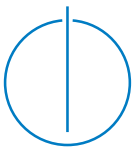
Christoph Doblander

Compression in Publish/Subscribe Systems

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Compression in Publish/Subscribe Systems

Christoph Doblander

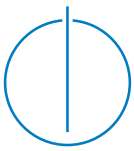
Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr.-Ing. Matthias Althoff
Prüfer der Dissertation:	1. Prof. Dr. Hans-Arno Jacobsen
	2. Prof. Dr. Peter Triantafillou The University of Warwick

Die Dissertation wurde am 29.10.2018 bei der Technische Universät München eingereicht und durch die Fakultät für Informatik am 23.11.2018 angenommen.



Technische Universität München

Department of Informatics

Compression in Publish/Subscribe Systems

Christoph Doblander

Complete copy of the dissertation approved by the Department of Informatics of the Technical University of Munich in partial fulfillment of the requirements for the degree of

Doktors der Naturwissenschaften (Dr. rer. nat.)

Chair:	Prof. Dr.-Ing. Matthias Althoff
Dissertation examiners:	1. Prof. Dr. Hans-Arno Jacobsen
	2. Prof. Dr. Peter Triantafillou The University of Warwick

The dissertation was submitted to the Technical University of Munich on 29.10.2018 and accepted by the degree-awarding institution of Department of Informatics on 23.11.2018

Abstract

Constraint bandwidth imposes a challenge to publish/subscribe systems (pub/sub). Publish/Subscribe systems are often used by interactive mobile applications to communicate with system back-ends. State-of-the art compression techniques such as GZIP or DEFLATE can be universally employed to reduce bandwidth. We propose dictionary-based compression and online self-learning dictionary maintenance algorithms to go well beyond the bandwidth reductions achieved with the state-of-the-art.

In this thesis, we show how to reduce bandwidth even further by employing Shared Dictionary Compression (SDC) in pub/sub. However, SDC requires a dictionary to be generated and disseminated prior to compression, which introduces additional computational and bandwidth overhead. The overhead is amortized by high compression ratios on messages. To support SDC, we propose extensions for pub/sub and employ a new class of brokers, called sampling brokers. Sampling brokers have the responsibility to create, disseminate and maintain the dictionaries over time. Dictionary maintenance is performed in an online way to adapt to new content biases and keep the bandwidth reductions high using a Dictionary Maintenance Algorithm (DMA). The evaluation of our proposed design shows that it is possible to compensate for the introduced overhead and achieve significant bandwidth reduction over off-the-shelf compression algorithms like DEFLATE.

The dictionary is created based on multiple parameters. How large the dictionary is, based on how many messages it is composed and how long the dictionary is valid. We analyse all the interactions of the variables to derive parameters for the compression. In the first DMA we present, called ADAPTIVE we use heuristics to find good parameter combinations. In the second approach we present, called PREDICT we use variable fitting techniques and machine learning to derive even better parameter combinations specific to the nature of the content of the messages and the topology in an automatic way. The third approach we present, called TAPD (Topology aware PREDICT), goes even further and takes the graph of the overlay into account to also achieve high bandwidth reductions in complex overlays with varying amount of messages sent per publishers.

Zusammenfassung

Bandbreitenlimitationen stellen Publish/Subscribe Systeme (Pub/Sub) vor neue Herausforderungen. Pub/Sub wird häufig von mobilen Applikationen benutzt, um mit dem Rechenzentrum zu kommunizieren. Existierende Kompressionsmethoden wie GZip oder Deflate können universell eingesetzt werden, um die benötigte Bandbreite zu reduzieren. Wir stellen einen neuen selbst lernenden Ansatz vor, der Wörterbuch basierende Komprimierung über die Zeit hinweg adaptiert um weit höhere Bandbreitenreduktionen zu erreichen als mit bisher genutzen Methoden.

In dieser Dissertation zeigen wir, wie die benötigte Bandbreite weiter reduziert werden kann mit Wörterbuch basierender Komprimierung, die speziell auf die Anforderungen von pub/sub zugeschnitten wird. Dafür stellen wir Erweiterungen für Pub/Sub Systeme vor. Unter anderem den sogenannten "Sampling Broker", der die Komprimierung überwacht, Adaptionen der Parameter vornimmt und auf Basis dessen die Wörterbücher erstellt, mit denen hohe Komprimierung erreicht werden kann. Unsere Erweiterungen erzeugen die Wörterbücher und teilen sie zwischen den Publishern und den Subscribern. Die Komprimierung wird kontinuierlich überwacht und adaptiert um ständig hohe Komprimierungsgrade zu erreichen. Die inhaltliche Tendenz der Nachrichten ändert sich über Zeit. Damit das Wörterbuch nicht obsolet wird muss das Wörterbuch adaptiert werden. Dazu werden die derzeitigen Kompressionsgrade und die potenziell höhere Komprimierung mit einem neuen Wörterbuch verglichen. Sollte sich herausstellen, dass ein neues Wörterbuch die Komprimierung erhöht, wird den Publishern und Subscribern ein neues Wörterbuch unter der Berücksichtigung des hierbei induzierten Bandbreiten Aufwands mitgeteilt. Die Evaluierung unseres Ansatzes zeigt, dass es möglich ist den zusätzlichen Bandbreiten Aufwand von Wörterbuch basierter Komprimierung durch hohe Kompressionsraten bei Nachrichten zu amortisieren. Dadurch können weit höhere Bandbreitenreduktionen als mit GZip oder Deflate erreicht werden.

Das Wörterbuch wird unter der Berücksichtigung von mehreren Parametern erstellt. Wie groß das Wörterbuch ist, basierend auf wievielen Nachrichten es erzeugt wird und wie lange das Wörterbuch eingesetzt wird. Dazu haben wir alle Interaktionen der Variablen untersucht und schlagen auf Basis dessen mehrere Verfahren vor um gute Parameterkombinationen herauszufinden. Zuerst stellen wir den Algorithmus Adaptive vor, der eine Heuristik benutzt um gute Kombinationen zu eruieren. Als zweiten Punkt stellen

wir PreDict vor - ein Ansatz der durch automatische Kurvenanpassung und maschinelles Lernen gute Kombinationen von Variablen errechnet. Unser dritter Ansatz, TaPD nimmt den darunter liegenden Graphen der Kommunikation als Basis um noch höhere Komprimierungen bei variierender Anzahl von Nachrichten je Publisher zu erreichen.

Acknowledgments

This dissertation and the included research was written at the Department of Informatics at the Technical University of Munich under the supervision of Prof. Dr. Hans-Arno Jacobsen. I am deeply grateful for his valuable advice, motivation, positive attitude, and encouragement throughout the past years and the freedom to choose my own research topics. Having two kids during my PhD was not an easy task. I thank Prof. Jacobsen for being supportive to tackle the challenges from this side by providing a lot of flexibility on work related tasks.

I would also like to thank the rest of my thesis committee: Prof. Dr. Peter Triantafillou for agreeing to be the second examiner and providing suggestions for improvements to this dissertation. Further I want to thank Prof. Dr.-Ing. Matthias Althoff for accepting to chair the commitee.

I enjoyed my time in academia, special thanks go to …

- all other former and present colleagues at the chair. Especially Martin Jergler, Thomas Kriechbaumer, Victor del Razo, Jose Rivera and Pooya Salehi for being such great collegues.
- Prof. (FH) Dr. Johannes Lüthi and Dipl. Ing. Georg Giokas for backing my application to this position by sending their recommendation letters.
- the many students I had the pleasure to supervise for their bachelor or master thesis or guided research projects.
- my parents and my sister for all their support.
- my wife Gabi for her love and emotional support during stressful times and the understanding of often long working hours and weekends. She supported me unconditionally and kept me going at times when I was close to changing my mind. And my kids Xaver and Marlene, which often caused sleepless nights but compensated that 1000x by making my life happier.

Contents

Chapter 1

Introduction

Publish/subscribe (pub/sub) [1, 2, 3, 4, 5, 6, 7, 8, 9] is known as a scalable and efficient data dissemination mechanism that is widely used in the back-end of enterprise, smartphone and Internet of Things (IoT) applications [10, 11]. Its efficiency comes from the optimized routing algorithms. Pub/sub is used to decouple publishers and subscribers in the following dimensions [4]: Space (interacting clients do not need to know each other), time (interacting parties do not need to be actively participating at the same time) and syncronisation (publishers and subscribers communicate asynchronously). These properties are useful in a wide range of applications, hence pub/sub systems are used widely in industry. Spotify [12], a music streaming service, uses a pub/sub system to send notifications to the users. The Facebook Messenger uses Message Queuing Telemetry Transport (MQTT) [13], a pub/sub protocol, to communicate with the back-end. In many of these scenarios, bandwidth usage between the clients and the brokers is a concern.

1.1 Motivation

High bandwidth usage is a significant concern. According to EuroStat [14], 87% of the households in the EU-28 had access to the internet in 2017, while in 2007, only 55% of the households had internet access. Also, broadband usage has increased, 85% of the

households had broadband access which is more than double the households compared to 2007. Eurostat reports that also internet on mobile devices is on the rise. While in 2012, 36% of individuals aged 16 to 74 within the EU-28 used a mobile device to connect to the internet, in 2017 the share was 65%. While in 2016, 96% of the households in total were covered with Long-Term Evolution (LTE), 80% of the rural areas were covered with LTE. This number went up from 36%, LTE coverage in rural areas in 2015.

While broadband usage and high speed coverage is increasing, still many areas are not well covered. LTE coverage depends also on the country, while for example Norway is covered almost 100%, Romania is only covered around 45%. The 3G/4G penetration in Pakistan [15] was around 30% when accessed in October 2018.

Reducing the required bandwidth also reduces costs because mobile connections are typically metered. Additionally, reducing the bandwidth enables interactive mobile phone applications because more messages can be exchanged given the same available bandwidth. High data usage is a concern for smartphone applications and interactive mobile games [16]. Moreover, low bandwidth usage is important in Machine-to-Machine (M2M) communication and IoT [11]. For example, sensors for precision farming in agriculture are limited to transmitting a few kilobytes per day [17].

Bandwidth usage is a significant concern also when high data rates are available. Mobile internet connections are usually metered and contracts can have a limited budget of data transfer included and additional bandwidth usage must be paid. When the data rate is reduced by means of high compression, more information within the given bandwidth constraint can be exchanged. For example, we can increase the sampling frequency of sensors to report information more often, hence we are able to react on changing conditions more proactively.

1.2 Problem statement

State of the art compression methods such as GZip or Deflate can be generally employed to compress messages. These methods are often used in combination with batching, that increases compression ratios at the cost of latency. So far compression is

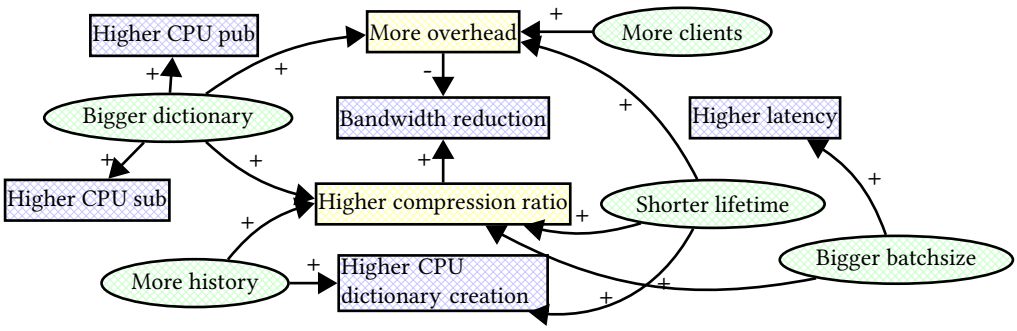


Figure 1.2.1: Relationships among all compression related variables and cost

a tradeoff between CPU, compression ratio and latency. Shared Dictionary Compression (SDC) can go farther in terms of compression ratio. However, SDC requires a dictionary to be generated and disseminated prior to compression, which introduces additional computational and bandwidth overhead. Hence, compression in pub/sub systems is a tradeoff between CPU, compression ratio, overhead and latency. If we use SDC, the tradeoffs can be adjusted more fine-grained. Figure 1.2.1 shows the variables we can adjust in green bubbles and the effect on the cost in rectangles. For example, a large dictionary is beneficial for higher compression, but also imposes high CPU load on the publishers for compressing the messages and sharing the dictionary between publishers and subscribers amounts to higher overhead. Batching messages together increases the redundancy within the batch, hence a higher compression ratio can be achieved, but batching multiple messages together increases the latency. Sharing the dictionary with more clients amounts to a higher bandwidth overhead, which lowers the overall bandwidth reduction. Creating the dictionary from a larger window of historic messages improves the compression ratio, but imposes higher CPU cost for creating the dictionary.

Our work contributes to the overall goal of reducing the required bandwidth in pub/sub systems by contributing to the understanding of these variable influences and tradeoffs and proposing approaches to automatically find good variable combinations. We focus on the following research questions:

1. How can we extend pub/sub systems to support shared dictionary compression in a user hidden way to enable higher bandwidth reductions than the state of the art?

2. Can we find good parameter configurations in an automated way for compression

using SDC to amortize the imposed overhead?

3. Since SDC imposes an overhead, which is amortized by enabling high compression ratios, can we adapt to varying distributions of message publications so that no additional overhead is imposed on publishers with low message rates?

1.3 Approach

Compression is a well-known technique for reducing data usage at the expense of higher computational costs. Off-the-shelf compression algorithms, such as DEFLATE [18], work well when redundancy can be leveraged. But single messages disseminated by the pub/sub system typically do not exhibit a lot of redundancy. However, several subsequent messages can exhibit redundancy like the schema. If JavaScript Object Notation (JSON) or eXtensible Markup Language (XML) are used as a message serialization format, lots of redundancy is shared between messages through the schema. In message formats like Comma-separated values (CSV) or Protobuf (a binary message serialization format), there is not a lot of redundancy introduced through the serialization format. Nevertheless there are certain content biases, such as trending topics in twitter or multiple sensors measure similiar values (e.g., temperature), which also have certain redundancy over time. We propose SDC to capture the redundancy in a dictionary, further share the dictionary to the publishers and subscribers so that publishers can use the dictionary to compress messages and subscribers to uncompress the messages. To capture the content biases, we propose a Dictionary Maintenance Algorithm (DMA), which observes the message stream and creates a new dictionary to capture these biases.

First, we sketch the idea and propose an architecture to enable SDC for pub/sub since it is one of the first works in that area. We propose a novel and lightweight protocol for pub/sub which employs a new class of broker, called Sampling Broker (SB), to support SDC. Our solution creates, and disseminates dictionaries using the SB. Dictionary maintenance is performed regularly using an adaptive algorithm. We introduce a basic heuristic which configures most of the parameters needed for creating dictionaries in a semi-automatic way (called ADAPTIVE). We evaluate the approach using a simulation and a prototypical implementation. The evaluation of our proposed design shows that it

is possible to compensate for the introduced overhead and achieve significant bandwidth reduction over DEFLATE.

Then, we focus on a fully automatic approach called PREDICT, which makes manual configurations obsolete and optimizes for good variable combinations over time while reducing the CPU for publishers and subscribers. Our evaluation is done using a simulation and a prototypical implementation. We use many streaming datasets from social media, public transport, exchanges and sensors to evaluate our approach

In the final part, we extend PREDICT to work with a distributed pub/sub system with a graph overlay, called TAPD̃(Topology Aware PREDICT). Our approach aimes to introduce low overhead for publishers sending messages at low rates while keeping high compression ratios for publishers with high message rates. We achieve this by keeping dictionaries for selected publishers active for a longer time. This degrades the compression ratios to some extend but also reduces the imposed overhead which amounts to overall higher bandwidth reductions for publishers. Additionally we employ recoding in the overlay. Overall, this approach is more bandwidth efficient since less overhead is introduced.

1.4 Contributions

The main contributions of our work regarding compression in pub/sub systems are:

I. Introduction of SDC in pub/sub systems by extending the pub/sub design to support compression as a core component

II. We introduce a new zero configuration dictionary maintenance algorithm, called PREDICT, which derives all parameters using curve fitting and machine learning.

III. An extensions of PREDICT, called TAPD, which reduces the overhead in distributed broker overlay and increases the overall bandwidth efficiency.

Next we break up the main contribution in the following sub-contributions:

Shared Dictionary Compression in Publish/Subscribe Systems

We introduce the foundation for how pub/sub can be extended with SDC. We extend the broker overlay with an additional role, called the Sampling Broker (SB). Then, we evaluate many parameter combinations and motivate why continuous dictionary maintenance is necessary to maintain high bandwidth reductions. Furthermore, we introduce a simple heuristic for dictionary maintenance called ADAPTIVE. This approach is partially manually configured for each topic and uses a heuristic to adapt some of the parameters.

Our contributions in detail are:

i. We present our solution *Simple SDC for pub/sub* (SPSS): a fault-tolerant pub/sub design with SDC for efficient publication traffic reduction.

ii. We introduce the use of *sampling* brokers for generating, maintaining, and disseminating dictionaries.

iii. We provide an adaptive algorithm for dictionary maintenance, which considers the benefits of generating a new dictionary vs. the dictionary sharing overhead with varying parameters.

iv. We evaluate our algorithm using several real world datasets by comparing to DEFLATE. Our evaluation is implemented on top of an MQTT broker using a smartphone client.

