ4`, and `ZSTD`), the maximum transfer rate is above the 600Mbps max transfer rate lower bound and it is reached when using compression over this data set. This result is mainly due to the high compression ratio over the compressible data set shown previously in Table 5.1. With such small memory buffer size the system can withstand much higher transfer rates than the ones achieved through our testing.

Real data set

The analyzed data shows that compression algorithms on the real data set provide higher maximum transfer rates at small buffer sizes when compared to the baseline. The best performance is achieved with the `ZSTD` compression algorithm, which is able to support a `client-rate-limit` of 100Mbps with a `client-output-buffer` of 250MB. To the contrary, the no compression case supports a `client-rate-limit` of 100Mbps with 850MB. The buffer size needed for a maximum transfer rate of 100Mbps is more than three times smaller when using the `ZSTD` compression.

When looking at the `LZ4` compression algorithm, we observe a worse performance than the `ZSTD` algorithm. `LZ4` compression can support a maximum transfer rate of 100Mbps with an output buffer size of 400MB, which is a buffer two times bigger than when using `ZSTD`. In addition, lower bound transfer rate is achieved when the `client-output-buffer` size is 750MB, which is with 200MB more than what `ZSTD` was able to reach.

The `LZF` algorithm provided the worst maximum transfer rate for the tested output buffer sizes. In order to achieve a maximum transfer rate of 100MB the compression algorithm needs a buffer size of $400MB$. This result is similar to the performance given by `LZ4`. However, achieving the lower bound threshold needs a bigger buffer size than all the other compression algorithms tested. It requires a buffer size of 800MB, while `LZ4` needs with 50MB less.

5.5 Performance metrics

This section presents many performance metrics such as Page Faults, CPU Utilization, and Memory Usage for the Container; besides, we present Flame Graphs results. Please note that the CPU Utilization and Memory Usage plots had $t = 0$ as the point when the test scenario reached the replica restart phase.

5.5.1 Page Faults

There is a correlation between performance and page faults. The higher the page fault number, the bigger the performance impact. When a soft page fault occurs, the process tries to access a memory block stored somewhere else in memory. Memory needs to be brought to the proper location to be accessed. This mechanism slows the `redis-server` process, as it needs to wait for memory access. Table 5.3 showcase the number of page faults for each compression type, data set type, and command type.

No Compression			
Command type	Random	Compressible	Real
`SET`	4718674 ± 8349	**4716620** ± 6269	**4922791** ± 577
`MSET`	4489141 ± 26149	4535460 ± 25582	4885821 ± 351
`HSET`	4714659 ± 4545	4703759 ± 4890	N/A

LZF			
Command type	Random	Compressible	Real
`SET`	4743947 ± 3651	2333726 ± 3074	3398651 ± 231
`MSET`	4633038 ± 4702	2291031 ± 7193	3390270 ± 507
`HSET`	4736606 ± 2991	2285642 ± 3033	N/A

LZ4			
Command type	Random	Compressible	Real
`SET`	4729454 ± 5293	2307166 ± 2555	3379078 ± 262
`MSET`	4541235 ± 16138	2228610 ± 11841	3371620 ± 303
`HSET`	4721163 ± 4888	2263235 ± 3316	N/A

ZSTD			
Command type	Random	Compressible	Real
`SET`	4730340 ± 5134	**2293199** ± 3910	**3004188** ± 595
`MSET`	4581633 ± 12316	2224023 ± 9215	2999980 ± 338
`HSET`	4727390 ± 5268	2249039 ± 3130	N/A

Table 5.3: Number of Page Faults and the corresponding standard deviation for each compression type, including no compression, for all command types. Results are averaged over 20 test executions.

Data shows a significant reduction of page faults when using the compressible data set, by about 51% when using `ZSTD` compression, in the case of `SET` command. As a general trend, there is about a 50% reduction on average from the baseline case and all other compression types when the compressible data set is analyzed. Intuitively, the random data set did not see any reduction in page faults from compression. This is correlated to compression ratios from Table 5.1. However, `MSET` has a slightly smaller page fault count, and this is due to the nature of the command, where the chunks of data are ten times bigger.

Regarding the real data set, our analysis shows a reduced number of page faults by about 38.5% when using `ZSTD` compression. Overall, the number of page faults is reduced by about 30% regardless of the compression type. The result mentioned before shows that with data that is compressible, we do not only gain memory reduction and can store more data (see Section 5.1), but also gain better performance from not spending CPU time on page faults. However, there is a trade-off between the CPU impact of compression and the gains from reducing the number of page faults. The impact of compression on the CPU utilization is discussed next in Section 5.5.2.

5.5.2 CPU Impact

This subsection shows extracted data from Grafana regarding CPU utilization. Figure 5.6 presents CPU User Space Usage rate over 5 seconds, for each compression type, and for the SET and MSET command when real data set is tested. The plot starting point is relative to the time when the Redis replica node is restarted. We omit the random and compressible data set for legibility but they are included in Appendix A.

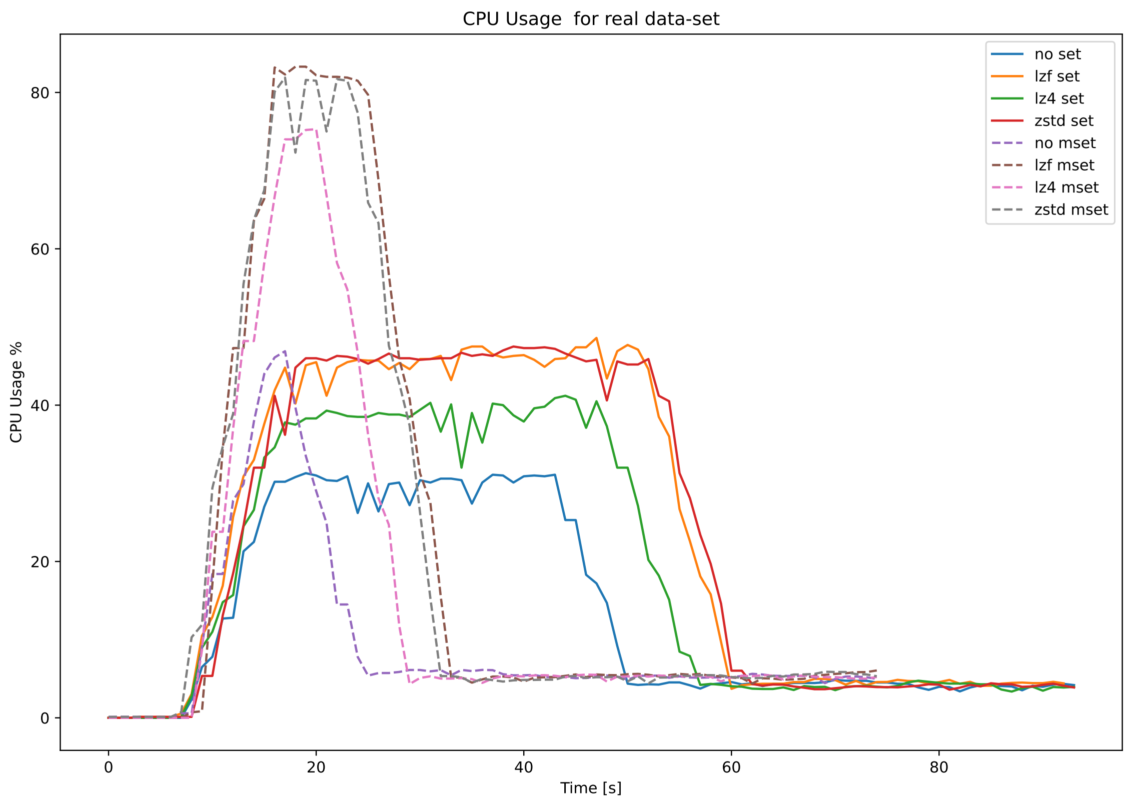

Figure 5.6: CPU Usage for SET and MSET data types over time. Labels show compression type.