PreDict: Predictive Dictionary Maintenance for Message Compression in Publish/Subscribe

The main shortcoming of our previous approach ADAPTIVE is that it has to be manually configured for each topic and the heuristic ends up with too large dictionaries. Large dictionaries impact the overhead when shared between the clients and lead to high CPU cost for publishers when compressing the messages, see Figure 1.2.1. Furthermore, the manual configuration is specific to the content of the messages. In cases of large brokers handling lots of different topics, this can become a operational burden.

We propose PREDICT, a DMA for cloud-based pub/sub systems, which determines all parameters in an automatic way using curve fitting methods and machine learning. Furthermore, we take the overhead introduced to clients into account, based on a system model for cloud-based pub/sub systems. In our evaluation, we show with a wide variety of datasets that we can achieve bandwidth reductions with different publisher-to-subscriber-ratios without any manual configuration and our approach is on par with the best permutations of manually configured approaches.

In detail, our list of contributions are:

i. An in-depth analysis of the connection between the variables, the computational cost and bandwidth reduction, which allows us to derive beneficial variable combinations in the DMA.

ii. A new self-adapting dictionary maintenance algorithm, called PREDICT, which uses small dictionaries, feature extraction methods and machine learning to reduce the computational costs for the publishers and subscribers while achieving high bandwidth reductions.

iii. An in-depth comparison of multiple DMAs and off-the-shelf compression using DEFLATE and Diff algorithms on large streaming datasets collected from social media, currency exchanges and other sources. We compare the computational cost of the algorithm itself, the additional computational cost for publishers and subscribers and the bandwidth reduction including the introduced overhead.

iv. Additional evaluation of PREDICT in combination with batching and a prototypical implementation based on Apache Pulsar.

TaPD: Topology-aware PreDict

TAPD extends our work for cloud-based pub/sub systems to distributed broker overlays. While PREDICT performs well in scenarios where each publisher sends equal amounts of messages, in cases where some publishers send many and lots of publishers just a few messages, many publishers could be imposed by bandwidth overhead which cannot be

amortized by high compression ratios on messages. TaPD proposes a mechanism to prolong dictionaries for publishers, fallback to Deflate and recoding of messages in the network. This allows us to have multiple dictionary versions active at the same time. The main focus of this approach is to reduce the overhead for publishers which send messages at low message rates.

The contributions of this approach are:

i. TaPD, a way to make PreDict aware of the overlay and to further reduce the required bandwidth in cases where the message rates of publishers are distributed exponentially.

ii. Pseudocodes and specifications of all algorithms for a real world implementation.

iii. A evaluation of TaPD using a simulation over various large streaming datasets collected from social media, exchanges, and other sources.

Publications

This thesis contains material from two papers, a poster and a demo in addition to several unpublished works. Parts of the content and contributions of this work have been published in:

- C. Doblander, T. Ghinaiya, K. Zhang, and H.-A. Jacobsen. "Shared Dictionary Compression in Publish/Subscribe Systems." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 117–124. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933308

- C. Doblander, A. Khatayee, and H.-A. Jacobsen. "PreDict: Predictive Dictionary Maintenance for Message Compression in Publish/Subscribe." In: Middleware '18. Rennes, France: ACM, 2018. ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274822

- C. Doblander, K. Zhang, and H. A. Jacobsen. "Publish/Subscribe for Mobile Applications Using Shared Dictionary Compression." In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 775–776. DOI: 10.1109/ICDCS.2016.70
- C. Doblander, S. Zimmermann, K. Zhang, and H.-A. Jacobsen. "Demo Abstract: MOS: A Bandwidth-Efficient Cross-Platform Middleware for Publish/Subscribe." In: *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference*. Middleware Posters and Demos '16. Trento, Italy: ACM, 2016, pp. 27–28. ISBN: 978-1-4503-4666-5. DOI: 10.1145/3007592.3007607

1.5 Organization

In the following chapter, Chapter 2, we introduce the relevant background. Chapter 3 presents relevant related work in our field. Chapter 4 introduces the extended design for pub/sub and presents ADAPTIVE, a heuristic for dictionary maintenance. Chapter 5 presents a zero configuration dictionary maintenance algorithm which derives all variables from curve fitting and machine learning. Furthermore, we provide additional evaluation of PREDICT using an implementation on top of a real world distributed pub/sub, Apache Pulsar. Additionally, we evaluate batching. Chapter 6 shows the efforts towards making PREDICT aware of the overlay, called TAPD. In Chapter 7 we conclude and present future work. In the appendix we introduce the evaluation datasets and additional results.

Chapter 2

Background

First we introduce pub/sub and distributed pub/sub systems. Then we introduce the relevant background regarding compression and dictionary compression.

2.1 Publish/subscribe systems

Pub/sub is an effective method to disseminate data while decoupling data sources and sinks [8]. Event sources publish notifications (also called publications) on a topic and brokers route the notifications to interested subscribers. In a topic-based model, subscribers express their interest in a topic. The events belonging to that topic are then routed through the broker overlay network to the interested subscribers. In a content-based system, subscribers can additionally express predicates in their subscriptions which further filter publications belonging to a topic, by comparing those predicates with attribute-value pairs embedded in the publications.

2.2 Distributed publish/subscribe systems

Pub/sub is used to communicate within the datacenter [23] and also with the edge devices such as smartphones or sensors. Many pub/sub systems form a broker overlay [5, 6, 7, 24]. The message rates within the overlay not only depends on how many messages each publisher produces but also to which brokers the publishers and subscribers are connected. Some approaches allow for some kind of bandwidth economy through routing messages in the overlay [25, 26, 27, 28], but these approaches do not reduce the size of the messages.

2.3 Lossless compression

Compression schemes, like GZip [29], Snappy or Deflate [18], use a sliding window and leverage the redundancy within that window to compress a notification. The redundancy within a window tends to be low compared to the redundancy between two subsequent notifications. Therefore, we argue that SDC reduces bandwidth beyond what is possible with traditional compression methods.

DEFLATE [18] uses a combination of LZ77 and Huffman coding. GZIP [29] uses DEFLATE as the compressed data format. GZIP appends a header and a CRC32 checksum to Deflate. When small notifications are compressed, it is possible that the compression gains do not outweigh the additional large header. Thus, we employ DEFLATE, not GZIP, as a baseline for comparison.

2.4 Dictionary compression

SDC leverages similarity between notifications to improve compression ratios. One of the first papers [30] about dictionary-based compression achieves 60%-70% compression for English documents using a small dictionary. This idea is further explored in [31] and subsequently extended in [32]. In this paper, we use SDC to refer to a combination

of dictionary-based compression and multiple passes of Huffman Coding. Using this method, references to the dictionary are represented very efficiently.

For dictionary-based compression, we use the open-source library FemtoZip [33], which we modified for our approach. FemtoZip is a versatile library that offers many compression algorithms and that can also be used to create dictionaries. In this paper, when we refer to a dictionary, we mean a combination of a dictionary and a Huffman table as defined in the *FemtoZipCompressionModel*. An alternative would be gzip [29], which would also support setting a dictionary. We do not use gzip dictionary compression because the maximum size of a dictionary is 16 kB and gzip adds checksums, which unnecessarily increases the sizes of the messages. Furthermore, gzip uses the dictionary only for the first sliding window.

The dictionary creation with FemtoZip works as follows. A dictionary is sampled from a set of messages by computing the Longest Common Substrings (LCS), ranking the substrings according to their occurrences and reducing overlap by merging the substrings. By ranking the substrings, the most valuable entries tend to be placed at the end. Finally, the dictionary is truncated from the beginning to a certain size to ensure that only the highest ranked substrings are present in the final dictionary. Then, the Huffman tables are created from a set of messages and the truncated dictionary. In this way, references to dictionary entries can also be present in the table. The dictionary and Huffman tables together form a compression model that is shared with the clients and used by publishers to compress messages and by subscribers to decompress messages. A dictionary can be thought of as a blob prepended to the message and entries are referenced relative to a given position. As an example, a reference *<-10,4>* at position 3 of the message would go back to the *-7* position of the dictionary and take the next *4* bytes. Although many workloads are text based, the dictionary is not limited to strings and can contain arbitrary byte sequences. Because the dictionary entries are merged together and because the Huffman table is built from the messages that are compressed with the dictionary, the symbols of the Huffman table also weigh the references to the dictionary accordingly. This is efficient for compression but makes it difficult to incrementally update the dictionary. A substitute for incrementally updating the whole compression model would be to compress the new model using the previous model. Because this would increase the corner cases of the protocol (e.g., high churn of $\mathbb{S}$ and $\mathbb{P}$) we did not

consider it. But we compress the compressmodel using DEFLATE, which reduces the size by ≈ 40%.

2.5 Example dictionary compression

As an example, Listing 2.5.1 shows the dictionary created from the Section 1.3 which is truncated to the most valuable 300bytes of data.

Listing 2.5.1: Example dictionary from introduction

```
u disseminate evaluation introduced that the i. Our e for e the ed
    overhead compression for pub/sub redundancy in, we for publishers
     content biases, pub/sub system ing dictionaries bandwidth
    reduction publishers and subscribers, which using a simulation
    and a prototypical implementation. approach
```

Listing 2.5.2: Example sentence

```
We evaluate the approach using a simulation and a prototypical
    implementation.
```

When we compress an example sentence, see Listing 2.5.2 using the dictionary in Listing 2.5.1, we end up with the following compressed sentence, see Listing 2.5.3. As an example, "<-289,9>" refers to "e evaluation ", the "e", which completes the first word is taken from the end of "disseminate". The dictionary can be though of a prepended blob of data and each position in the blob is referenced relative.

Listing 2.5.3: Example sentence compressed

```
W<-289,9><-252,6><-25,9><-88,53>
```

Chapter 3

Related Work

To the best of our knowledge, there is no paper that explores the use of SDC or compression of individual messages within pub/sub. Traditional compression such as GZIP or DEFLATE can be universally employed, hence do not need a specific extension to the pub/sub system. Consequently, we extend the scope of our related work to general bandwidth reduction mechanisms for pub/sub. Then we look at delta compression [34] and deduplication [35]. Finally we look at related fields in communication protocols (HTTP) and the use of dictionaries in databases.

3.1 Bandwidth reduction in pub/sub

Other approaches for reducing bandwidth in pub/sub consider user-defined aggregation functions, which assist application programmers in implementing efficient context propagation [36]. One main goal of our approach is to remove any operational complexity via automatic configuration; hence, our approach can easily be adopted in existing pub/sub systems. Routing mechanism in pub/sub are used to provide bandwidth economy [37]. That way messages can be routed through non-congested links at different speeds.

3.2 Delta encoding and deduplication

REAP [34] presents a data differencing algorithm. MultiDiff performs significantly better than simple delta encoding. The patch can reference multiple other preceding notifications. The protocol uses windows of notifications, which can be referenced by the diff. To address out-of-order notifications, the delta is always computed to notifications within a certain preceding window; the subscribers are then instructed to cache this window. The notifications in the window can also be diffs of preceding notifications; hence, at-least once guarantees are mandatory. Our work targets a different kind of environment. Many publishers can send messages on a single topic. Publishers would need the messages from other publishers to compress messages. In addition, our approach should not require ordering guarantees that are difficult to fulfill in cyclic pub/sub broker overlays. In content-based pub/sub, this approach has another weakness, since subscribers to the same topic receive different publications due to their unique predicates. In this case, the deltas would have to be re-encoded between communicating peers introducing additional computational overhead.

We are unable to compare with their results because the utilized datasets are not publicly available. Their workloads are mainly NATO-specific XML messages or Simple Object Access Protocol (SOAP) messages.

Deduplication algorithms are used to reduce bandwidth usage in the context of database replication [35]. In delta compression [38], one notification is described in terms of another notification.

For comparison, we implement an approach using delta encoding with VCDIFF as a representative approach in this area.

3.3 Dictionary compression in HTTP

Dictionary based compression is proposed for HTTP [39]. SDCH is a proposal for a HTTP/1.1-compatible extension to enable inter-response data compression. LinkedIn

reported [40] an average bandwidth reduction of 24% on top of GZIP. Brotli [41] also contains a static dictionary sampled from a multi-lingual web corpus to improve compression ratios specifically for web pages.

While the techniques to create a dictionary are similar, we optimize towards pub/sub systems. Furthermore, we look at topologies of brokers decoupling publishers from subscribers. Here SDC is used only between client and server.

3.4 Dictionary compression in databases

Dictionary compression is also widely used in databases [42] to achieve performance gains for I/O-intensive queries. Percona Server for MySQL [43] supports dictionary compression for columns. Dictionary compression is beneficial in the back end databases of ERP systems because more than half of the columns are string columns [44]. Databases also often use so-called lightweight compression schemes [42]. These schemes allow querying on compressed data or allow the dictionary to be used as an index. HBase uses a dictionary to compress the write-ahead log [45]. The implementation available in HBase captures the most recently used messages in a dictionary. Entries are evicted when the maximum number of entries is reached; in the worst case, 160 MB is used.

BigTable [46], a distributed storage system for structured data, uses a custom compression scheme, which includes sampling a dictionary. First, *Data Compression Using Long Common Strings* [31] is used. In the second pass, a small window is used, similar to Deflate. They report a 10-to-1 reduction in space. The main reason is that web pages from a single host share large amounts of boiler plate which is identified in the first run and included in the dictionary. Dictionary-based compression is also used in main memory column stores [47]. The dictionary can also be used to index and to rewrite queries. This approach reduces the memory consumption of the database.

The main differences among the works on databases and our work, which targets pub/sub, are the different costs and optimization goals. In compression for pub/sub, the goal is to reduce the overall bandwidth between publishers and subscribers, including all the related overhead, which has different scaling properties than the work targeting

databases. A 160 MB dictionary would be too large for IoT sensors, although it would be beneficial for bandwidth reduction. Our work concerns the trade off for enabling dictionary-based compression in pub/sub and how to determine the sweet spot in terms of bandwidth reduction in an online and automated manner. As an example, we need to reduce the size of the dictionary to reduce the overhead of sharing a dictionary; smaller dictionaries simultaneously also reduce the compression ratio. The dictionaries that we use are ≈ 10 kB or smaller, depending on the message content of the stream. Furthermore, sharing a dictionary in topologies that have many more publishers than subscribers is challenging because the dictionary is difficult to amortize when only a few messages are sent per publisher.

Chapter 4

Shared Dictionary Compression in Publish/Subscribe Systems

Popular compression methods, such as GZip or Deflate can generally be applied for moderate performance gains. In this chapter we demonstrate how higher bandwidth reductions can be obtained by employing Shared Dictionary Compression (SDC) in pub/sub.

First, we give an overview of our approach in Section 4.1. Then, in Section 4.2, we present our design for pub/sub using SDC. We then compare our approach to the state of the art in Section 4.3. In Section 4.3.5 we show the results of our prototypical evaluation on top of Message Queuing Telemetry Transport (MQTT).

Parts of the content of this chapter are published as [19, 21, 22].

4.1 Overview

Our approach adds a new type of broker, subsequently called *sampling broker*, to the pub/sub design. A Sampling Broker (SB) is responsible for sampling notifications to

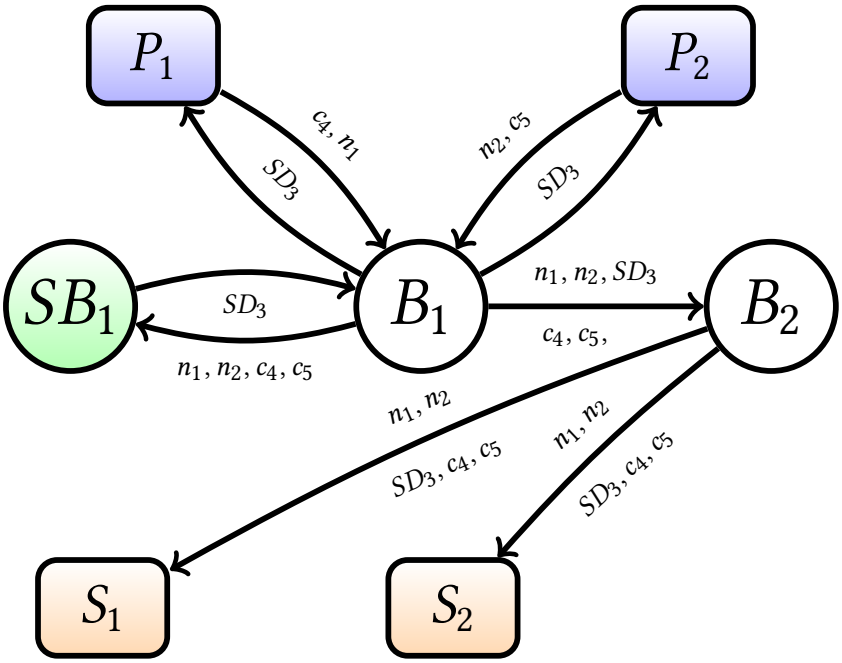


Figure 4.1.1: Notification delivery in pub/sub overlay

create dictionaries, maintenance of said dictionaries over time and disseminating them in the overlay network. An adaptive algorithm is employed for periodical maintenance which creates a new dictionary with specific parameters when it is beneficial to do so over keeping an older version, and fault-tolerance is provided by sending the dictionaries to a caching service. Figure 4.1.1 shows a high level overview of a broker overlay extended by the sampling broker SB_1. Publishers $P_{1,2}$ send messages to the subscribers $S_{1,2}$. After the notifications $n_{1,2}$ are published, the adaptive algorithm decides that the bandwidth can be reduced by employing a dictionary. A new dictionary SD_3 is sampled and sent through the pub/sub overlay. The dictionary can then be used by $P_{1,2}$ to compress the notifications $c_{4,5}$ and $S_{1,2}$ can uncompress the notifications. The figure is used as an example throughout the whole paper.

To motivate and evaluate our work, we consider the following representative use case. We imagine a scenario where mobile phones use an application to communicate with a back end, which provides various services. Upon starting the application on a smartphone, a map centered around its current location is displayed. Additionally, the application subscribes to nearby notifications in order to refresh its view of the map. All user-facing events are sent via the mobile phone network. While fast mobile connections are readily

available in city centers, the same cannot be said for rural areas. Providing the user a real-time view of the map requires a high incoming event throughput rate, which may not be desirable if the phone is operating on a metered data plan. We evaluate this use case using the dataset of the DEBS 2015 Grand Challenge [48], which contains taxi trips records in NYC for a year. We measure the bandwidth savings when these events are sent to a mobile phone that displays the information on a map in real-time.

4.2 SDC in publish/subscribe

Our approach, called Simple Shared dictionary compression for Pub/Sub (SSPS), introduces several new components. We introduce a new broker class, called Sampling Broker (SB), which can be a separate broker instance or a role assigned to an existing broker. Additionally, we introduce a caching service (CS) to provide fault tolerance. The responsibility of a SB is to create dictionaries, monitor the compression ratio, and maintain the dictionary over time. The CS caches the dictionaries, such that newly joining subscribers or publishers can acquire required dictionaries.

Dictionary sampling

To enable SDC in pub/sub, we first have to sample notifications to create dictionaries. In SSPS, we propose a single dedicated SB per topic, with a dictionary generated for each. A SB carries the main computational load of our approach, which is the sampling of publications. To balance the computational load of many topics, consistent hashing [49] of the topics could be used to assign the SB to a physical instance. A SB subscribes to the topic and accumulates the notifications in a ring buffer with a fixed size (B_{size}). A predefined hash function is used to generate the dictionary based on the stored notifications.

Sharing of the dictionary

After the dictionary is sampled, it is added to the cache and published on the corresponding dictionary topic. Note that all publishers and subscribers sending and receiving data on a topic are subscribed to the corresponding dictionary topic and will receive the appropriate dictionary. After a fixed expiry timestamp T_{exp} attached to each dictionary elapses, no publisher is allowed to compress a notification using this dictionary. L_{max} is a predefined time interval, which is well above the worst case maximum publisher-subscriber latency and the maximum clock skew within the network. After $T_{exp} + L_{max}$ the subscriber is also allowed to dismiss the dictionary. Each dictionary has an identifier, which is referenced to in the compressed notifications.

Caching service

The caching service provides fault tolerance for the dictionaries. Equation 4.2.1 shows how long every SD is cached. The CS is essentially a fault-tolerant key-value store.

$$T_{exp} + 2 \times R \times L_{max} \tag{4.2.1}$$

Continuous dictionary maintenance

Before a dictionary expires, the SD has to either increase the expiry time or sample a new dictionary. The SB is only allowed to increase the expiry time until $T_{exp} - L_{max}$. First, the expiry in the caching service is increased, then a new notification on the dictionary topic is published, which increases the expiry of the SD at the publishers and subscribers.

New publishers

We have to consider two cases. The first occurs when a publisher creates a new topic and starts publishing, the second case occurs when the publisher starts publishing on

an existing topic. In the first case, the publisher starts sending notifications on the topic and also subscribes to the dictionary topic. When the SB has created a dictionary, it will be published on the dictionary topic which belongs to the notification topic. The publisher always compresses the notifications using the dictionary whose expiry is the farthest in future. In the second case, the publisher can start sending the notifications without compression and eventually acquire the SD from the CS. No additional latency is incurred since the publisher can always publish uncompressed notifications.

New subscribers

When a subscriber subscribes to a topic, it also issues a subscription to the corresponding dictionary topic, which is calculated using a predefined hash function on the original topic. When the subscriber receives a compressed notification and the dictionary is not available, the dictionary has to first be acquired from the caching service. In this case, an additional latency of L_{max} can occur when receiving the first message.

Dictionary maintenance

The dictionary has to be resampled to maintain high bandwidth savings. Hence, we propose an adaptive algorithm, which probes the notifications to detect if a new dictionary would improve the compression (see Section 4.2.2). Each dictionary has an identifier, which is unique within a topic and within the timespan $T_{exp} + 2 \times L_{max}$. In our experiments, a single byte is used to represent this ID. Figure 4.1.1 shows an example of the algorithm in practice. The subscript numbers on the notifications denote the sequence. The assumption is that the subscribers $S_{1,2}$ are already connected, but no notification have been sent so far. The publishers $P_{1,2}$ start to publish the notifications $n_{1,2}$. SB_1 and $S_{1,2}$ receive the notifications. SB_1 begins to sample the notifications and publishes a new SD_3 on the corresponding dictionary topic. $P_{1,2}$ and $S_{1,2}$ are subscribed to the topic and receive D_3. From now on, $P_{1,2}$ can publish compressed notifications $c_{4,5}$. The subscribers $S_{1,2}$ can decompress the notifications using the cached dictionary.

4.2.1 Analysis of overhead

Dictionary publication

The dictionary size is bounded to a multiple of the average uncompressed notification size. In our evaluation, we consider multipliers of up to 21 of the original notification size + Huffman Trees. All publishers $|P|$ and subscribers $|S|$ have to receive the dictionary. Equation 4.2.2 shows the total bandwidth used for spreading a new dictionary.

$$(|P| + |S|) \times SD_{size} \tag{4.2.2}$$

Memory consumption of dictionaries across the overlay

There can be situations when multiple dictionaries are active at the same time. This situation occurs when the bandwidth savings of the new dictionary pay off faster than waiting for the expiry of the old dictionary. This depends on the behavior of the dictionary maintenance algorithm and how it adapts to changes in the content of publications. In general over time, the sampling frequency and estimation of T_{exp} should be stable. Hence the additional memory consumption of the SD is on the publisher and subscriber SD_{size}.

Dictionary sampling time

Sampling a dictionary is at worst an $O(N^{3/2})$ operation [31]. For practical workloads, the average case is $O(N)$. N stands for the total number of bytes of the notifications in the buffer, $\sum_{i=0}^{bufferlength} = |N_i|$. Our experiments confirm this observation. Once the dictionary is sampled, it is limited to a specific size. The cost of trimming a dictionary to a certain size is negligible compared to the sampling time.

4.2.2 Adaptive algorithm

We propose an adaptive algorithm to choose $SD_{multiplier}$, B_{size}, and T_{exp}. Our proposed heuristic is conservative in the sense that a new dictionary is only spread when it is certain that the cost of spreading can be amortized.

Overview

Every time a dictionary expires (T_{exp} is reached), the algorithm first computes the amount of bandwidth reduction that can be achieved if a new dictionary would be published. If the gain from a new dictionary is higher than a certain threshold, the dictionary is published, otherwise the existing dictionary is prolonged. In addition, the algorithm changes the parameters of the dictionary creation ($SD_{multiplier}$ or B_{size}) for the bandwidth evaluation at the next expiry time.

Bandwith reduction

To calculate the amortization time, we need to estimate the bandwidth reduction (BR) of a new dictionary. BR is estimated by splitting the recorded messages into a training and validation set. Splitting the data into two independent sets is a technique known from machine learning. In case the set is not split, the calculated bandwidth reductions tend to be not reproducible similar to when a machine learning method is validated on the training data. A dictionary is sampled on the training set and the bandwidth reduction is derived by comparing the uncompressed validation set with the compressed validation set. We decided to take 70% for training and 30% of the notifications for validation.

We can calculate the bandwidth reduction (BR) of the new dictionary using the uncompressed size of the notifications TB and the size of the compressed notifications CTB using Equation 8:

$$BR = 1 - \frac{CTB}{TB} \tag{4.2.3}$$

Amortization time

The time needed to amortize the dictionary $T_{amortize}$ is calculate using the current rate (R) and the estimated bandwidth reductions (BR). The rate, see Equation 4.2.4, is derived by dividing the total size of the notifications by the time span between the first and last message in the buffer.

$$\begin{aligned} TS &= t_{buffersize} - t_0 \\ R &= \frac{\sum_{i=0}^{buffersize} |n_i|}{TS} \end{aligned} \tag{4.2.4}$$

The amortization time then takes the size of the dictionary and the bandwidth reduction into account using the estimated rate R (see Equation 11):

$$T_{amortize} = \frac{|SD|}{R \times BR} \tag{4.2.5}$$

Dictionary parameters

When the dictionary is sampled, two parameters are taken into account: $SD_{multiplier}$ and B_{size}. When use of a dictionary is prolonged, the $SD_{multiplier}$ and B_{size} parameters are increased and taken into account for the next evaluation. It could be that an increase of the dictionary size or sampling window size result in greater bandwidth reduction. Every time the configuration of a dictionary is changed, the counter $adaptations$ is increased. After a certain amount of tries with increasing variations, B_{size} is no longer increased since the computational costs become too high. This parameter should instead be chosen according to the average notification size.

Expiry time

Equation 4.2.6 shows how the expiry time of a dictionary is calculated. A dictionary should at least amortize 10×, this value being chosen experimentally. Additionally, we added a factor based on how often different parameters of the dictionary were chosen. Each time the parameters of the dictionary are changed, the adaptation counter is increased. The computational cost of the dictionary sampling grows with increasing B_{size} and the compression cost of the publishers increase when the $SD_{multiplier}$ is increased.

$$T_{exp} = T_{amortize} * adaptations^3 * 10 \tag{4.2.6}$$

4.3 Evaluation

We present three experiments. First, we evaluate the compression potential and computational cost of SSPS using varying permutations to find the best configuration. Then, we present an evaluation of the adaptive dictionary maintenance algorithm and discuss the trade-off between the computational cost and bandwidth reduction in view of the practical limits assessed in the first experiment. Finally, we present a practical use case evaluation using an experimental implementation on top of MQTT [50].

4.3.1 Datasets

Compression performance depends on the redundancy within a notification and between notifications. For that purpose, we took several real world datasets with varying degrees of redundancy. The DEBS15 dataset contains taxi trips from New York. The original dataset is published in Comma-separated values (CSV). To study the effect of different formats we converted the notifications into JSON, XML and Google Protobuf. Google Protobuf [51] generates an efficient binary notification based on a field description. The Twitter dataset is acquired from the Twitter API and contains Tweets including metadata from New York. We extracted only the content of the tweets to measure the compression

performance on a highly variable text without any schema overhead. The Air Quality dataset comes from the NYC Open Data Portal [52]. This is used as an example of an IoT dataset, which includes many sensor readings. The EPEX dataset is extracted from the energy spot market auctions from the European Power Exchange. We use this dataset to test our solution with financial data, which mostly contains numbers and some repetitive information like contract types.

4.3.2 Compression potential using SDC

We emulated the connection of a publisher to the broker and evaluated multiple configuration dimensions. The size of the buffer B_{size} is based on how many preceding notifications the dictionary has sampled. The window W_{size} denotes how often the buffer is sampled. The maximum size of the dictionary is set to a multiple of the average notification size. Equations 4.3.1,4.3.2 show the permutations. As an example, the configuration W_{300}, B_{100}, SD_2 means that every 300 notifications, a dictionary is built using the last 100 notifications and the dictionary size is limited to 2× the average notification size. At least 20k notifications are employed from each dataset.

$$size = \{50, 100, 200, 300, 500, 800, 1300, 2100, 3400, 5500\} \tag{4.3.1}$$

$$multiplier = \{0.3, 0.5, 1, 2, 3, 5, 8, 13, 21\} \tag{4.3.2}$$

The evaluations are conducted on a machine with 4×Intel Xeon CPU E7-4850 v3 @ 2.20GHz.

The emulation uses the library FemtoZip [33]. FemtoZip is a SDC library which can be used to build dictionaries. We used the class `FemtoZipCompressionModel` to generate the dictionaries from the notifications. The compression model creates also an optimal Huffman tree. The dictionary size is the sum of the size of both components. The

Dataset	Mean size	Best $B/W/M$	SD size	SD OH	DEFLATE (size)	SDC (size)	DEFLATE (%)	SDC (%)
DEBS15-XML	711.41	2100/5500/21	22460.0	5.08	332.36	85.18	53.3	88.0
DEBS15-JSON	530.41	2100/5500/21	18659.0	4.39	289.68	83.99	45.4	84.2
DEBS15-CSV	191.41	500/5500/21	11540.0	3.1	125.2	72.75	34.5	62.0
DEBS15-PB	174.96	500/5500/21	11183.0	3.03	161.02	83.64	8.0	52.2
EPEX Spot	94.72	1300/3400/3	7810.0	3.3	73.79	24.59	21.6	74.0
AirQ-XML	536.53	3400/5500/21	18798.0	4.42	321.52	77.43	40.0	85.6
Twitter (meta)	2995.74	100/5500/21	69739.0	13.68	1241.52	438.74	58.7	85.4
Twitter (tweets)	84.87	5500/5500/21	9304.0	2.69	80.19	53.44	5.7	37.0

Table 4.3.1: Shared Dictionary compression vs. Deflate

dictionary part is limited to a multiple of the average notification size, see Equation 4.3.2.

The heatmaps in Figure 4.3.1 show the sweet spot of each dataset when the update frequency of the dictionary is set to 5500. The y-axis shows the dictionary multiplier, the x-axis the buffer size. The overall tendency is that bandwidth savings increase as more notifications are sampled. However after a certain size, it does not make sense to further grow the buffer. The same applies to the dictionary. While in most cases the best bandwidth savings are achieved with the biggest dictionary sizes, at a certain threshold the gain in bandwidth savings are not substantial.

Table 4.3.1 shows the best configuration for SDC. Column *Mean size* is the mean size of the notifications. Column *Best B/W* denotes how long the buffer should be and after how many notifications the dictionary should be refreshed. Column *SD size* shows how many additional bytes the average overhead per message is and column *SD OH* shows the introduced overhead in average per notification. The columns DEFLATE *(size)* and *SDC (size)* show how big in average the message is including the average overhead of the protocol. The last two columns show how big the bandwidth savings are. The overhead is largely introduced by the dictionary. Note that we added one byte per message for the protocol overhead to denote if the message is uncompressed or which dictionary identifier is used. The DEFLATE evaluation has no dictionaries and thus does not introduce any protocol overhead to the notifications.