Our data shows that CPU usage on the primary Redis Server is higher when using the MSET command. In the baseline case, the CPU Usage reaches about 46%, while when using compression, it almost doubled, reaching about 82%. The LZ4 algorithm has the lowest CPU usage, reaching 75%. In contrast, the other compression algorithms went over 80%. The SET command does not reach high CPU Usage, peaking at about 46%. In the baseline case, tests executed with SET command show a maximum CPU Usage of about 31%. However, when comparing the CPU usage over time, MSET tests spent half of the SET time to finish.

5.5.3 Memory Usage

Figure 5.7 shows the memory used by the container that deployed the `redis-server` primary instance. Time-wise, the plot starting time correlates with the replica restart point. Moreover, Figure 5.7 presents memory usage for each compression type when the real data set test is executed. Both SET and MSET commands are shown. It is observed that the memory increases faster for the MSET command, and this is due to the data size being significantly bigger; thus, the Redis Server processes these commands faster. Memory usage comparison between the SET and MSET commands in the baseline case is rather different. SET memory usage is slightly higher than with MSET. This difference is mainly due to increased command strings that the SET commands produces. Moreover, this slight difference in compression size between the two commands is preserved throughout all the compression types tested (LZF, LZ4, ZSTD).

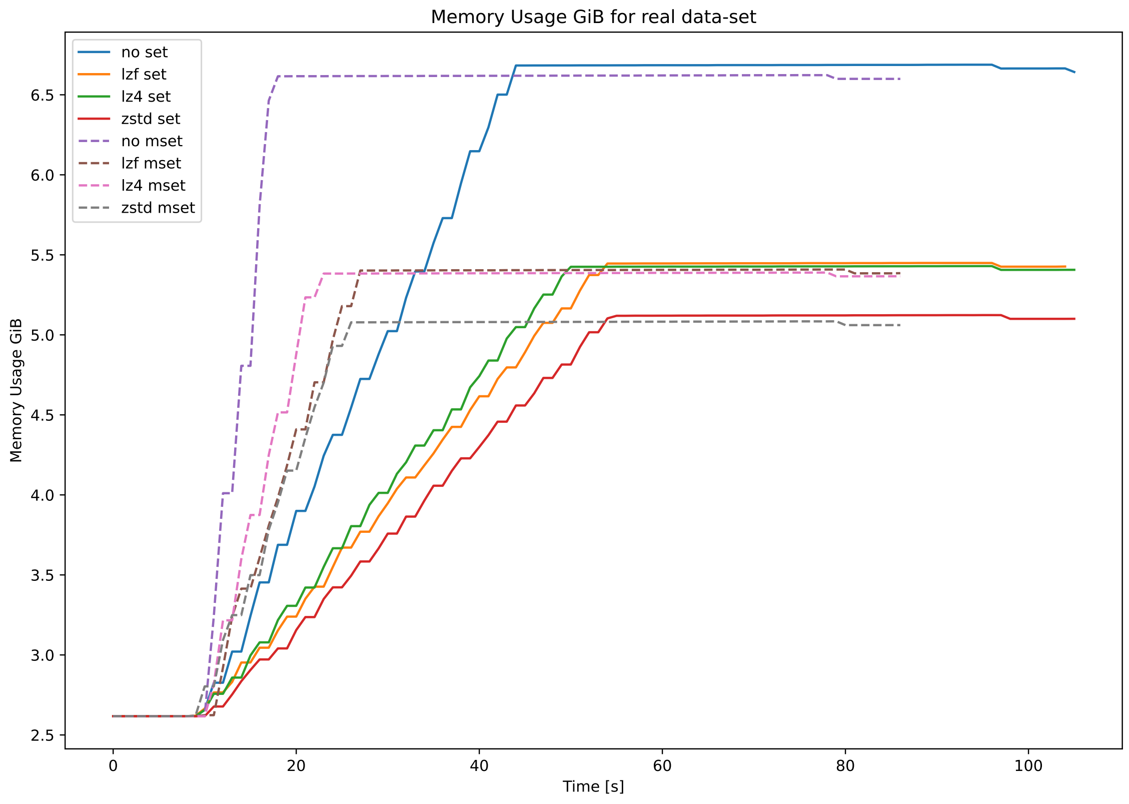

Figure 5.7: Memory Usage for SET and MSET data types over time. Labels show compression type.

5.5.4 Flame Graph analysis

This Section presents the Flame Graph results arising from the evaluation. Table 5.4 and 5.5 show the CPU usage of compress and decompress functions over the course of test execution. As a general trend, the *compression* functions are more CPU intensive than the *decompression* functions.

When looking at the compression functions over the tests executed with `MSET` command, it is observed a two-times increase in CPU usage compared to the `SET` command case. Moreover, we remark that `LZF` compression function uses around 60% of the CPU time throughout the test, while `LZ4` spends only about 10%. The result mentioned above shows that `LZF` is more processor intensive than `LZ4` in the random data set case. Next, we notice that the compressible data set when testing `MSET` command used the least resources, reaching a maximum of 22% time spent on the CPU. Lastly, when the `LZ4` algorithm is utilised (on compressible data set), the compression function only uses 2.4% CPU time.

Our gathered information over the real data set shows about the same CPU usage when comparing `ZSTD` and `LZ4` *compression* functions, with the former being moderately better (18.7%) than the latter (20.5%); nonetheless, `LZF` has the worst performance amongst the three, reaching a CPU usage of about 30% and 17.7% in `MSET` and `SET` case respectively.

LZ4			
Command type	Random	Compressible	Real
`SET`	10.6%	1.3%	10.9%
`MSET`	18.6%	2.4%	20.5%
`HSET`	10.8%	1.1%	N/A

LZF			
Command type	Random	Compressible	Real
`SET`	41.1%	12.2%	17.7%
`MSET`	60.9%	22.4%	30.07%
`HSET`	39.4%	11.9%	N/A

ZSTD			
Command type	Random	Compressible	Real
`SET`	12.6%	1.9%	10.9%
`MSET`	23.4%	3.7%	18.7%
`HSET`	12.3%	1.9%	N/A

Table 5.4: Flame Graphs results. Percentage (from max 100%) for relative time spend on CPU for **compress** functions.

The execution of the decompression function is way less CPU intensive than it's counterpart. Data from Table 5.5 over the random data set shows little to no CPU usage, compared to the other data sets. The reason behind the result will be detailed in the Discussion Chapter 6, but the general idea is that random data set is not compressible and thus decompression function calls are scarce.

Regardless of compression types, the MSET command has twice the CPU utilization of SET. This observation holds for all data sets, exception making some random data set scenarios where decompression is almost 0%. In addition, the HSET command behaved similarly to the SET command, having the same CPU utilization throughout all the relevant compression and data set types.