An interesting observation is that there is nearly no difference in the bandwidth usage between XML, JSON and Protobuf when SDC is used. This is not the case when DEFLATE is used. The schema overhead and common attribute combination are promoted in the shared dictionary, which is similar in all schema variations. Therefore, the developer can choose the notification format which is most convenient without worrying about data

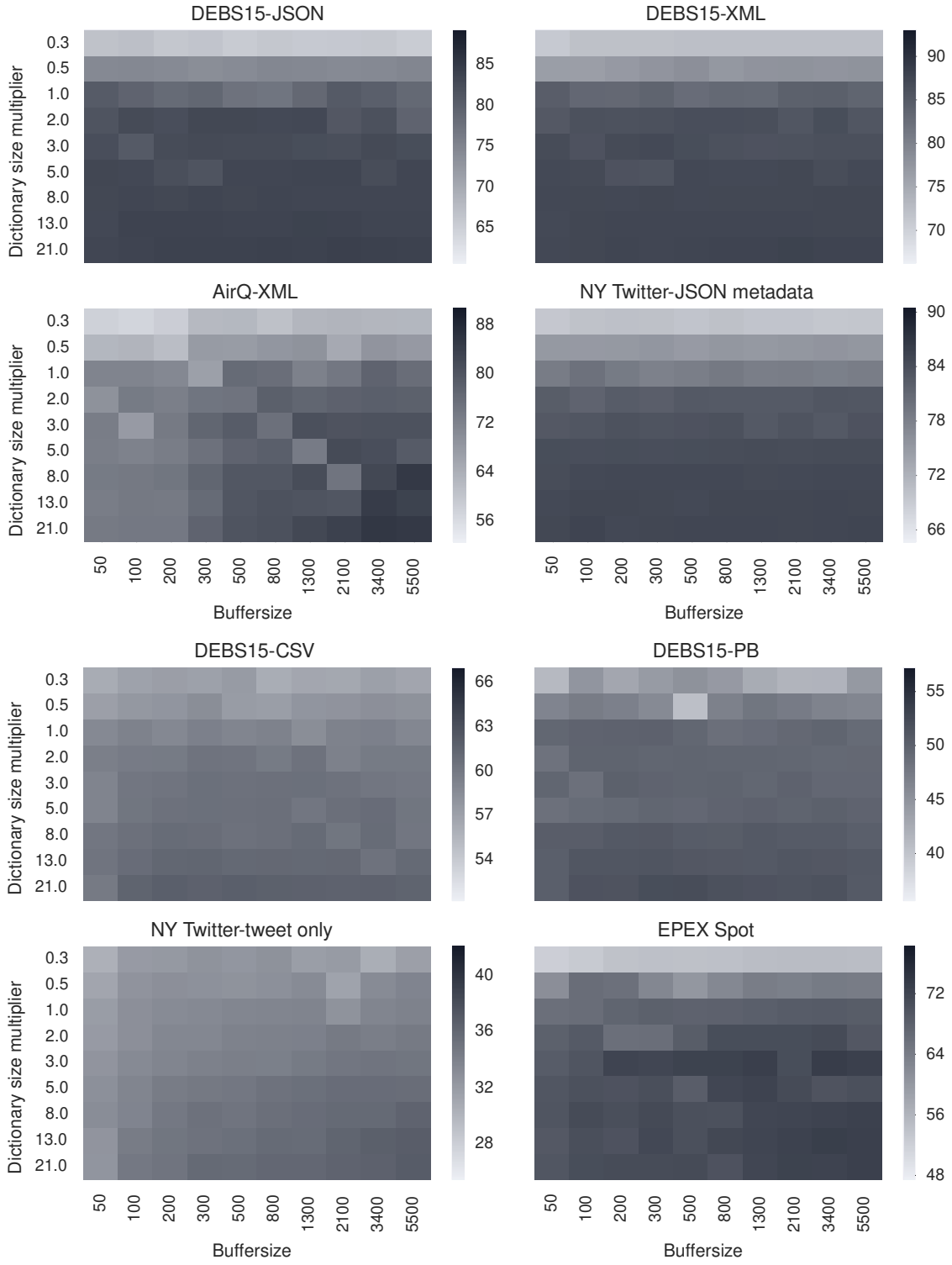


Figure 4.3.1: Buffer size vs. dictionary multiplier, update freq. set to 5500

size.

The overall tendency in our evaluation shows that it makes sense to sample bigger dictionaries and exchange them less frequently. The best dictionary multiplier in the EPEX Spot market data stream is only 3. This has to do that there is a limited amount of variation in the text properties, while the rest of the properties is numerical.

Figure 4.3.2 summarizes the performance comparison between SDC and DEFLATE. One observation is that small notifications are less compressible using DEFLATE, since there is less repetition within a notification. Tweets also have less repetition within a single tweet, hence the DEFLATE performance is low. Using SDC, where common words and tags are promoted to the SD, the bandwidth savings are higher.

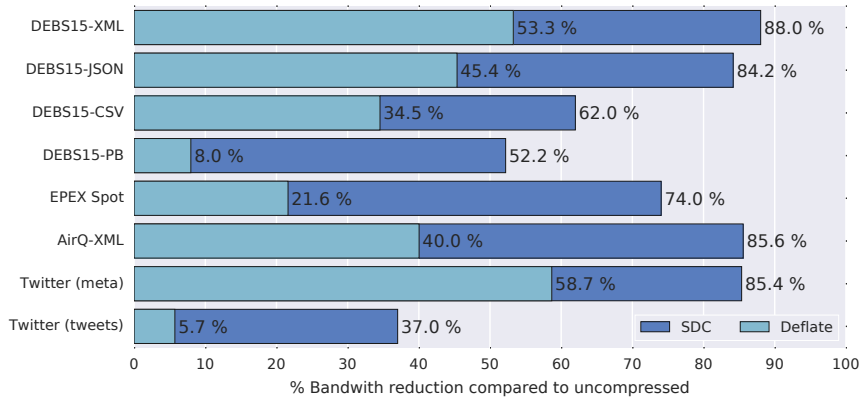


Figure 4.3.2: % bandwidth reduction incl. overhead

4.3.3 Computational costs of SDC

In the emulation experiment, we record wall clock time of the individual operations. Our experiment employs one fixed thread per core. Figure 4.3.3 shows that the time to compress a single message is linear to the size of the dictionary. Note the near-exponential scale on the x-axis. The uncompression process is performed in constant time. Figure 4.3.1 shows that at a certain size of the dictionary not a lot more bandwidth reductions manifest. Additionally, the computational power needed to compress messages grows. Hence it makes sense that the adaptive algorithm probes if a bigger dictionary pays off instead of

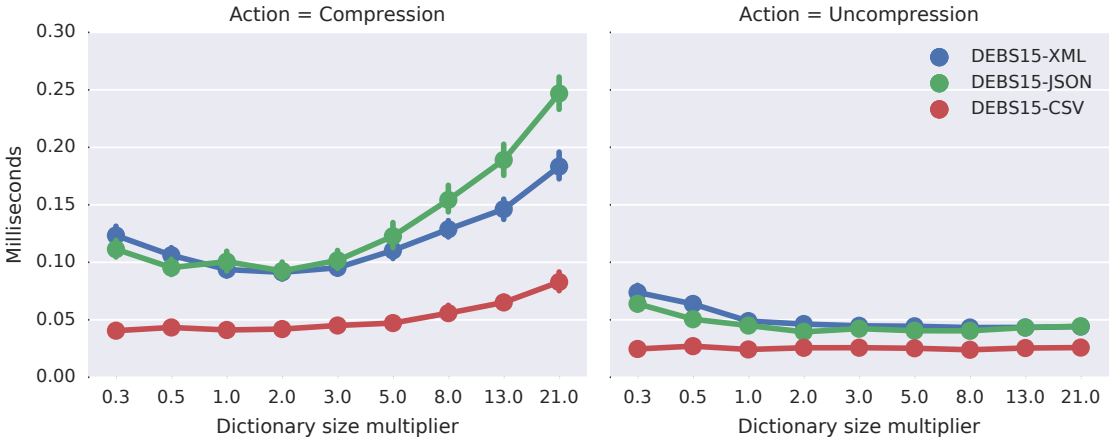


Figure 4.3.3: Compression/Uncompression time

defaulting to a bigger $SD_{multiplier}$.

4.3.4 Adaptive algorithm

We tested the adaptive algorithm in an emulation that publishes at a fixed rate of 100 notifications/sec. We simulated the DEBS data stream over the first 100k notifications.

Algorithm 4.2 shows the pseudocode of ADAPTIVE. It is initialized with the parameter $minImpr$ which changes how aggressive the Dictionary Maintenance Algorithm (DMA) is towards reaching higher bandwidth reductions. When the TTL is reached, which is expressed as point in time in the future, the algorithm evaluates the current savings and how high the savings would be with a new dictionary. In case the gain with the new dictionary is higher, a new dictionary is created. The TTL is set according the amortization time of the dictionary and how often the parameters oft the dictionary were already increased. The reason the existing adaptations are counted is that each adaptation increases the parameters and increased parameters means increased CPU cost for the sampling broker. That way, the sampler should find an equilibrium after some time.

Figure 4.3.4 shows the behavior of the algorithm over time. When a dictionary expires, a decision is made to either create a **New SD** or to **Prolong SD**, shown by the blue

Algorithm 4.1: Adpative - Estimate savings

Data: Set of messages $\mathbb{M}$

Procedure *estimateBR*($\mathbb{M}$)

- $\mathbb{M}_{train} \leftarrow \{\mathbb{M}_{buffersize \times 0.3} ... \mathbb{M}_{buffersize}\}$
- $\mathbb{M}_{test} \leftarrow \{\mathbb{M}_0 ... \mathbb{M}_{buffersize \times 0.3}\}$
- $SizeU_{test} \leftarrow \sum_{i=0}^{buffersize \times 0.3} |n_i|$ // Uncompressed size
- $dc \leftarrow buildDictionary(\mathbb{M}_{train})$
- $compress(\mathbb{M}_{test})$
- $SizeC_{test} \leftarrow \sum_{i=0}^{buffersize \times 0.3} |n_i|$ // Compressed size
- **return** $1 - \frac{SizeU_{test}}{SizeC_{test}}$ // Estimated *BR*

and green dots. The decision is based on the **Current Savings** and the estimated **New Savings**. The line **Total BW reduction** shows the bandwidth reduction over time taking into account the introduced overhead. The initial dictionary holds for more than 50k notifications and not enough new savings could be acquired to justify spreading a new dictionary. After 50k of notifications the content of the notifications changes which causes a drop of the bandwidth savings with the dictionary. The adaptive algorithm reacts by creating a new dictionary and changing the configuration. Alternating once the $SD_{multiplier}$ and once the B_{size} is increased.

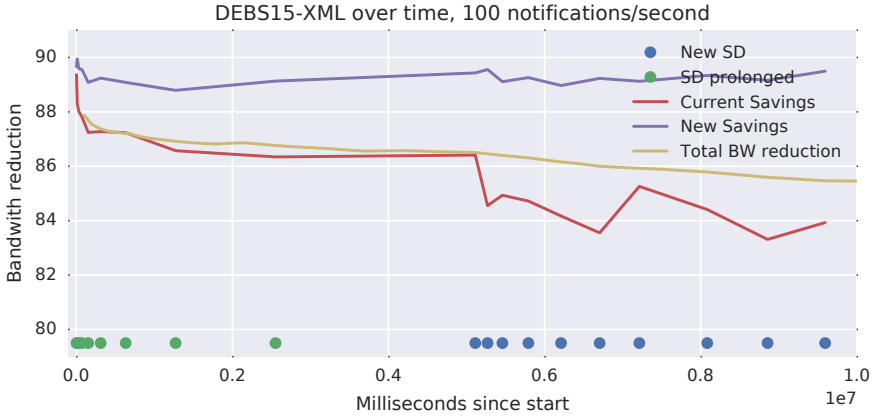


Figure 4.3.4: Adaptive algorithm over time

The adaptive algorithm could not reach the optimal results we have shown in the evaluation of the compression potential (see Table 4.3.2). While the JSON and XML variants are roughly ≈ 4% worse than the best parameter combination, the CSV variant is more than

Algorithm 4.2: Adaptive - Main loop

```
   Data: Minimum Improvement minImpr
 1 Procedure runAdaptive(minImpr)
 2     initialize 𝕄 , BS and , DM arrays
 3     msgCount, TTL, bufferIdx, multiplierIdx ← 0
 4     adaptationsCount, adaptations ← 0
 5     firstBreak ← 100
 6     step ← increaseBuffer while msgCount < firstBreak do
 7         𝕄 ← append(𝕄, m)
 8         msgCount ← msgCount + 1
 9     dc ← sampleDictionary(buffer)
10     BR ← estimateBR(𝕄)
11     T_amortize ← |dc| / (R×BR)
12     TTL ← T_armortize * 2000
13     publish(dc, TTL)
14     while m ← nextMsg() do
15         𝕄 ← append(𝕄, m)
16         if TTL is reached then
17             currentSavings ← calcSavings(dc)
18             newSavings ← estimateBR(𝕄)
19             BR_gain ← newSavings − currentSavings
20             if BR_gain > minImpr then
21                 adaptations ← adaptations + 1
22                 dc ← sampleDictionary(𝕄)
23                 T_amortize ← |dc| / (R×BR)
24                 TTL ← T_armortize * 10 * adaptations^3
25                 payoffNotSuccess ← 2 * payoffTimeMsec
26                 publish(dictionary, TTL)
27             else
28                 adaptations ← 0
29                 TTL ← payoffNotSuccess
30                 if adaptationsCount < 4 then
31                     alternateIncrease(...)                    // Algorithm 4.3
32
33                 adaptationsCount ← adaptationsCount + 1
34             payoffNotSuccess ← 2 * payoffTimeMsec
```

Algorithm 4.3: Adaptive - Alternating increase

```
   Data: Minimum Improvement minImpr
1  Procedure alternateIncrease()
2      if step = increaseBuffer then
3          bufferIdx++
4          𝕄 ← resize(𝕄, BS[bufferIdx])
5          step ← increaseDictSize
6      else
7          multiplierIdx++
8          step ← increaseBuffer
```

≈ 20% worse. The adaptive algorithm starts without knowing what the best parameter combination is and over time tries to achieve higher bandwidth savings. Every time a dictionary is sampled, a computational cost is incurred, hence it is not feasible to try out too many combinations for the sampling broker. Nevertheless, in all three cases the results are better than DEFLATE.

Dataset	Best in eval.	DEFLATE	Adaptive
DEBS15-XML	88.0 %	53.3 %	85.46 %
DEBS15-JSON	84.2 %	45.4 %	80.17 %
DEBS15-CSV	62.0 %	34.5 %	41.65 %

Table 4.3.2: Adaptive algorithm

4.3.5 Implementation on top of MQTT

The prototype is implemented on top of MQTT [50]. An existing MQTT-compliant broker [53] is extended to handle the sampling broker role. We measure the throughput in a real-world environment which includes a hosted server at a provider, an Android application as a subscriber and another server as publisher.

The Android phone is switched to 2G Mode. The signal quality is between −91 and $-82 dBm$, which explains the flakiness of the connection and of our results. Ping latency is between 290 – $650 ms$, which models our motivating rural scenario.

Figure 4.3.5 shows end to end latency at percentile in this experiment. Every 100 no-

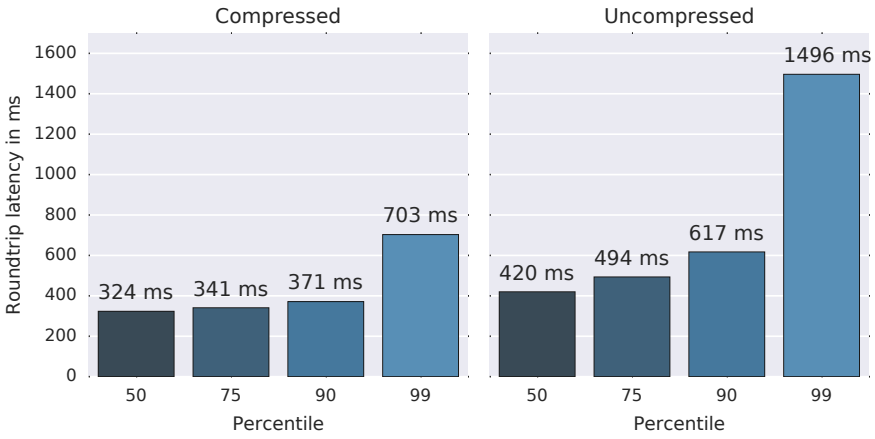


Figure 4.3.5: Latency

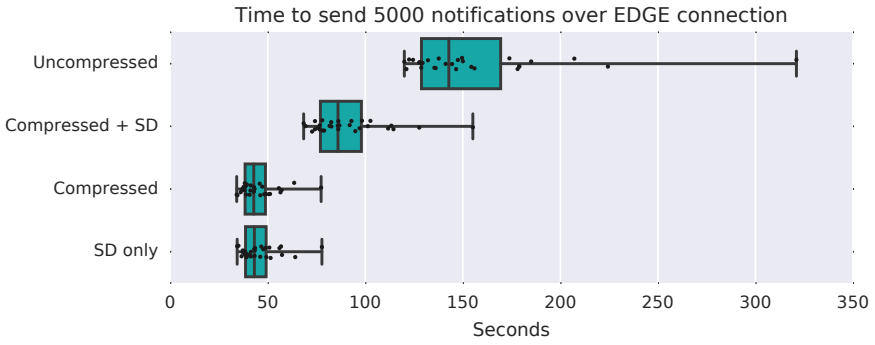


Figure 4.3.6: Throughput

tifications, the subscriber responds to a notification by publishing on a separate topic. The subscriber receives this notification and derives the latency. The results show a significant reduction in latency especially in the higher percentiles. Figure 4.3.6 shows the throughput rate in our experiment. First, 5000 uncompressed notifications are sent with the time span between the first and last message recorded (see bar **Uncompressed**). We then measured the time taken to send the dictionary (bar **SD only**). The dictionary multiplier is set to 1. We then send 5000 compressed notifications using the dictionary and also measured the time interval between the first and last message (bar **Compressed**). The bar **Compressed + SD** shows the time to transmit both the dictionary and 5000 compressed notifications. The chart shows that using SDC, ≈ 40% less time is needed to send 5000 notifications. Potentially, the dictionary can be used for another 5000 notifications which would amortize the dictionary overhead even more.

Chapter 5

PreDict: Predictive Dictionary Maintenance for Message Compression in Publish/Subscribe

Shared dictionary compression (Shared Dictionary Compression (SDC)) is able to go further in terms of bandwidth reduction by extracting the redundancy from a large set of messages into a dictionary [32]. However, the challenge with SDC is in choosing the parameters of the dictionary depending on the topology and the content of the messages, as well as amortizing the overhead introduced by sharing the dictionaries in the topology. Thus, SDC for pub/sub is an operational burden because, for each topic, a different set of parameters has to be configured or a Dictionary Maintenance Algorithm (DMA), which does not consider the topology, may choose unfavorable parameter combinations or introduce bandwidth overhead that cannot be amortized.

To address this challenge, we design a new dictionary maintenance algorithm called PreDict that adjusts its operation over time by adapting its parameters to the message stream and that amortizes the resulting compression-induced bandwidth overhead by enabling high compression ratios.

Parts of the content of this chapter are published as [20]. Additionally, this thesis presents

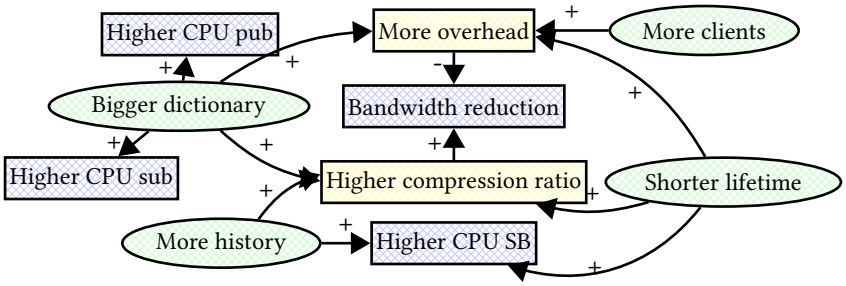


Figure 5.1.1: Relationships among algorithm variables (green) and cost (blue)

an evaluation on even more datasets and over larger windows of messages.

5.1 Overview

PREDICT observes the message stream, takes the costs specific to pub/sub into account and uses machine learning and parameter fitting to adapt the parameters of dictionary compression to match the characteristics of the streaming messages continuously over time. The primary goal is to reduce the overall bandwidth of data dissemination without any manual parameterization.

The challenge that we address in this paper is continuously adapting the parameters and the dictionary used for compression in a resource-efficient manner while still being able to react to new content biases in the stream to ensure high bandwidth reductions given the topology. Figure 5.1.1 summarizes the variable influences and conflicting parameter adjustments and effects on the costs in pub/sub. An adequate DMA would solve the operational concerns of selecting the parameters for dictionary compression in an automatic and online way and make this approach practicable for real-world usage. The content of messages can change over time, e.g., a stream from weather sensors changes depending on the season and Twitter messages change depending on current events such as elections and football games. Having a vast dictionary that covers all potential entries is not beneficial because the references to the entries become longer. Moreover, a large dictionary introduces overhead when the dictionary is shared in the pub/sub topology and causes higher computational costs for publishers in compressing

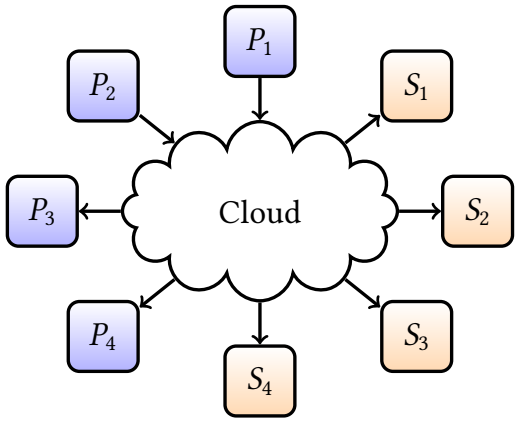


Figure 5.1.2: System model pub/sub overlay

the messages and for subscribers in decompressing the messages.

The system model we assume for our DMA is topic-based pub/sub with a single intermediary decoupling publishers and subscribers, henceforth called topology in this paper. See Figure 5.1.2 The single intermediary can be implemented in a scalable and fault-tolerant manner [54, 55, 56]. Many pub/sub systems available from cloud providers expose a single API endpoint while the backend is implemented as a fault-tolerant broker overlay.

PREDICT, our DMA, continuously observes the message stream at one of the brokers and decides to either recreate the dictionary based on more promising parameters to achieve higher bandwidth reductions or to retain the current dictionary. We use a combination of machine learning and model fitting to determine the parameters for the dictionary, therein also considering the overhead caused by sharing the dictionary in the pub/sub topology.

The remainder of this paper is organized as follows. In Section 5.2.3, we first introduce our metrics and then, we introduce dictionary-based compression and analyze the relationships among key variables to draw conclusions concerning our DMA. In Section 5.3, we present our PREDICT approach. In Section 5.4, we introduce the evaluation benchmarks and finally we present experimental results.

5.2 Dictionary parameter analysis

First, we introduce our target metrics; then, we introduce dictionary-based compression and the libraries that we use. Before we introduce PREDICT, our dictionary maintenance algorithm (DMA), and to understand important interactions, we analyze the relationships among the variables shown in Figure 5.1.1.

5.2.1 Evaluation metrics

The goal of our approach is to reduce bandwidth usage overall (*BR*). The *BR* metric is calculated from the bandwidth usage of the uncompressed messages ($B.u$) and the compressed messages ($B.c$) which include the overhead introduced by dictionary compression; see Equation 5.2.1. As outlined in our system model, which is similiar to that of cloud-based pub/sub we assume a single intermediary broker. We do not account for overhead of sharing the dictionary in the broker overlay because brokers do not need to compress or uncompress messages, hence, do not need a dictionary. Nevertheless an implementation would have to disseminate the dictionary to the clients either through the broker overlay or through other means, like a CDN (Content Delivery Network). Since brokers reside within the cloud and are connected by high bandwidth connections, overhead is less concern between the clients and the cloud service.

The bandwidth usage of the compressed messages ($B.c$) is calculated from the size of the compressed messages $|\mathbb{M}.c|$ that are sent once from the publishers ($\mathbb{P}$) to the broker. The broker forwards all compressed messages ($\mathbb{M}.c$) to each subscriber ($\mathbb{S}$). Each dictionary is shared with all publishers and subscribes; hence, the overhead equals the size of the dictionary $|DC|$ that is shared with each publisher and subscriber ($|\mathbb{P}| + |\mathbb{S}|$). The challenge for SDC in pub/sub is that each $\mathbb{P}$ sends only $\frac{|\mathbb{M}.c|}{|\mathbb{P}|}$ messages, while each subscriber ($\mathbb{S}$) receives all compressed messages ($\mathbb{M}.c$). Because there are fewer messages being published per $\mathbb{P}$, it is difficult to amortize the introduced overhead through a higher compression ratio of messages in topologies that have many more publishers ($\mathbb{P}$) than

subscribers (𝕊).

$$
\begin{aligned}
B.u &= |\mathbb{M}.u| \ (1 + |\mathbb{S}|) \\
B.c &= |\mathbb{M}.c| \ (1 + |\mathbb{S}|) + |DC| \ (|\mathbb{P}| + |\mathbb{S}|) \\
BR &= 100 - \frac{100 \ BR.c}{BR.u}
\end{aligned}
\tag{5.2.1}
$$

Topologies with many more ℙ than 𝕊 are frequent in the Internet of Things (IoT) where many sensors (ℙ) send messages to a few analytics engines or centralized data collection points (𝕊) [57]. In Smart City applications [58], many sensors send their information to a few interested parties or to a single database for offline analytics tasks.

5.2.2 DMA variable interactions

Dictionary maintenance affects bandwidth reduction via certain key variables. Figure 5.1.1 shows an overview of all variables (green) and the effect on costs (blue). Creating a larger dictionary is beneficial for higher compression ratios but increases the overhead when sharing dictionaries with clients. Creating the dictionary from more history increases the quality of the dictionary and only affects the Sampling Broker (SB) performance. When the dictionary is refreshed more often (low *MC*), new content biases can be incorporated faster, thereby improving the compression ratio. However, overhead from sharing th dictionary is incurred more often, which depends on the topology.

Dictionary sampling in the DMA has two parameters: the history size (*HS*) and the dictionary size. The dictionary size is expressed as a multiplier of the mean message size, called the dictionary multiplier (*DM*). After the DMA has published a dictionary, it waits until a certain number of messages (*MC*) is reached. Then, the DMA creates a new dictionary and measures the impact. A larger history results in a more accurate estimation of the impact of a new dictionary but also in higher computational cost. A smaller *MC* results in higher computational costs for the SB. However, it is possible to react faster to changes in the content of the message stream and to incorporate beneficial substrings faster to further reduce bandwidth.

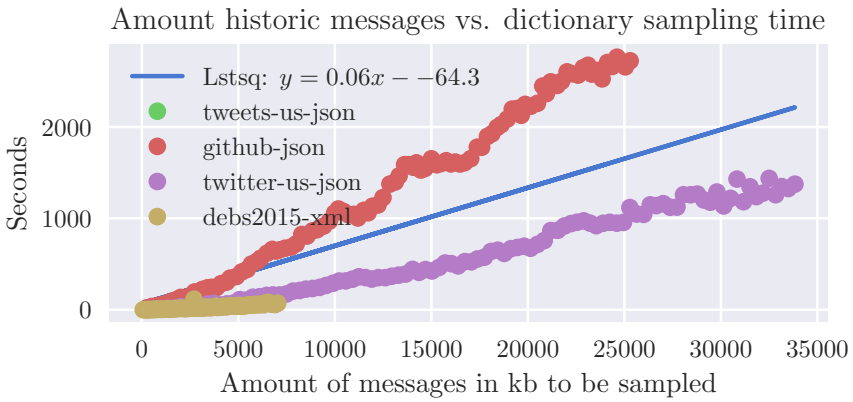


Figure 5.2.1: Dictionary sampling time

5.2.3 Dictionary analysis and costs

To design a DMA, we have to understand the relationship between costs and bandwidth reduction of the two dictionary parameters from which the algorithm can choose from: the history size and the dictionary multiplier. In the plots, we show as an example the Twitter dataset, which is extracted from the Twitter Firehose; more details regarding the datasets are given in Section 5.4.

History length

Figure 5.2.1 shows the relation between the number of historic messages in kB and the time to create a compression model in seconds. The blue line shows the least-squares regression line over all datasets, and the colored dots show the four chosen datasets and the actually measured values. The figure shows that the relation is linear but depends on the content of the messages. This cost is incurred by the SB when it estimates the impact of a new dictionary.

Figure 5.2.2 shows the effect of the number of histories on the bandwidth reduction. In this experiment, we take a number of messages from the history and compress the next 10k messages. Generally speaking, the bandwidth reduction increases when more messages from the history are sampled, but at a certain point, the gains plateau. In

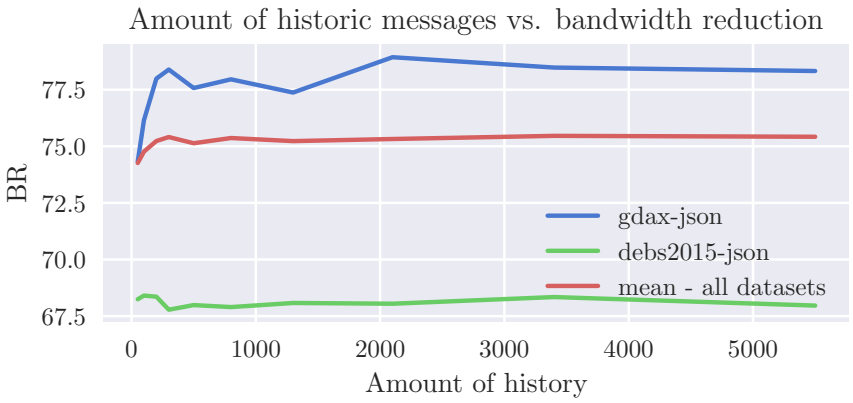


Figure 5.2.2: Number of histories and bandwidth reduction

certain datasets, e.g., Meetup-comments, some gains are still achieved; however, with more than 5500 messages, we did not observe any significant improvement in bandwidth reductions.

Dictionary size

Figure 5.2.3 shows the relationship between the dictionary size and the time needed to compress 1000 messages. For the twitter-us.json dataset, the relation increases linearly until a dictionary size of 70 kb, which corresponds to a size of 21× the average message size. After this point, the CPU time remains nearly constant. The CPU time plateaus because the additional entries in the dictionary do not tend to be useful, as shown in Figure 5.2.4. This graph presents least-squares regression lines with the same boundary at 70 kb. The pattern is similar in all datasets, but the thresholds are content dependent, e.g., in the tpch-lineitem dataset, the threshold is ≈ 8 kb, which corresponds to a multiplier of 65× the mean message size.

An ideal dictionary only contains entries that are actually used. We differentiate dictionary usage and internal usage. Internal usage refers to reused redundancy within a message. Table 5.2.1 shows the average entry usage from all datasets. In this experiment, we first create a dictionary from 5500 messages, and then we use it to compress the next 10k messages and record which parts of the message are reused (internal usage)

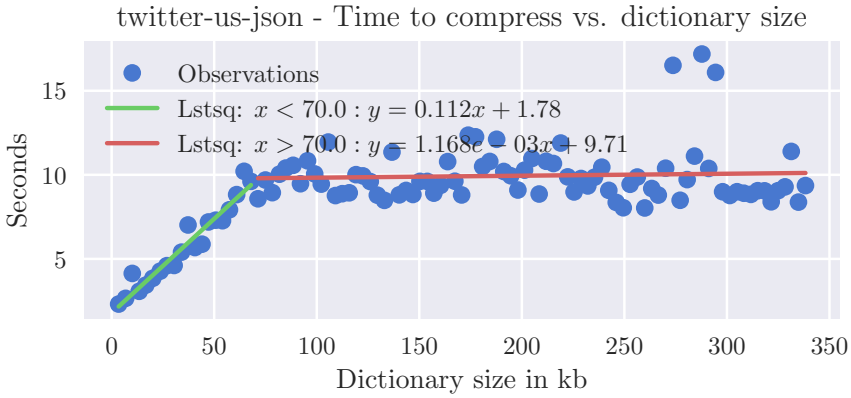


Figure 5.2.3: Time to compress and dictionary size

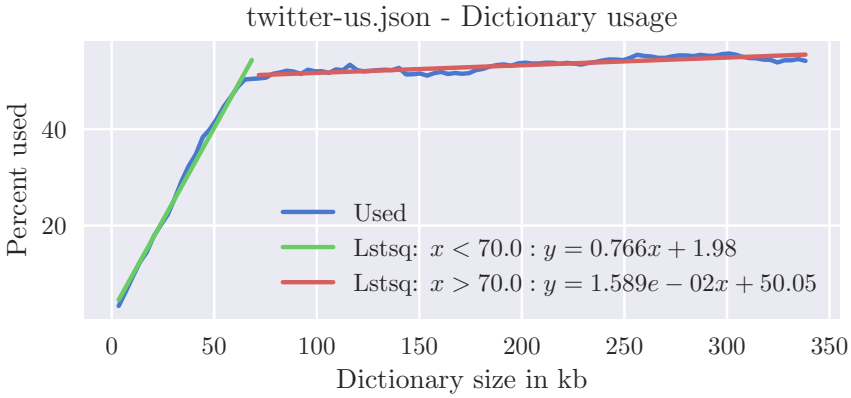


Figure 5.2.4: Dictionary usage vs. size

Multiplier	Dictionary usage	Internal usage
1.0x	97.8%	19.4%
2.0x	93.9%	15.9%
3.0x	94.4%	15.0%
5.0x	91.3%	12.6%
8.0x	91.6%	12.5%
13.0x	91.0%	12.4%
21.0x	87.4%	11.2%
34.0x	82.6%	10.5%
55.0x	74.6%	10.2%
91.0x	65.7%	10.3%

Table 5.2.1: Dictionary usage as a function of message size, mean of all datasets

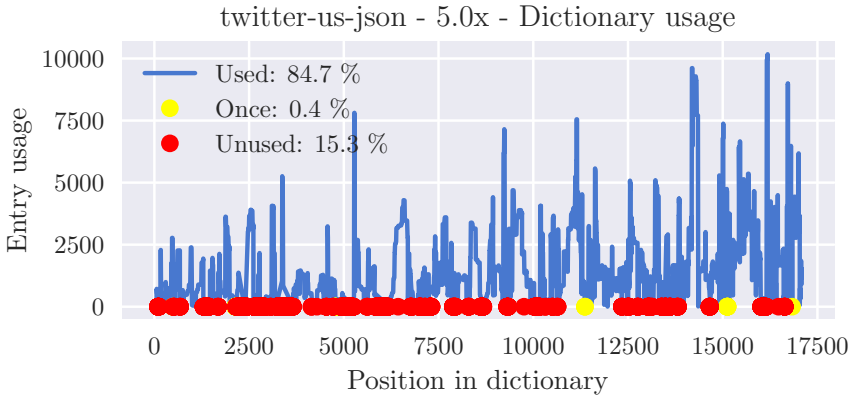


Figure 5.2.5: Dictionary usage

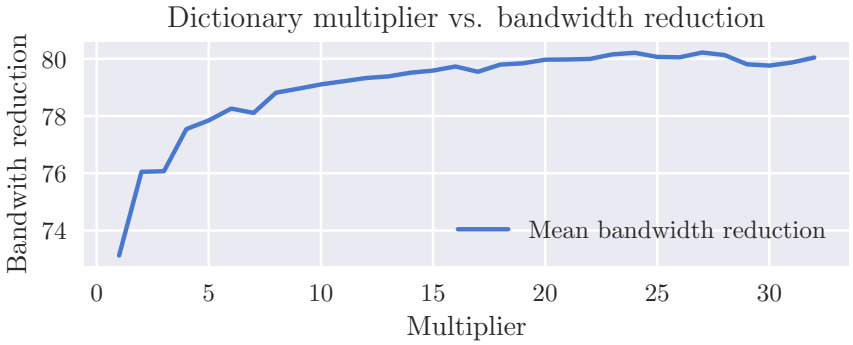


Figure 5.2.6: *BR* and size (multiple of mean message size)

and which parts of the dictionary are used. The usage of the dictionary is the highest when the dictionary size equals the message size. When the dictionary is 5× the message size, 10% of the dictionary remains unused. Although the overall usage of the dictionary decreases, more entries still reference the dictionary. The internal usage is approximately 20% when a dictionary is equal to the size of an average message. When the dictionary becomes larger, the internal usage decreases. This is because longer substrings can be referenced from the dictionary rather than from the redundancy within the message.