LZ4			
Command type	Random	Compressible	Real
SET	1.1%	0.6%	2.6%
MSET	2.5%	1.2%	5.2%
HSET	1.1%	0.5%	N/A
LZF			
Command type	Random	Compressible	Real
SET	0%	6.2%	5.4%
MSET	0%	11.9%	9.7%
HSET	1.9%	6.2%	N/A
ZSTD			
Command type	Random	Compressible	Real
SET	1.3%	1.4%	3.6%
MSET	0%	3.1%	6.5%
HSET	1.3%	1.4%	N/A

Table 5.5: Flame Graphs results. Percentage (from max 100%) for relative time spend on CPU for **decompress** functions.

Figures 5.8 and 5.9 show the results of the metrics extracted via the `perf` tool using Flame Graphs. The `perf` tool is started inside the container that deploys the primary Redis Server and monitors the CPU Usage throughout the test execution. The perf process monitors all `redis-server` processes, including its children and threads. Once the test finishes, the perf process is killed, and the scrip that generates Flame Graphs is run over for data resulting from perf execution. Both Flame Graphs Figures represent the results of executing the ZSTD compression algorithm.

The first Figure (5.8) presents the perf stacktrace when using the SET command. The Most CPU intensive functions are the `writeToClient()` (about 27% CPU time) and `connSocketEventHandler()` (about 34% CPU time) that are called inside the `aeProcessEvents()` polling function. The ZSTD compress, decompress, and auxiliary function calls are separate from the
textttmain() function of the `redis-server`. The reason for it is that ZSTD is included as a library and the `perf` tool included these calls at the same level as the main function call (`__libc_start_main()`). In total, ZSTD functions account for about 27% of the CPU usage.

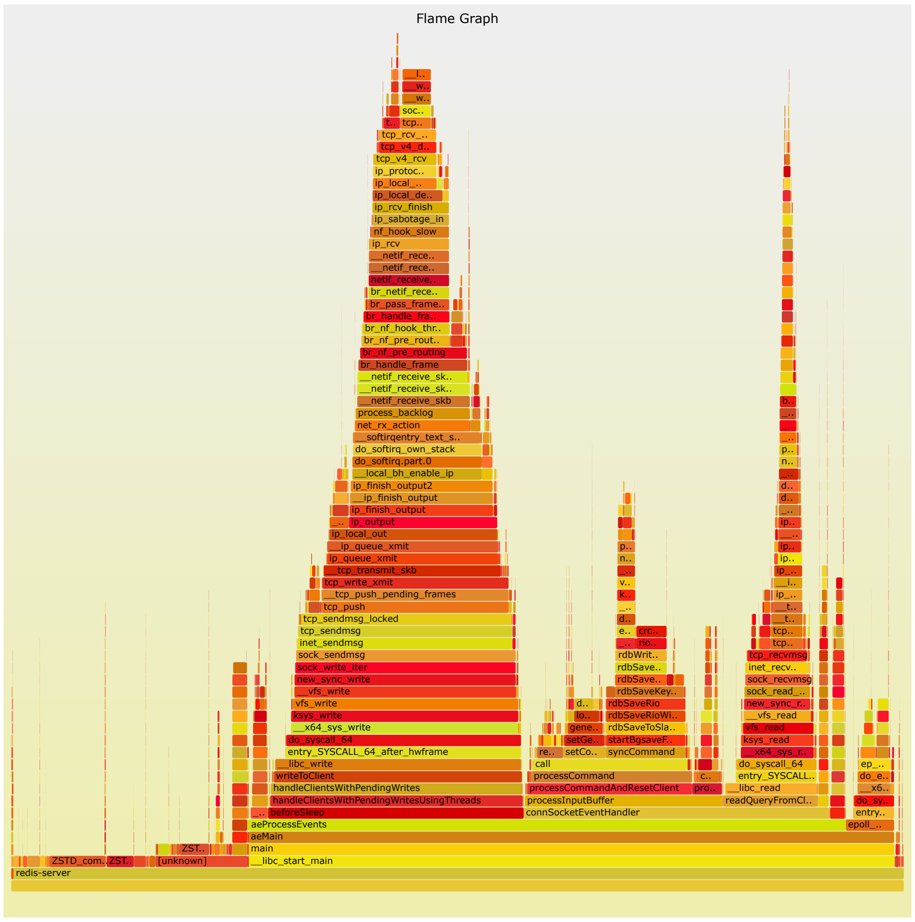

Figure 5.8: Flame Graphs for `SET` command over real data set using `ZSTD` compression.

The second Figure (5.9) presents the perf stacktrace when using the `MSET` command. The Most CPU intensive functions are different from the `SET` case, having the `ZSTD` compression, decompression, and auxiliary function calls accounting for more than 42% of the total CPU usage. The `connSocketEventHandler()` function call spent about 38% CPU time. Conversely, the `writeToClient()` function accounted for only approximately 4% of the CPU time.

Comparing the two executions, we can observe that the `ZSTD` compression algorithm utilizes more CPU time when using the `MSET` command than when using `SET`. In addition, the `writeToClient()` is less significant when it comes to the CPU impact in the `MSET` case, being a difference in CPU usage of 23% between command types.

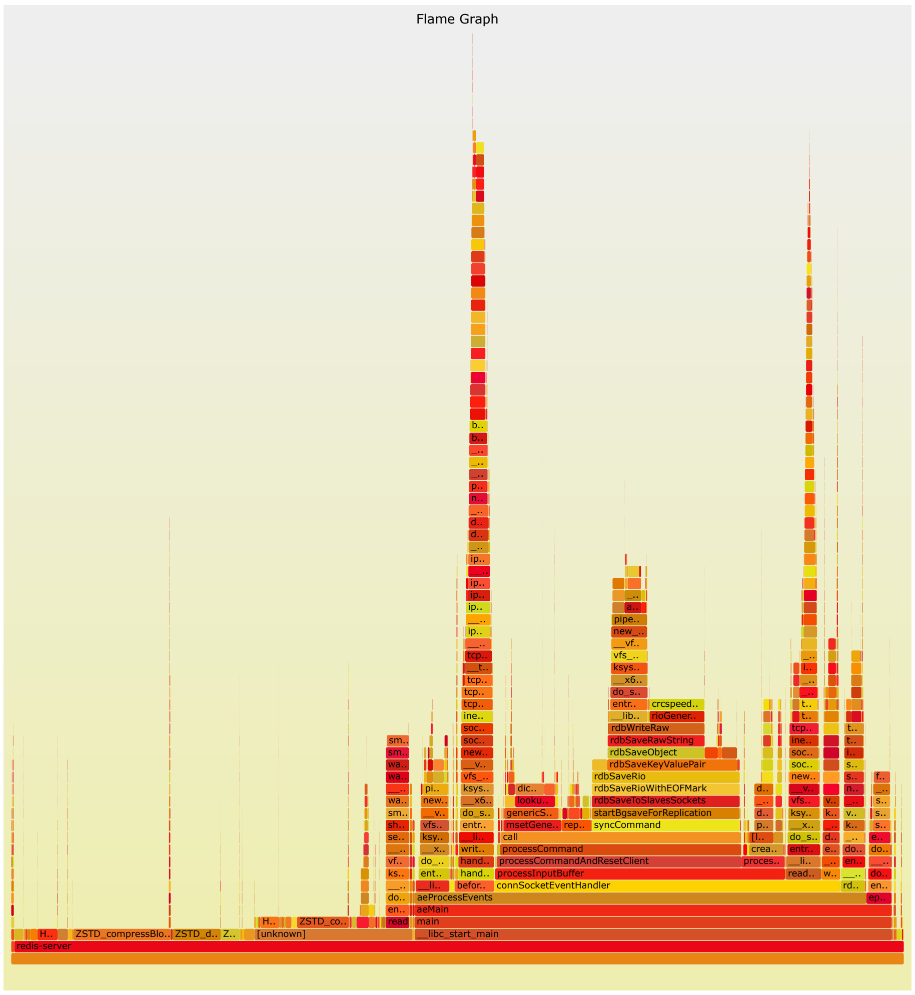

Figure 5.9: Flame Graphs for `MSET` command over real data set using `ZSTD` compression.