Figure 5.2.5 shows how often each chunk in the dictionary is used. Many chunks of the dictionary are used more than 5000 times, corresponding to nearly every other message. Small parts are even used more than 10k times, which means that these parts

are referenced multiple times within a single message. The substrings are sorted before they are merged, and the most common substrings are at the end of the dictionary.

Entries that are not used are a result of changes in the bias of the stream. Higher dictionary usage can be achieved if the content is known upfront, which is not possible in a stream. Figure 5.2.6 shows the mean bandwidth reduction and the dictionary size as a multiplier of the average message size. The general trend is that larger dictionaries induce higher compression ratios. However, this trend plateaus because the additional entries in the dictionary are not used very often.

Parameter selection conclusions

The main computational load is from extracting the Longest Common Substrings (LCS) and ranking them. After we have sampled the dictionary, we can evaluate different dictionary sizes with low additional computational costs. For this purpose, we take the dictionary, truncate it to a specific size, create a compression model that includes the Huffman tables and evaluate the bandwidth reduction. As shown in Figure 5.2.6, the bandwidth reduction can be modeled using a polynomial. In this way, we can determine a good parameter for the dictionary size by observing only several points and then fitting a polynomial over several observations.

We select ≈ 300 messages as a qualified number of historic messages to create a dictionary, as shown in Figure 5.2.2. The mean of all datasets plateaus at this threshold. Nevertheless, when we examine the compression ratios of the specific dictionary over longer parts of the stream, we observe that using more messages creates dictionaries that retain the compression ratios longer.

5.3 PreDict dictionary maintenance

Here, we introduce our approach for an online DMA that considers the conclusions drawn in the parameter analysis and enables automatic configuration of the content of

the message stream through parameter fitting and machine learning.

The DMA shares the dictionary with the clients and defines after how many messages (*MC*) the stream is observed again. Then, publishers can use the dictionary to compress the messages, and the subscribers decompress the messages. When the *MC* is reached, the DMA has to decide to either keep the dictionary or share a new dictionary with the clients that promises higher compression ratios. It is important to determine an accurate *MC*. An *MC* that is too high may miss additional *BR* gains with a new dictionary. An *MC* that is too low causes additional bandwidth overhead and computational effort when assessing the current *BR*. The *MC* is specific to content bias changes in the stream; hence, we use machine learning techniques to determine the variable. Machine learning requires a vector of features to predict a target, which is the *MC* in our case. The features that we use are derived from the substring scores of the dictionary. To extract the features, we introduce the substring repository (*SR*), which tracks the substrings and their scores over time and calculates the features to predict the *MC*.

5.3.1 Substring repository

The *SR* retains for each substring the following information: current score, occurrences in previous dictionaries (initialized with 1), mean scores of past occurrences (initialized with the current score) and sum of squared current scores to further derive the standard deviation (μ) of the scores. When the *MC* is reached, the LCS are extracted from a set of messages. Then, we update the *SR* using the new substrings and scores. The new substrings are not the same, but substrings from the *SR* can contain one or more new substrings. Hence, the *SR* has two operations: put and demerge. Put adds a new substring to the repository, which does not have any overlaps. Demerge is chosen when the substring overlaps with existing substrings. As an example, assume that a substring *abcdfg* is sampled and that the *SR* contains two entries: *abcdfgwxyz* and *abcdfgjklp*. After a demerge operation, the *SR* would contain *abcdfg*, *wxyz* and *jklp*.

5.3.2 Features for *MC* prediction

The substring score is a metric that represents the importance of the substring for the compression of messages. Equation 5.3.1 shows the calculation of the score as in the FemtoZip library [33], where S denotes the substring and O denotes the occurrences within the messages that were used to create the dictionary. The overhead of referencing a substring in the dictionary is three bytes; hence, substrings longer than three bytes should be included in the dictionary.

$$f(S,O) = \begin{cases} \frac{O\,(|S|-3)}{|S|}, & \text{if } |S| \geq 3 \\ 0, & \text{otherwise} \end{cases} \tag{5.3.1}$$

Periodically, we reassess the substring every time the *MC* is reached and extract the features used for machine learning. One feature is the standard deviation (μ), which shows how stable the substring score is. When the standard deviation is smaller, the entries do not change as much and remain equally important between the probes. The intuition behind the standard deviation is the following: when it is higher, the entries are more volatile, e.g., trending topics on Twitter. Hence, the *MC* should be shorter to recreate up-to-date dictionaries more often. The mean score indicates how important the substrings are for compression. Some message streams exhibit constant changes, and some datasets also include periods of large changes. Hence, predicting the *MC* based on two features that express the importance and volatility appears to be reasonable.

Using the scores and occurrences recorded by the *SR*, we calculate the features. We use the mean standard deviation ($\mu.mean$) to express the volatility of the substring score, as shown in Equation 5.3.2, where S_i denotes the score of the substring at the i^{th} time of its evaluation, and O denotes the occurrences. The second feature used for prediction is the mean of the scores of substrings ($s.mean$), which expresses the average importance, as shown in Equation 5.3.3, where N denotes the number of substrings in the dictionary.

$$\mu^2 = \frac{\sum_{i=1}^{N} S_i^2 - O\,(\frac{\sum_{i=1}^{N} S_i}{O})^2}{O} \tag{5.3.2}$$

$$S.mean = \frac{\sum_{i=1}^{N} S_i}{N} \tag{5.3.3}$$

Bootstrapping and recording training data – The main challenge in predicting the *MC* is creating a training dataset. Creating a training dataset is time consuming and computationally expensive because we need to create a new dictionary and measure how well it performs. Without sufficient training data, the prediction algorithm can easily predict an *MC* that is out of bounds. To reduce the reliance on training data, we first gather the *MC.base* derived from the amortization time of the overhead introduced by sharing the dictionary, as shown in Equation 5.3.4. The intuition behind *MC.base* is that the overhead introduced by sharing a dictionary should pay off; otherwise, no bandwidth reductions can be obtained and *MC.base* is a safe value, where the additional bandwidth overhead from sharing a dictionary is 0.1%. *MC.base* is calculated from the estimated bandwidth reduction (*BR*), the mean message size (∅), and the overhead (*oh*). Furthermore, we choose a factor of 1000, which means that the sharing of the dictionary should be amortized at least 1000× or that the introduced overhead from sharing a dictionary is 0.1%. This value has been found to be a safe compromise between overhead and bandwidth reduction.

$$\begin{aligned} oh &= |DC|\ (|P| + |S|) \\ MC.base &= \frac{oh\ (100 - BR)}{\emptyset\ 100}\ 1000 \end{aligned} \tag{5.3.4}$$

To bootstrap the training dataset, we use *MC.base* until a sufficient number of observations has been recorded (see Equation 5.3.5). Once we collect a sufficient number of observations, the algorithm constructs the machine learning model and further *MC* predictions are derived from the model.

$$f(\mu.mean, s.mean) = \begin{cases} < 6, & MC.base, \text{ see Equation 5.3.4} \\ \geq 6 \text{ obs.}, & p(\mu.mean, s.mean) \end{cases} \tag{5.3.5}$$

The training data is created in the following manner: when *MC* is reached, the DMA

probes the dictionary. The dictionary has an age counter of how often *MC* is allowed to be prolonged (Case 5.3.6). This is because there may be new biases in the message stream that could be represented in the dictionary to increase *BR*. Because the dictionary has already been amortized more than initially expected, we create a new dictionary to determine whether the message stream exhibits new patterns and potentially achieves further bandwidth reductions.

$$f(dc_{age}) = \begin{cases} dc_{age} \geq 6, & \text{Dictionary too old} \\ otherwise, & \text{Equation 5.3.8} \end{cases} \tag{5.3.6}$$

The algorithm creates a new dictionary and calculates $BR.dict$ by compressing several recent messages. The new dictionary should achieve an at least a 1% improvement ($I.min$) in terms of *BR* compared to $BR.curr$, which is the currently deployed dictionary; see Equation 5.3.7. To make the algorithm more robust against small fluctuations in compression performance, we also track $BR.avg$, which is the average *BR* achieved. $BR.imp$ then shows how good the bandwidth reductions of the new dictionary are compared to the currently active dictionary.

$$\begin{aligned} I.min &= \frac{BR.curr}{100} \\ BR.imp &= max(BR.avg, BR.curr) - BR.curr \end{aligned} \tag{5.3.7}$$

$$f(BR.imp) = \begin{cases} BR.imp \geq I.min, & \text{a) increase} \\ BR.imp \leq I.min, & \text{b) decrease} \\ otherwise, & \text{c) } \approx \text{ same} \end{cases} \tag{5.3.8}$$

In Case 5.3.8.a, the current dictionary performs better than initially measured. Hence, we extend *MC* by the same number of messages that it has served for and also increase the expectations of *BR* to the current evaluation ($BR.dict$). We update the outcome regarding *MC* in the training dataset for future predictions. The dictionary age is not increased. When the bandwidth reductions decrease by more than $BR.imp$ (5.3.8.b), a new dictionary

is created. We append it to the training dataset using only half of the validity time of the dictionary ($\frac{MC}{2}$). The assumption taken is that given the value of the features extracted from the *SR*, the number of messages for which the dictionary was valid (*MC*) should be lower. In Case 5.3.8.c, the dictionary performed well, and the current messages can be compressed as well as before. We record this observation in the training dataset. Then, we extend the dictionary again and also increase the age counter of the dictionary.

Regression model – Over time, we construct a training dataset and move from the heuristic to the prediction model. The main problems that we face are a limited amount of training data and outliers. Because training data is not readily available and computationally expensive to obtain, we cannot use a machine learning method that relies on a large dataset. Using a sophisticated prediction model on a small dataset is difficult because overfitting becomes more difficult to avoid. By starting with the heuristic and values that ensure a low overhead, we prime the training dataset with feasible values. Hence, the predictions are also in that range and further evolve using future data. For the prediction model, we use ordinary least squares (OLS).

Dictionary variables – The dictionary is created using two variables: the dictionary size (expressed as the dictionary multiplier) and the number of historic messages. Figure 5.2.6 shows that it is possible to model the relation between the bandwidth reduction and the dictionary size using a second-order polynomial. To fit the polynomial, we create dictionaries at multiple points using the dictionary multipliers shown in Equation 5.3.9.

$$\mathbb{DM} = \{0.5, 1, 2, 3, 5, 8, 13, 21\} \tag{5.3.9}$$

First, we split the messages into training and test datasets. The reason for this split is that when a dictionary is constructed from the same set of messages on which the bandwidth reduction is calculated, the bandwidth reduction is overestimated. A similar effect is known in machine learning as overfitting. Then, we construct a dictionary using the training dataset and shorten it using a multiple of the mean message size, as shown in Alg. 5.1 Line 7. Next, we construct the Huffman model and test the bandwidth reduction using the test dataset. Using the recorded observations, we fit a function using a Gauss-Newton Optimizer, as shown in Alg. 5.1 Line 11. Then, we find a point where

the steepness of the curve starts to flatten. This point is defined as the point where a less than 3% increase is achieved; see Alg. 5.1.17. We take this point as a dictionary multiplier (𝔻𝕄).

The second variable, the number of historic messages (ℍ𝕊), is increased in ascending steps until 500 is reached; see Equation 5.3.10. This is because after 500 messages, the bandwidth reduction does not increase significantly; see Figure 5.2.2.

$$\mathbb{HS} = \{50, 100, 200, 300, 500\} \tag{5.3.10}$$

PreDict pseudocode – We specify the implementation of PreDict in pseudocode; see Alg. 5.2. The algorithm waits until 500 messages have been collected; see Line 6. Then, the algorithm creates a dictionary, predicts the *MC* and shares the dictionary with the clients. After the initial phase is completed, the algorithm switches to continuous maintenance mode of the dictionary. When both the *MC* and the maximum dictionary age are reached, the algorithm creates a new dictionary and shares the dictionary with the clients; see Line 12. If the *MC* is reached, the algorithm probes the bandwidth reduction in the current serving dictionary to check whether the $BR.curr$ has decreased; see Line 18. The `updateModel` function adjusts the recorded *MC* in the training dataset and rebuilds the prediction model.

5.4 Evaluation

We first describe how we compare with traditional compression and delta encoding. Then, we introduce the simplest way to implement dictionary compression (SingleDict) and an approach that creates new dictionaries at fixed time intervals. Subsequently, we introduce Adaptive the current state-of-the-art DMA. Finally, we introduce the datasets and their statistics.

Algorithm 5.1: Find dictionary multiplier

Data: $\mathbb{M}$ representing historic messages
Result: Dictionary size in bytes

```
1  Procedure probeMultiplier(𝕄)
2      𝕄.train ← {𝕄_{buffersize×0.3} ... 𝕄_{buffersize}}
3      𝕄.test ← {𝕄_0 ... 𝕄_{buffersize×0.3}}
4      DC ← buildDictionary(𝕄.train)
5      𝔹ℝ ← init()                          // List of Tuples(multiplier, br)
6      for dm ∈ 𝔻𝕄 do
7          dc ← resize(DC, dm × meanM)
8          hm ← buildHuffmann(dc, 𝕄.train)
9          br ← evaluate(hm, 𝕄.test)        // Evaluate with test
10         𝔹ℝ ← append(𝔹ℝ, (br, dm))
11     f ← GaussNewtonOptimizer(𝔹ℝ)
12     I.old, I.new, dm, i ← 0
13     repeat
14         dmbest++, i++
15         I.old ← I.new
16         I.new ← f(i)
17     until (I.n − I.o) < 3% or i > max(𝔻𝕄)
18     return dmbest × meanMessageSize(𝕄)
```

5.4.1 Other approaches

For each dataset, we run all DMAs with combinations of configuration options to determine the best performing combination for each dataset and sampler. We evaluate five different algorithms, see Table 5.4.1, and compare them in terms of the CPU time for publishers, subscribers and *SB*, and the overall bandwidth reduction, which includes the introduced overhead.

DEFLATE

To compare with traditional compression, we choose DEFLATE [18]. The compression is end-to-end, which means that a publisher compresses the message and the subscriber decompresses the message; hence, no SB is needed, and no additional protocol overhead has to be factored in.

Algorithm 5.2: Main procedure

```
   Data: stream of messages on a single topic
 1 Procedure maintainDictionaries(... )
 2     initialize 𝕄 and , dictionary mlp arrays
 3     ℍ𝕊 ← {50, 100, 200, 300, 500}
 4     MC, HSIndex ← 0
 5     predicted ← false
 6     while |𝕄| < 500 do
 7         𝕄 ← append(𝕄, nextMsg())
 8     MC ← predictMC(...)                          // Start, see Equation 5.3.5
 9     BR.dict ← shareDictionary(𝕄, MC)
                                                    // Alg. 5.3
10     while m ← nextMsg() do
11         𝕄 ← append(𝕄, m)
12         if MC reached and dictage > maxdictage then
13             MC ← predictMC(... )
14             predicted ← true
15             BR.dict ← shareDictionary(𝕄, MC)
16         else if MC reached then
17             I.min ← BR.dict / 100
18             BR.curr ← calculateSavings()
19             BR.imp ← ...                         // See Equation 5.3.7
20             updateAvgSavings(BR.curr)
21             if BR.imp ≥ I.min then               // See Equation 5.3.8.a
22                 updatePred(2 × MC, predicted)
23                 MC ← 2 × MC
24                 BR.dict ← BR.curr                // Raise expectations
25                 continue                         // Keep current dictionary
26             else if BR.imp ≤ I.min then          // See Equation 5.3.8.b
27                 updatePred( MC/2, predicted)
28                 increaseHSIndex(... )            // More history
29             else                                 // See Equation 5.3.8.c
30                 predicted ? updatePred( MC, predicted)
31                 predicted ← false
32                 dictage++
33                 if dictage < maxDictAge then
34                     MC ← 2 × MC
35                     continue                     // Keep current dictionary
36             MC ← predictMC(... )
37             shareDictionary(𝕄, MC)
```

Algorithm 5.3: Helper functions

```
   Data: Dictionary
1  Procedure shareDictionary(𝕄, MC)
2      bSize ← buffArray[bufferIdx]
3      dSize ← dictArray[multiplierIdx]
4      dc ← sampleDictionary(𝕄, bSize, dSize)
5      BR ← calculateSavings()
6      predictionMade ← true
7      sendDict(dc, MC)                          // Send dictionary to clients
8      return BR
9  Procedure updatePred(MC.new, predicticted)
10     if predicticted then
11         updateFeatureMatrix(MC.new)
```

Algorithm	Description
DEFLATE	Off-the shelf compression
VCDiff	Delta encoding
Fixed rate	Best possible combination
Singledict	Simplest benchmark
Adaptive	Baseline and state-of-the-art
PREDICT	Our approach

Table 5.4.1: Algorithm overview

VCDiff

To use VCDIFF in pub/sub, we encode s messages as the differential to the previous message. We use a Java library [59] that implements RFC3284 [60]. We also evaluate sending the differential of two consecutive messages using the O(ND) Difference Algorithm [61]. In our experiments, the *BR* achieved using VCDIFF was found to be higher; hence, we use VCDIFF as the baseline achievable with delta encoding.