6 Discussion

This chapter comments on the data presented in Results Chapter 5, interpreting and connecting performance metrics results with the performance of the compression algorithms.

6.1 Command types

We have observed that the memory footprint depends on the command type that the client is using to write to the `redis-server`. For example, the `MSET` and SET commands yield different memory footprints when compression was applied. This dependency takes place because the reply buffer allocates bigger chunks of data in the case of the `MSET` command, and thus the compression is applied on a bigger chunk of data. In addition, the `MSET` command utilizes the CPU better, as data is sent faster and the memory increases sharply. While for the SET command, the sending time is longer, the memory increases steadily and the CPU usage is kept at a lower utilization. In connection to command types, it was also observed that the bigger the data size, the higher the resource utilization and the faster the requests were processed by the primary Redis Server.

6.2 Flame Graph analysis

When comparing compression and decompression functions CPU usage over the random data set we can observe a big CPU utilization difference between them. Looking at the `SET` and `MSET` commands when the selected compression algorithm is `LZF`, Table 5.4 shows that CPU usage is 41% and 60%, respectively; in contrast, over the same selection of parameters, we see in Table 5.5 that the CPU usage is 0% for the `decompression` function. The reasoning for the mentioned result resides in the nature of the random data set. Because random data does not compress well, the number of chunks to decompress is so small that is almost 0%; unsurprisingly, the compression algorithm still needs to try the compression of such data, resulting in high CPU usage, but with no to little benefits. Most test results show lower CPU usage on random data set than over compressible and real data sets. The decompression function is called more often on data sets with a good compression ratio than to the random data set where the compression ratio is 1.0.

6.3 No performance degradation

Our tests concluded that there was better RPS performance when using `LZ4`, compared to the baseline, over the compressible data set. In other words, there is no performance degradation when running `LZ4` compression over the compressible data set. The finding holds true for all command types: `SET`, `MSET`, and `HSET`. The time spent to allocate memory was reduced, so the processor spent less time on page faults (show page faults for this case) and allocation. In this case, we have added compression and had no cost on performance, gaining better RPS performance. In addition, data over `ZSTD` shows a positive performance when `HSET` command was tested.

Next, we present the reasoning for why the achieved RPS metric was better when adding an extra step to the `redis-server's` command processing procedure. Lastly, all the below arguments combined achieved the performance benefit and were not taken separately.

First explanation: When data is received, the allocator gives the `redis-server` a pointer with some allocated block of data. This data is about 1KB. Because the compression ratio is very good with the compressible data set, the reallocation, when calling the `zfree()` function, does not release the memory right away, keeping it for later use. `Jemalloc` uses background threads to purge memory asynchronously from time to time. This gives the `redis-server` process more time to serve incoming commands instead of requesting memory. In the baseline case when using the `SET` command, the function `jemalloc_bg_thd()` spends a total of 1.62% on the processor, while when using `ZSTD` it spent only 0.03% on the processor. Note that compression ratios for compressible data set are shown in Table 5.1 and the percentages are relative to the total time spent on the processor while running the tests.

Second explanation: Compression on chunks of the same letter repeating was not as CPU intensive as random or real data set. This was presented in more detail in Section 5.5.2 and in 5.5.4.

Third explanation: Analyzing the page faults for each compression type, data set, and command type, we saw a smaller number of page faults when the compressible data set was tested. The reduction in page faults was visible for all compression algorithms, but the performance benefit was only achieved in combination with the other arguments presented above. For example, `ZSTD` had a reduction in page-faults of 51% compared to baseline when using `SET` command.

6.4 Sync duration

Table 5.2 shows the synchronization duration differences between compression algorithms and baseline. Data shows a general trend of not having more than 3.77%

difference in sync duration, averaging at 2.25%. However, these results could potentially be affected by performance tools that run on VM, such as *perf, Grafana, Prometheus, cAdvisor*. Most of these use the same network bandwidth, and when there is no limit on Ethernet devices, there could be interference, as the transfer rates between the primary and replica nodes can reach up to `13Gbps` transfer rate. Transfer rate without traffic control limits can be easily approximated by dividing the transferred data size over the sync duration.

Based on the gathered data, it could be interpreted that the decompression algorithm does not impact the synchronization duration. We saw faster sync duration when decompression algorithms are used. The slight differences in sync time are not significant enough to show a performance impact of decompression algorithms when no transfer limit is applied.

Moreover, by inspecting Table 5.5, it can be observed that decompression functions do not spend too much time on the CPU, the worst case being 9.7% relative CPU time throughout the test execution. In addition, it needs to be specified that the number of page faults is significantly reduced when compression algorithms are used. By combining the performance loss from CPU spent time and the benefits of reduced page faults, it can be said that sync time is not affected by applying decompression on the data sent to the replica.

The synchronization time was overall shorter with about 10% (1 second) when using `MSET` command. This observation holds when `MSET` command is compared against all other command types, regardless of what type of data set or compression algorithm is used.

7 Conclusion

The thesis work focused on analyzing how a widely used solution such as the Redis DB behaved when stream compression was added to output buffers present on the Primary node while a total synchronization was ongoing. The tested system consisted of a Primary node and a replica node. The replica node was a read-only node that kept an exact copy of the data received from the primary Redis Server. We chose three compression algorithms: `LZF`, `LZ4`, and `ZSTD`. The first one was already present in the Redis source code and used for compressing `Redis Objects`. While the last ones are industry-standard compression algorithms widely adopted.

Compression was added into the `Redis Network Layer`, where all the client (client - from the node's point of view) data communication is done. While synchronization between nodes is ongoing, the compressed data was buffered until they were back in `online` state. From here on, decompression was added. Our evaluation was conducted by extracting the following metrics for the Redis Server: memory usage of the application, CPU Usage of the application, supported Requests Per Second (RPS) by the Redis Server, Page Faults, Maximum Transfer Rate for a given `client-output-buffer`, synchronization Duration, and Flame Graphs for compression and decompression functions. Our tests were averaged over 20 test executions for the RPS and Sync Duration metrics.

We saw that compression significantly reduced memory usage for our most data sets, exception being the random data set. In addition, the CPU usage was about two times higher than the baseline when compression was added. On one of the data sets (compressible), we have had a two times reduction in page faults. The page-fault reduction positively impacted the performance, and the compression case performed better on the RPS metric.