SingleDict dictionary maintenance

SINGLEDICT is the simplest method for implementing dictionary compression in pub/sub. It creates only one dictionary from sampling the first 10k messages. We choose a large

range of dictionary multipliers, see Equation 5.4.1, and evaluate all permutations. This benchmark shows the bandwidth reduction when only a single dictionary is used without any maintenance.

$$\mathbb{DM} = \{0.3, 0.5, 1, 2, 3, 5, 8, 13, 21, 35, 56, 91\} \tag{5.4.1}$$

FixedRate dictionary maintenance

FixedRate is a benchmark for PreDict. It does not consider any feedabck; hence, it is able to find the baseline of when no heuristic or prediction is used to determine the *MC*. Furthermore, we evaluate $\mathbb{DM} \times \mathbb{BS} \times \mathbb{WS}$ parameter permutations; see Equation 5.4.2. We then show the best performing permutation in terms of *BR*. $\mathbb{DM}$ is the set of dictionary multipliers used, $\mathbb{HS}$ lists the buffer sizes for historic messages and $\mathbb{MC}$ indicates how long the dictionary was valid. The initial $\mathbb{DM}$ and $\mathbb{BS}$ used to sample the dictionary are set to the first values of the sets. Every 10 dictionaries, the algorithm increases the history size and the dictionary size by one in an alternating manner. The goal of this algorithm is to show what *BR* can be achieved when the DMA does not react to content bias changes or is unaware of the topology and only optimizes for the highest compression ratio of messages.

$$\begin{aligned} \mathbb{DM} &= \{0.3, 0.5, 1, 2, 3, 5, 8, 13, 21\} \\ \mathbb{BS} &= \{50, 100, 200, 300, 500, 800, 1300, 2100, 3400, 5500\} \\ \mathbb{MC} &= \{1500, 7500, 15000, 30000\} \end{aligned} \tag{5.4.2}$$

Adaptive sampler

Adaptive is an implementation of the DMA presented in [19] and represents the current state of the art. After a warm-up phase of 100 messages, where the messages are buffered, the algorithm begins to sample a dictionary. Then, the current message rate (R) is derived, and the impact on *BR* is estimated. The adaptive algorithm considers

the amortization time calculated based on the bandwidth usage of the stream and the estimated bandwidth reductions of a new dictionary. The calculation does not consider the ratio of publishers to subscribers. AS defines that the overhead of the new dictionary should pay off at least 2000 times and sets the TTL accordingly. This approach assumes a TTL in seconds, which means that it would also need a prediction of the message rate. The evaluation of ADAPTIVE did not consider changing message rates over time and assumed that the message rate remains constant [19]. If the message rate decreases, the computational overhead would increase since the sampling broker would have to make the decision whether to renew the dictionary or prolong the dictionary. In the contrast, when the message rate increases, an outdated dictionary would be active longer and may not achieve high bandwidth reductions. In the evaluation of this approach, we consider a constant message rate; hence, these cases do not occur. The *BR* estimation splits the dataset into training (70%) and test datasets (30%). The dictionary is constructed from the training dataset, and the test dataset is compressed using the dictionary, from which the *BR* is derived. Because the estimation of *BR* creates a new dictionary, it also causes a high computational load for the SB. When the TTL is reached, the algorithm first calculates the *BR* with the current savings and compares it to the gain in savings for the case where a new dictionary would be used. If the estimated gain in *BR* is larger than a predefined threshold (*MI*), a new dictionary is created from the entire buffer. The calculation of the TTL includes an exponential that expresses how many changes to the dictionary configuration have been made. Because the parameters regarding the dictionary size only increase over time, every change to the parameters means a higher cost for publishers, subscribers and the SB. If no gain above the minimum improvement (*MI*) is reached, the dictionary parameters are increased. Increasing the buffer size or the dictionary size can improve the compression, and thus, when the next TTL is reached, it is quite likely that a new dictionary reaches the *MI*. The main assumption behind this approach is that the parameters will reach a certain equilibrium and stop increasing because no *MI* can be reached. We evaluate ADAPTIVE with $\mathbb{MI} = \{1, 2, 3, 4, 5\}$ and present the best performance in terms of *BR*.

Dataset	Short	Format	Messages	Size	Mean NSim	Std NSim
DEBS 2015	DEBS csv	csv	1999999	367M	83.90	4.50
DEBS 2015	DEBS json	json	1999999	1G	97.19	0.67
DEBS 2015	DEBS xml	xml	1999999	1.4G	98.03	0.47
GDAX Exchange	Gdax json	json	1729210	429M	81.69	18.96
Github	GH json	json	126245	302M	92.05	4.34
Meetup comments	Comm json	json	1213782	1.2G	95.80	1.66
Meetup events	Eve json	json	1014903	2.4G	88.91	4.10
Meetup RSVPs	RSVPS json	json	5641665	7.4G	97.50	0.77
Neutron	Neut log	log	800228	229M	81.77	14.97
Twitter US	Twitter json	json	635037	2G	96.04	3.70
TPCH lineitems	tpch tbl	tbl	6001215	725M	98.12	0.72
Bart arrivals	Bart json	json	7935189	11G	86.30	14.32
Chicago bike	Chi-b json	json	6102802	2.7G	98.45	0.32
METAR AWC US	MET json	json	9411665	4.6G	97.62	0.86
NYC Bike Live	NYC-B json	json	10192643	4G	98.08	0.54
GTFS EDmonton	GTFS-E	json	19791498	6.4G	96.80	0.58
Satellites	sat json	json	29609645	5.9G	95.08	0.73
Twitch Events	twitch json	json	1753147	2.0G	58.76	36.35

Table 5.4.2: Dataset statistics

5.4.2 Datasets

We collected several datasets from public API endpoints, as shown in Table 5.4.2. The DEBS2015 dataset [48] contains all taxi trips within New York over a year. We converted the dataset into JSON, XML and ProtoBuf [51] to also observe the *BR* with different message formats. The GDAX dataset was acquired from a public API of the GDAX Exchange. The Github dataset was also extracted from the publicly available API and contains notifications of various hosted repositories. The Meetup datasets were acquired from three different public API endpoints. The Meetup comments feed contains text messages that users post to comment on an event. The Meetup events stream contains all newly created events by their members, and Meetup RSVPs are a stream of people who attend the events. The Neutron log is a large logfile from an Openstack cluster that contains info and error messages of the networking service. The Twitter dataset was extracted from the public Twitter Firehose, therein being restricted to coordinates located within the U.S., and it was scraped in November 2016 during the US presidential elections. Additionally, we generated data from the TPC-H benchmark because dictionary-based compression can be employed when replicating databases over geographically separated

data centers. The datasets are realistic for evaluating the compression because they exhibit real-world content bias changes. Furthermore, all datasets are publicly available. Additionally, many of these Web services use pub/sub in there backend infrastructure [12] or as an abstraction for communicating with smartphone applications [13].

An important metric for the datasets is the similarity between messages. To quantify the datasets and make our work comparable to works that cannot publish their datasets, we computed the ratio of common substrings larger than 3; see Equation 5.4.3. We introduce two metrics that describe the datasets: the mean similarity and the standard deviation. A higher mean similiarity indicates a better performing dictionary-based compression. A higher standard deviation indicates a greater need for a DMA that reacts to bias changes. Table 5.4.2 shows the metrics in the last two columns.

Furthermore, the boxplots in Figure 5.4.1, show the variance of the similarity. The DEBS 2015 datasets in the various formats exhibit similarity as expected. The XML format introduces more overhead and hence a higher redundancy. The Neutron log has a broad distribution of similarity. When further examining the content of the dataset, we observe that, at night fewer users use the cluster, reconfigure the virtual networks and bring up new virtual machines; hence, similar status messages are dominant. Conversely, during the day, users of the cluster bring up or shut down virtual machines more frequently, producing less similar log messages.

$$
\begin{aligned}
n &= \{1, 2, \ldots 100\} \\
f(mt) &= \begin{cases} |mt|, & \text{if } |mt| > 3 \\ 0, & \text{otherwise} \end{cases} \\
s(m, n) &= \frac{2 \sum_{i=0}^{i=submatches} f(|sim_i|)}{|m| + |n|} \\
NSim &= \sum_{test=0}^{test=100} \sum_{eval=begin}^{eval=begin+100} s(m_{test}, m_{eval})
\end{aligned}
\tag{5.4.3}
$$

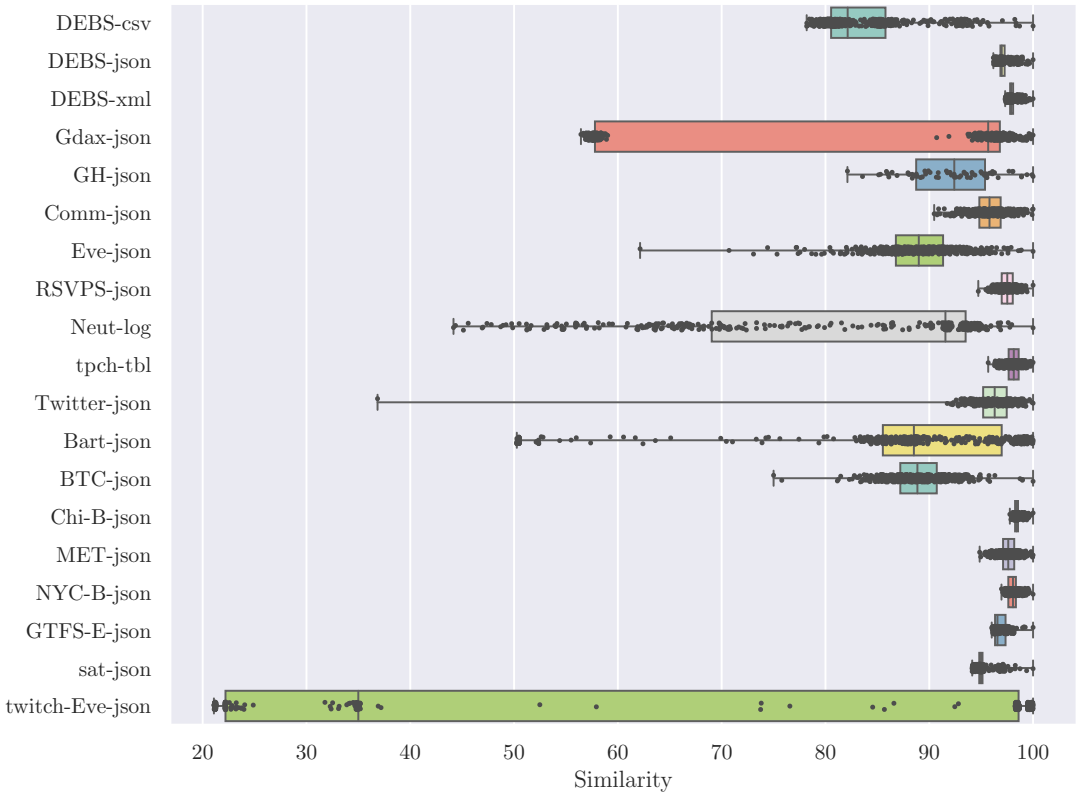


Figure 5.4.1: Normalized Similarity

5.4.3 Metrics

The *BR* is calculated from all messages in the topology; see Section 5.2.1, Equation 5.2.1. It includes all messages transmitted and the overhead introduced when dictionary-based compression is used according to the system model. For VCDIFF and DEFLATE, there is no additional overhead because these methods do not need to share information for compression. We do not consider any additional overhead for VCDIFF in pub/sub. In a real-world implementation, additional protocol overhead must be considered. In addition to the dictionary, all SDC methods assume a metadata field in the message of a single byte that identifies the dictionary that the subscribers use to choose the correct dictionary to decompress the message.

The computational costs are measured in terms of CPU time. For the computational cost of the publisher, we take the time needed to compress each message in nanoseconds and sum these times for all messages. We do the same for the subscribers, where we take the time to decompress each message in nanoseconds. Similarly, the computational overhead of the *SB* is the time needed for the evaluation of the dictionary performance, dictionary creation and updates of the prediction model. The server used for the evaluation contains 4x Intel Xeon E7v3 @ 2.20GHz CPUs.

5.4.4 Prediction error MC

Table 5.4.3 presents the prediction errors of PREDICT. We find that message formats such as *CSV* and *ProtoBuf* exhibit more changes in content biases because there is minimal schema overhead in the dictionary; hence, the predicted *MC* is shorter, and consequently, more dictionaries are used during the evaluation. The results of the Meetup comments and Github datasets are surprising because they are large datasets but no more than 5 dictionaries were used; hence, only the bootstrapping heuristic is used in PREDICT. This is because of the content of the messages. For example, in the Meetup comment dataset, a large part of a link is repeated 3×, and some other larger chunks of a message are nearly always the same. Only certain parts, e.g., the event id and the comments, are highly random and are not captured in the dictionary. Because *BR* remains approximately the same in these datasets, there is no need to continuously update the

dictionary; see also Table 5.4.4, where a permutation of SINGLEDICT performs the best in the Meetup comments dataset.

When examining the content of the dictionaries in the comments dataset, the schema is present but not many parts of the fields are present. Since comments are in free text, there does not seem to be enough bias to promote these parts in the dictionary. When looking at the results in Table 5.4.4, the *BR* is still high but the Singledict sampler performed best for the comments dataset.

Dataset	Dictionaries	MPE
Debs2015-pb	229	75.4%
Github-json	5	31.3%
Meetup-rsvps-json	71	6.1%
Meetup-events-json	12	33.7%
Meetup-comments-json	5	30.8%
Debs2015-xml	35	3.1%
Debs2015-csv	162	54.2%
Gdax-json	30	50.2%
Debs2015-json	50	24.3%
Twitter-us-json	15	0.0%
Neutron-log	22	50.9%

Table 5.4.3: Prediction error *MC*

Figure 5.4.2 shows the predicted *MC* and the actual *MC* over time for the debs2015-json dataset. In many cases, the predicted values were too small, and the dictionary could have been active for a longer time. In 75% of the cases, it was predicted correctly.

5.4.5 Bandwidth reduction

Table 5.4.4 presents the best result for each algorithm with respect to the achieved bandwidth reduction. The presented computational costs correspond to the best result in terms of bandwidth reduction. The unit of the computational cost is the total CPU time in minutes. The results for *BR* show the method with the highest *BR* of all permutations in the case with one subscriber and one publisher when applied to topologies that have many more publishers than subscribers.

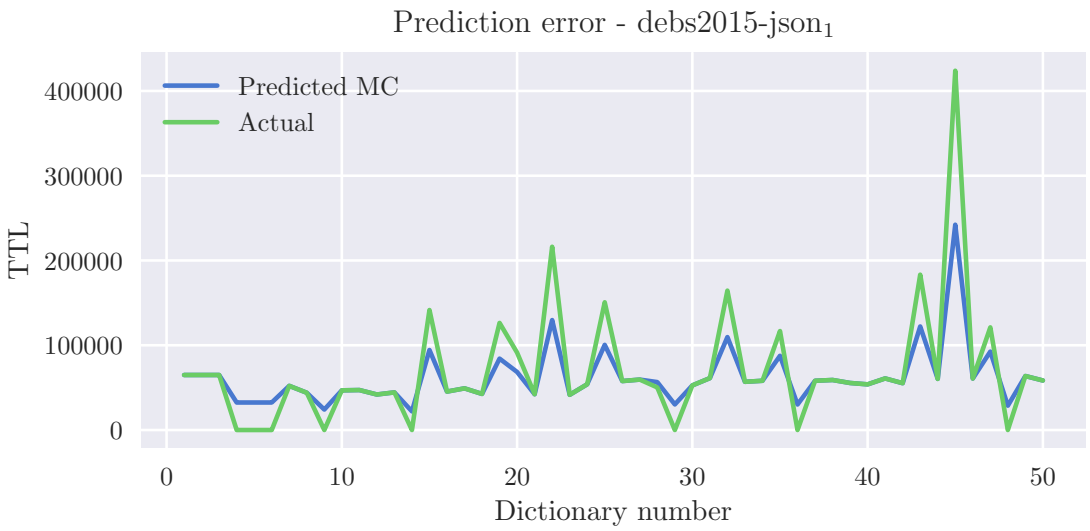


Figure 5.4.2: Prediction error of PREDICT

The lowest computational overhead for publishers is achieved with DEFLATE and VCDIFF. PREDICT uses ≈ 30% more CPU time on the publisher side. Note that DEFLATE uses the implementation available in Java SE 8 (java.util.zip.Deflater). The library that PREDICT uses is implemented in Java only. ADAPTIVE uses a heuristic that increases the size of the dictionary when no further *BR* is achieved and ultimately ends up with a large dictionary, which negatively affects the CPU overhead. PREDICT fits the parameters and takes as the dictionary size the value that is 3% below the maximum, which is a good compromise between overhead and *BR*.