The RPS was overall reduced when compression was added, thus impacting performance. However, compression increased the maximum threshold for transfer rate that a Redis Server can support from incoming client requests. In order to accommodate the same Maximum Transfer Rate, on the real data set, our results showed that compression reduced the `client-output-buffer` usage by an order of three. Our tests concluded that decompression does not have a significant performance impact over the synchronization duration.

In the future, based on the current work done in this area, we can expand on the following ideas:

- Instead of decompressing in the same node, we can pass the compressed data directly to the socket and expect the other end to decompress. Decompression done on the replica node could also have a bandwidth performance.
- Currently, the thesis work has focused on a primary and a replica node. In future work, the case with N primary nodes and K replicas could be studied, potentially with the built-in solution offered by Redis called `Cluster Nodes`.
- This work was none on an open-source project and with some minor refactoring, the compression addition could be submitted to the Redis Source Code to be widely available to all its users.
- The `ZSTD` and `LZ4` algorithms offer parameters to balance the trade-off between CPU utilization and compression ratio. The thesis work focused on using the default values for these algorithms. In future work, one can find the best values for the compression parameters to satisfy the application needs.

Bibliography

[1] Flame graphs `https://www.brendangregg.com/flamegraphs.html` (accessed: 22.12.2021).

[2] Google/cadvisor: Analyzes resource usage and performance characteristics of running containers. `https://github.com/google/cadvisor` (accessed: 06.05.2022).

[3] Lz4 - extremely fast compression `https://github.com/lz4/lz4` (accessed: 06.05.2022).

[4] Lz4 - extremely fast compression `http://www.oberhumer.com/opensource/lzo/` (accessed: 08.05.2022).

[5] Prometheus monitoring system & time series database. `https://prometheus.io/` (accessed: 16.05.2022).

[6] Redis benchmark `https://redis.io/docs/reference/optimization/benchmarks/` (accessed: 06.05.2022).

[7] Redis `https://github.com/redis/hiredis` (accessed: 16.04.2022).

[8] Redis `https://redis.io/docs/libraries/` (accessed: 14.04.2022).

[9] Redis `https://redis.io/documentation` (accessed: 22.12.2021).

[10] What is redis? `https://aws.amazon.com/redis` (accessed: 22.12.2021).

[11] C.-P. Bezemer, J. Pouwelse, and B. Gregg. Understanding software performance regressions using differential flame graphs. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 535–539, 2015.

[12] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. Towards scalable and reliable in-memory storage system: A case study with redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1660–1667, 2016.

[13] R. Chopade and V. Pachghare. A data recovery technique for redis using internal dictionary structure. *Forensic Science International: Digital Investigation*, 38:301218, 2021.

[14] Y. Collet and M. Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, Feb. 2021.

[15] L. P. Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[17] B. Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, may 2016.

[18] A. T. Kabakus and R. Kara. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences*, 29(4):520–525, 2017.

[19] S.-J. Kwon, S.-H. Kim, H.-J. Kim, and J.-S. Kim. Lz4m: A fast compression algorithm for in-memory data. In *2017 IEEE International Conference on Consumer Electronics (ICCE)*, pages 420–423, 2017.

[20] Q. Liu and H. Yuan. A high performance memory key-value database based on redis. *Journal of Computers*, 14:170–183, 2019.

[21] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover. Exact discovery of time series motifs. In *Proceedings of the 2009 SIAM international conference on data mining*, pages 473–484. SIAM, 2009.

[22] G. Wu, X. Wang, Z. Jiang, J. Cui, and B. Wang. Compression algorithms for log-based recovery in main-memory data management. volume 10055, pages 56–64, 11 2016.

[23] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1119–1134, New York, NY, USA, 2016. Association for Computing Machinery.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

A

Appendix 1

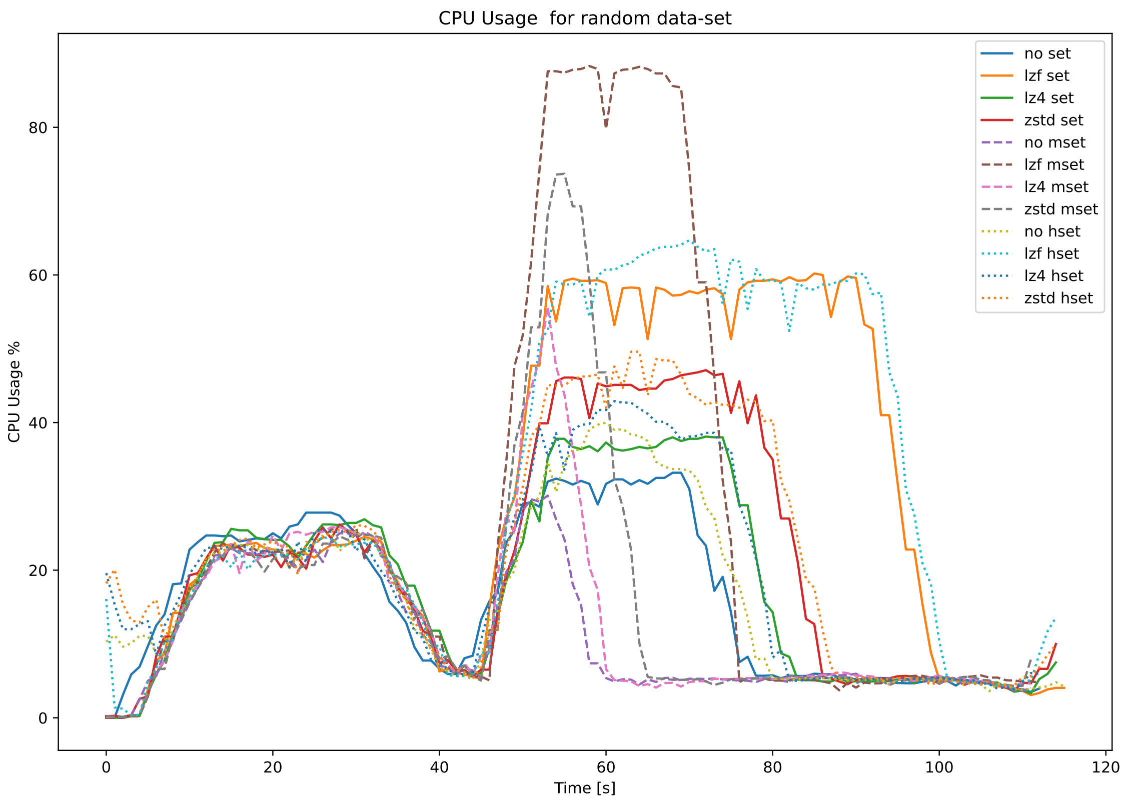

Figure A.1: CPU Usage for `SET`, `MSET` and `HSET` data types over time. Labels show compression type and command type

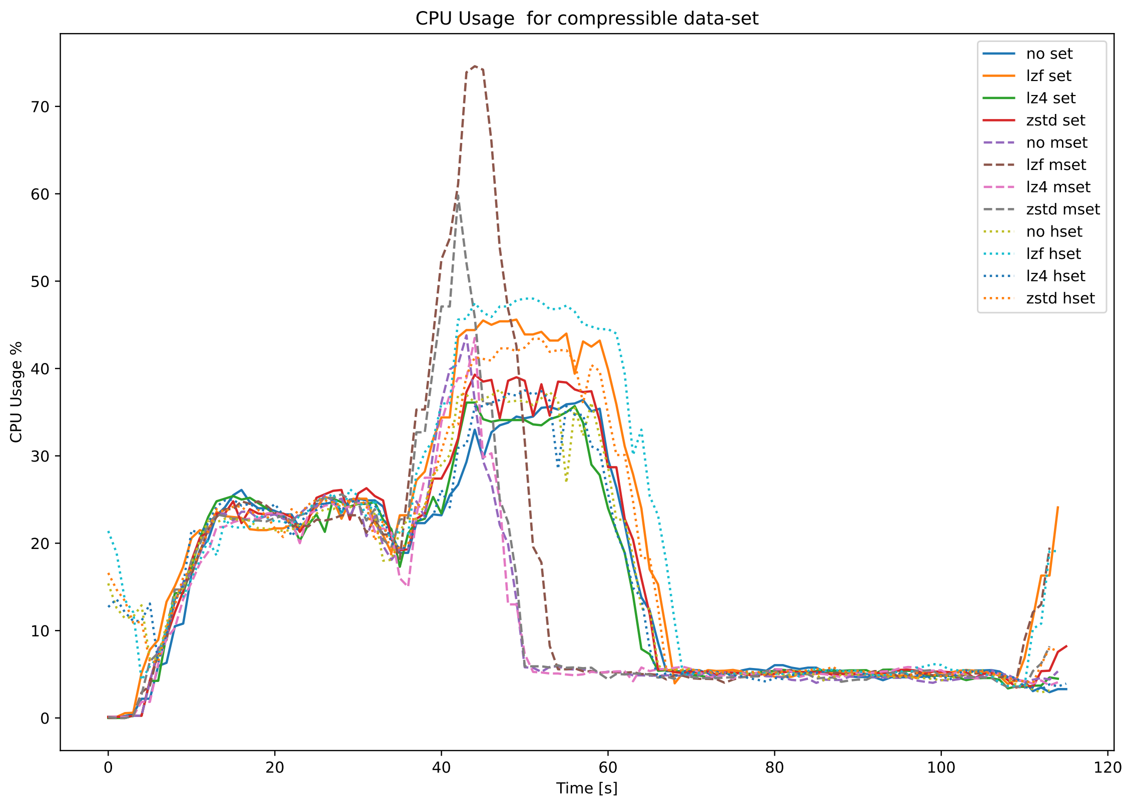

Figure A.2: CPU Usage for `SET`, `MSET` and `HSET` data types over time. Labels show compression type

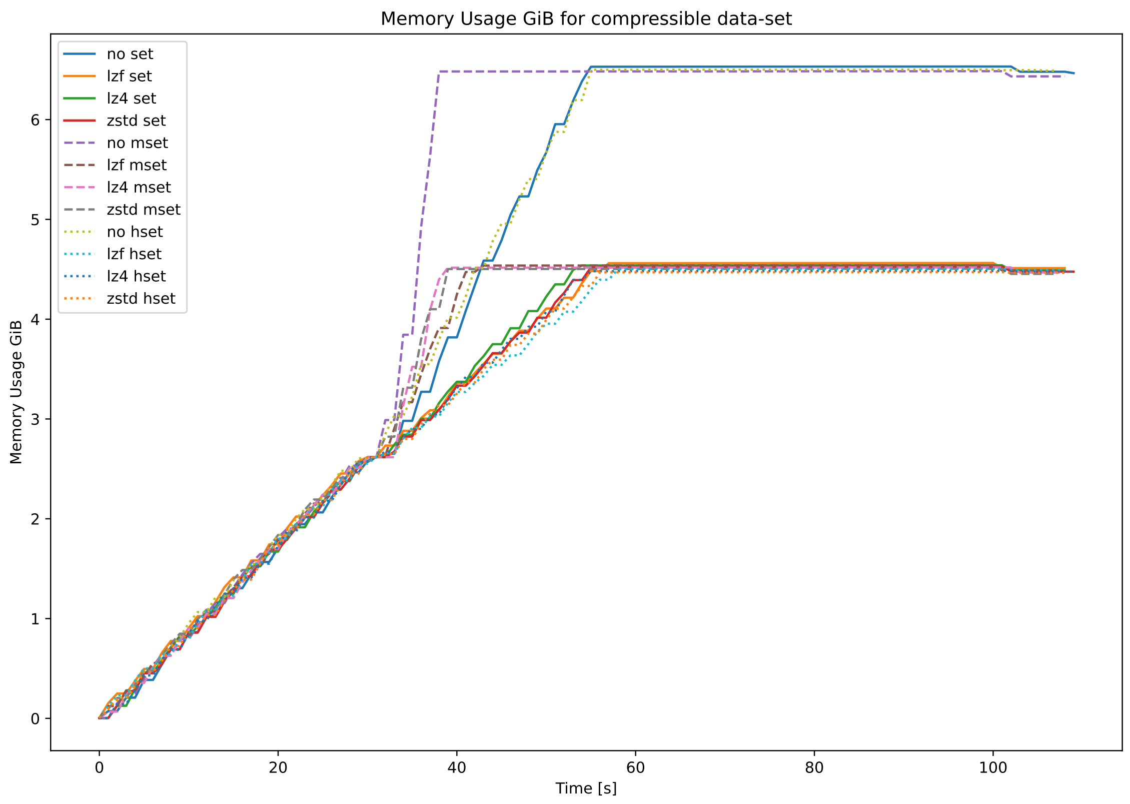

Figure A.4: Memory Usage for `SET`, `MSET` and `HSET` data types over time. Labels show compression type.

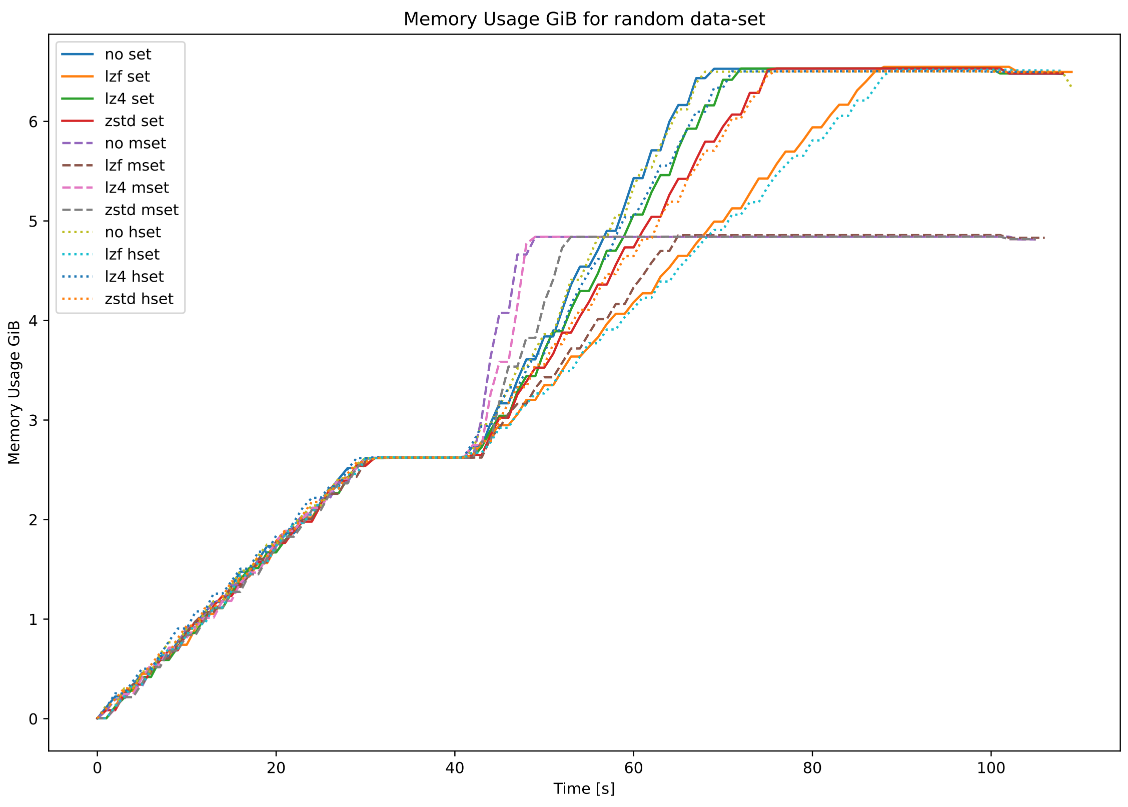

Figure A.3: Memory Usage for `SET`, `MSET` and `HSET` data types over time. Labels show compression type.

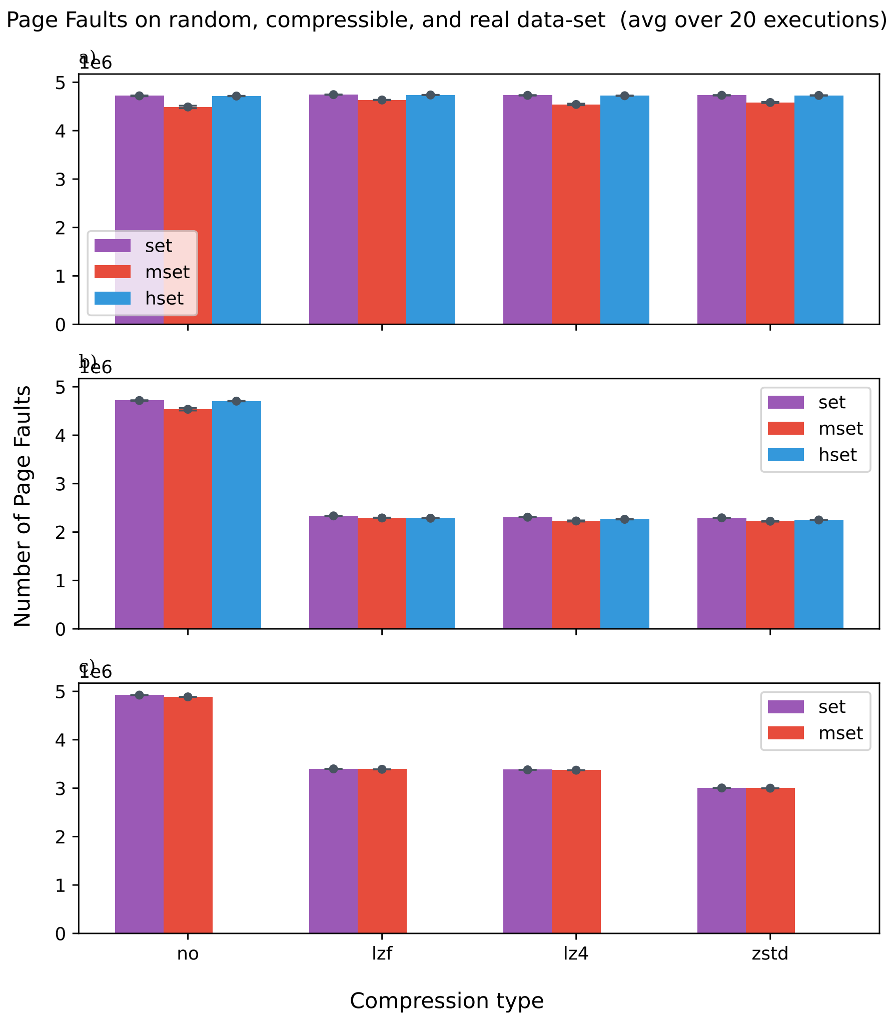

Figure A.5: Number of Page Faults during test executions and the corresponding standard deviation.

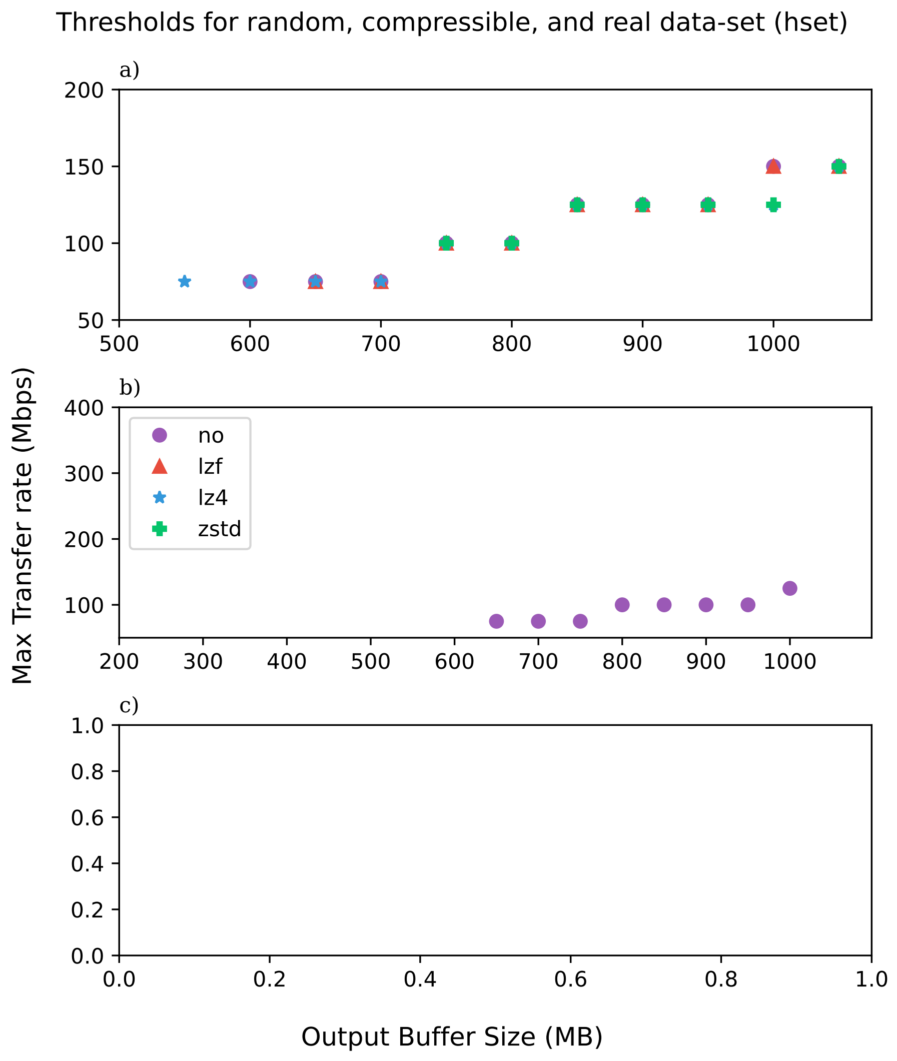

Figure A.6: Max Thresholds for different data types and the corresponding client-output-buffer size.

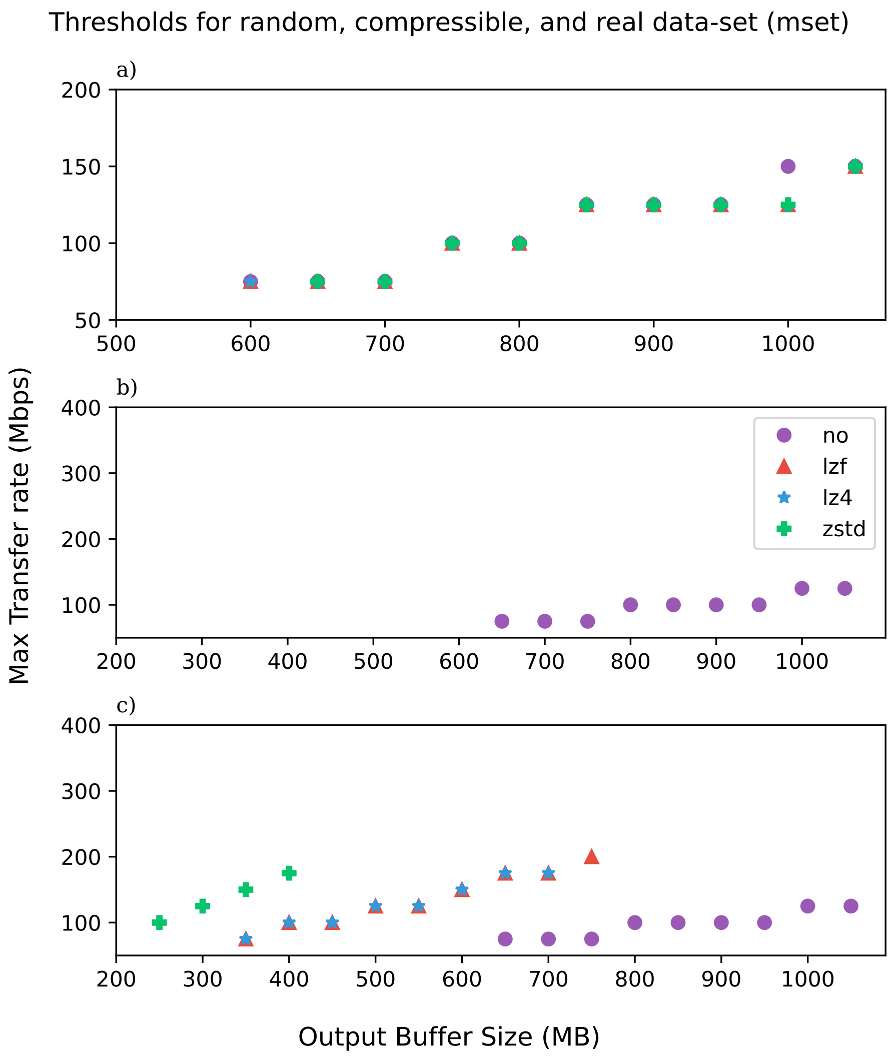

Figure A.7: Max Thresholds for different data types and the corresponding client-output-buffer size.

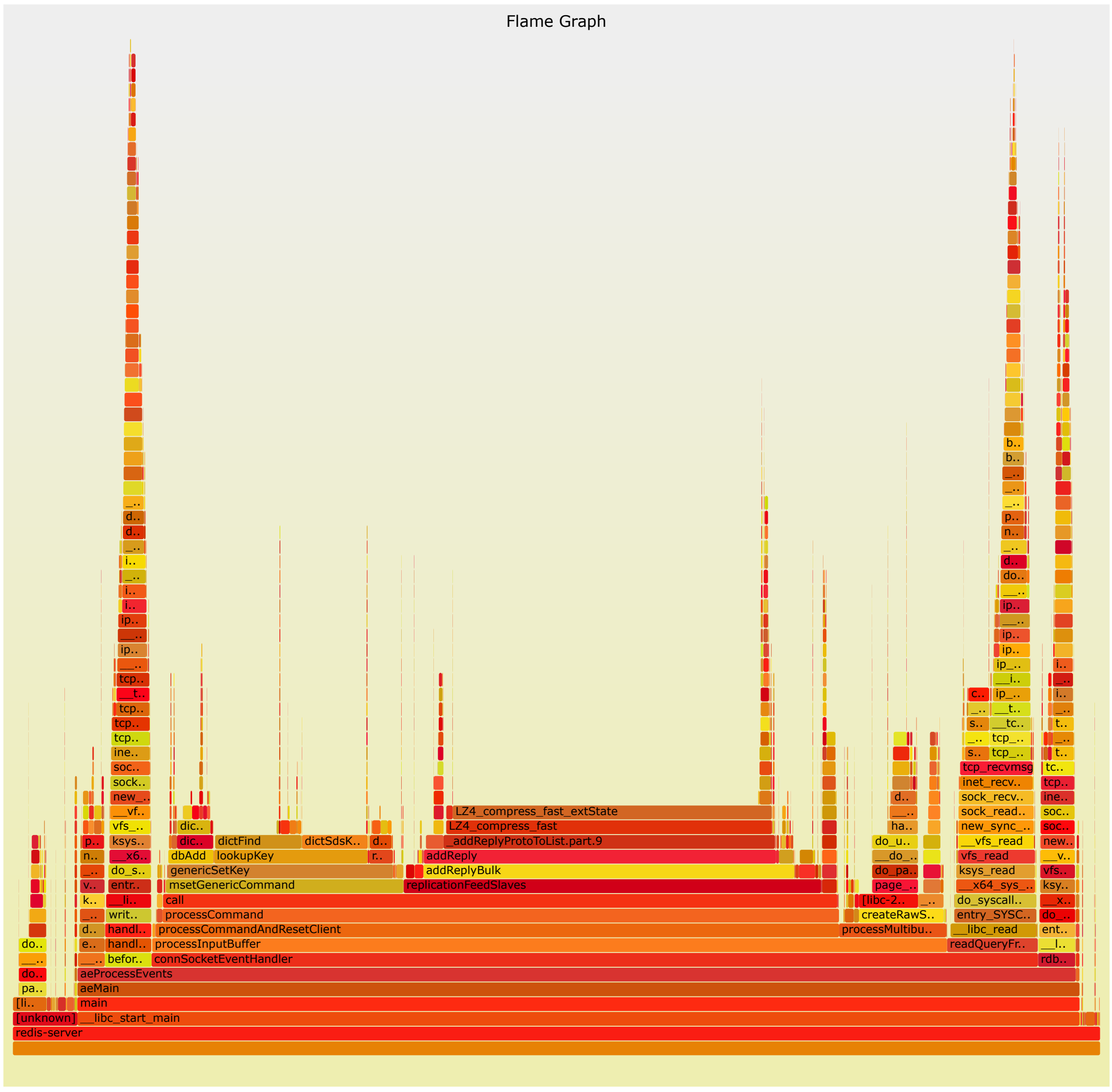

Figure A.8: Flamegraphs

Acronyms

RDB Redis DataBase. 5, 7, 9, 15
RPS Requests Per Second. xi, 1, 2, 16, 17, 22, 23, 31, 34–37, 50, 53

SUT System Under Test. 13, 17, 18, 26