The computational costs for the subscribers are similar under all approaches. Surprisingly, the cost for PREDICT is even lower than that of DEFLATE which uses the ZLib compression library. In addition, the Huffman table is shared upfront and often reused; hence, the runtime optimizations of the Java runtime can be used.

The CPU cost for SB is 0 with DEFLATE and VCDIFF because no DMA is needed. PREDICT uses the fewest resources because, compared to ADAPTIVE, no additional dictionary is built to estimate *BR*. FIXEDRATE shows the best performance when no feedback is incorporated and hence always creates a new dictionary.

Dictionary compression also works well with binary data; see the DEBS2015-pb dataset,

a ProtoBuf encoded variant. Using a DMA the *BR* is approximately 50%, while DEFLATE achieves a 7% *BR*. This shows that single messages lack sufficient redundancy that can contribute to the *BR* but there is sufficient redundancy that is shared among several messages. VCDIFF which only encodes the delta between two consecutive messages, has no significant *BR*.

The mean *BR* using DEFLATE was 41.3%. DEFLATE and VCDIFF have no additional overhead that depends on the topology; hence, the *BR* is the same for all topology configurations. ADAPTIVE, FIXEDRATE and SINGLEDICT are not aware of the publisher-to-subscriber ratio. In the case where there are many publishers, the overhead is excessive, and dictionary-based compression configured without knowing the topology leads to overhead that cannot be amortized; even off-the-shelf compression using DEFLATE is a better option. PREDICT considers the topology and still achieves ≈ 72% bandwidth reductions in toplogies with 1k publishers and a single subscriber. If the topology contains 10k publishers and and a single subscriber, the overall bandwidth reduction drop to 44% which is still 3%*pp* more than DEFLATE. The other approaches didn't reach bandwidth reductions over DEFLATE or even add more overhead than ever could be amortized.

Figure 5.4.3 shows the worst, best and median case results in boxplots for all permutations using ADAPTIVE and SINGLEDICT. We included PREDICT to emphasize that it does not need any configuration. ADAPTIVE and SINGLEDICT use parameters that are manually configured, while PREDICT adapts itself to the messages of the stream. For this figure, we took the best result in terms of *BR* as the baseline of each method, and the boxplot shows how much better or worse the results can be with other configurations that do not achieve that high *BR*. Note that the deviation of *BR* is expressed in percentage points, while the CPU costs are expressed relative to the result with best *BR* per DMA.

When SINGLEDICT is configured with unfit parameters, the *BR* can decrease by as much as 25%*pp*, and the CPU costs can be up to 85% worse for the publisher and up to 38% worse for the subscriber. When the optimal parameters are not taken for ADAPTIVE the *BR* can be decrease up to 6%*pp* while the CPU cost for the publishers can range between 25% better and 60% worse, with the median being 21% worse. The main reason for this behavior is that ADAPTIVE ends up with dictionaries that are too large, which do not have a high impact on the *BR*. Furthermore, if the DMA ends up with dictionaries that are

too large, then the overhead is amplified in topologies that have many more publishers than subscribers.

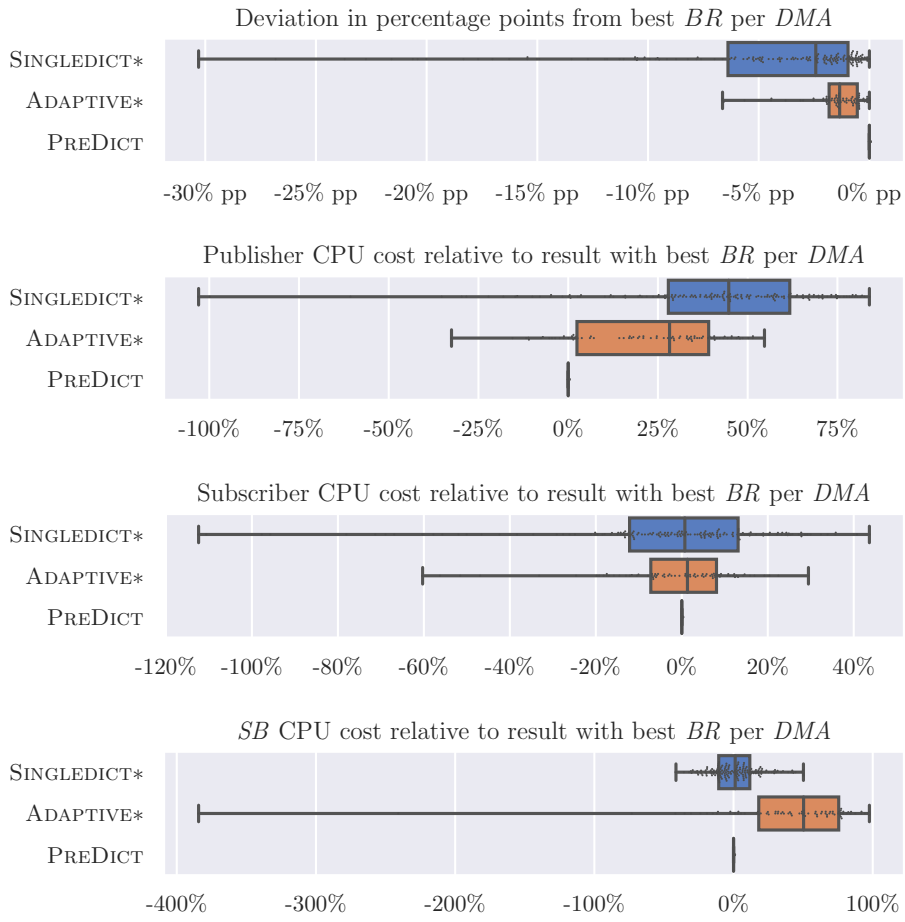


Figure 5.4.3: Worst, median, and best case deviations from the best result in terms of *BR*

5.4.6 Bandwidth reduction over time

Figure 5.4.4 shows both the performance of multiple DMAs and why dictionary maintenance is valuable. At $\approx 600k$ messages, the best SingleDict permutation achieves the highest *BR*; from this point on, the content of the dictionary becomes outdated, and the *BR* decreases. Adaptive and PreDict react by creating new dictionaries and can even improve the bandwidth reductions. The worst case configuration of Adaptive is even worse than SingleDict.

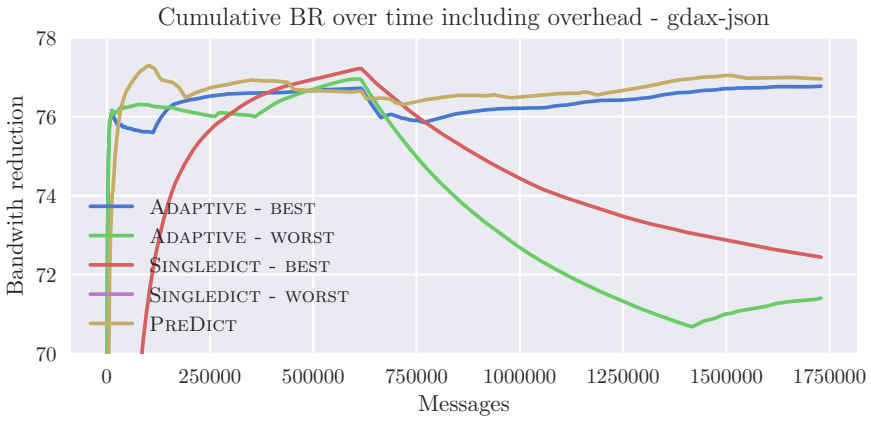


Figure 5.4.4: Bandwidth savings over time

Figure 5.4.5 shows the bandwidth reduction over time of PreDict, Adaptive and FixedRate for the debs2015.csv dataset. This dataset has no schema overhead because of the XML or JSON encoding; hence, the compression is only dependent on the repetitions of the content. PreDict evaluates 162× the compression, see Table 5.4.2, and decides to either prolong the dictionary or create a new one. We can observe that PreDict takes some time to fully adapt to the content of the stream, and then, the *BR* increases. This is a general trend in nearly all datasets, particularly in the Neutron dataset. The gap between the best and worst case SingleDict is approximately 6%.

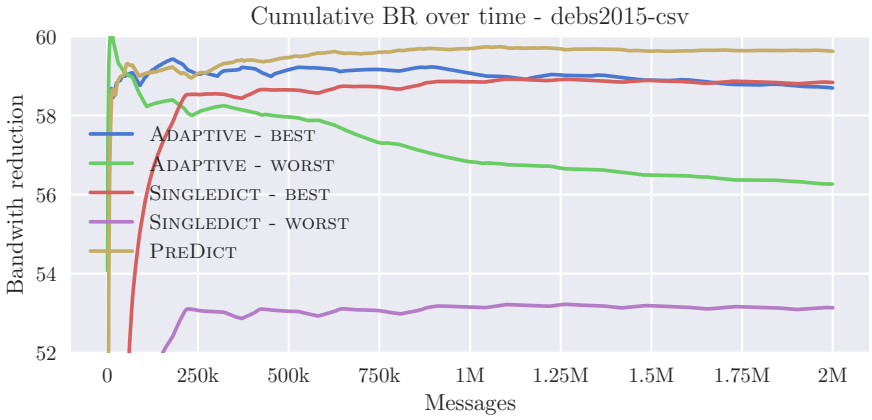


Figure 5.4.5: Bandwidth savings over time

The average *BR* of PreDict is 72.6%, which is 0.3% better than the best permutation of Adaptive and 0.5% better than the best permutation of FixedRate in the case of a

topology that has one publisher and one subscriber. Note that in both ADAPTIVE and FIXEDRATE, many permutations are evaluated, and only the best permutation in terms of *BR* is shown. PREDICT represents a single result without configuration. When more publishers are in the topology, PREDICT still achieves a high *BR* while ADAPTIVE and the benchmarks introduce overhead that cannot be amortized by high compression ratios. In a topology with 10k publishers and a single subscriber, PREDICT still achieves a *BR* of 70.7%, whereas all other methods achieve worse performances than off-the-shelf compression DEFLATE.

5.5 Supplementary evaluation

We evaluated another dimension batching. When multiple messages are batched together, the redundancy within a batch increases, hence off-the-shelf compression such as DEFLATE achieves higher compression ratios. In scenarios, such as our motivating examples in Internet of Things (IoT) or smartphone applications, latency is critical. Furthermore not so many messages are exchanged, hence batching would introduce high latency since multiple messages have to awaited to be batched together. However, batching is often used when workloads are not latency sensitive, such as background processing of large chunks of messages. We want to show at which tresholds off-the-shelf compression performs as good as SDC with the related protocol overhead.

Next we want to show the performance of PREDICT by extending Apache Pulsar, a distributed pub/sub system.

5.5.1 Batching of messages

We evaluate another dimension, batching to show the impact on the compression bandwidth reduction. Batching is tradeoff between latency and compression ratio in case of the shelf compression is used. Batching is available in many off-the-shelf pub/sub systems such as Kafka [62] or Apache Pulsar [63]. When Batching is activated, the publisher waits till several messages are published and then sends them to the broker as

a single batch. Furthermore the whole batch is acknowledged at once which improves throughput.

We evaluate the following permutations, see Equation 5.5.1:

$$\begin{aligned} \mathbb{BS} &= \{1, 2, 3, 5, 8, 13, 21, 34, 55\} \\ \mathbb{DMA} &= \{\textsc{PreDict}, \textsc{Deflate}\} \\ \mathbb{EV} &= \mathbb{BS} \times \mathbb{DMA} \end{aligned} \tag{5.5.1}$$

When multiple messages can be batched together, off-the-shelf compression such as Deflate performs better since the redundancy between messages can be used to reduce the size. Figure 5.5.1 shows that in the case where one publisher and subscriber, using PreDict very high compression can be achieved from the first message on. But off-the-shelf compression such as Deflate works nearly as good as PreDict when 20 or more messages can be batched together.

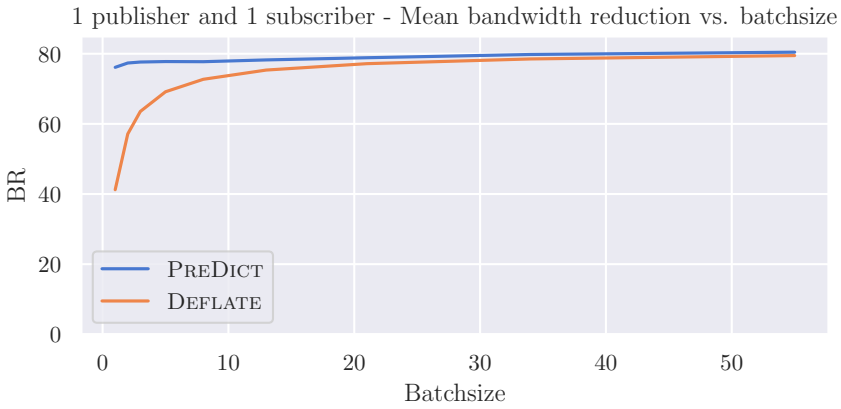


Figure 5.5.1: Batch compression, Deflate and PreDict

In case we have 1k publishers and just a single subscriber, we need to acknowledge the overhead of sharing the dictionary. The dictionary can only be amortized through high compression ratios on messages, but in this scenario, each publisher only shares $\frac{|M|}{|\mathbb{P}|}$. Since less messages are sent per publisher, batching would incure high latency.

For a real world use case batching of messages on the publisher side must be questioned.

The case of high numbers of publishers and a single subscriber are often the case in IoT. As an example, sensors in the field report soil moisture every hour. Batching of 10 messages together would result in bandwidth reductions similar to PreDict but the farmer may loose actionable information for many hours and the irrigation system may not be triggered in time. Nevertheless, if such a scenario is feasible, we can see that PreDict is ≈ 20% in the case of a batchsize of 1. At batchsize 10, no significant advantages of PreDict can be seen.

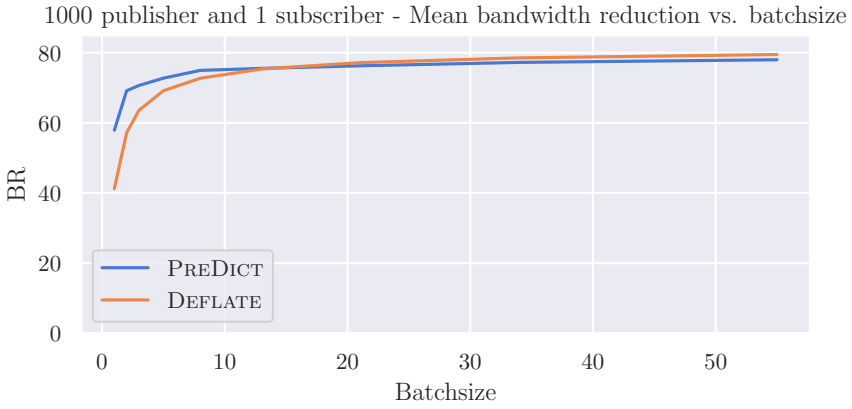


Figure 5.5.2: Batch compression, Deflate and PreDictconfigured with 1k publishers

Furthermore we include the results when a small batchsize of 2 is possible. Deflate, which achieves 41.2% bandwidth reduction in the scenario of batchsize 1, increases the bandwidth reduction to 57.1%. When there are 10k publishers and a single subscriber, this number is even better than PreDict which achieves 46.5%.

Additional results with different batchsizes are available in the Appendix B.

5.5.2 Extension of Apache Pulsar

We extended Apache Pulsar [63] to support SDC. The broker starts an additional service per topic which executes the PreDict. Pulsar is fully asyncronous using evented I O. Hence our extension is also implemented in that way. When a new dictionary is available, publishers and subscribers are notified. When they are ready, the pull the

dictionary from the broker. The compression and decompression is done in a user invisible way. The only difference is that when a publisher creates a new topic, it sets a flag that it should be compressed using SDC.

For the evaluation we evaluate different publisher to subscriber ratios and different compression methods. We evaluate PreDict, off-the-shelf compression using Deflate and no compression, see Equation 5.5.2. The rate is expressed in bits per second (bps).

$$\begin{aligned} |\mathbb{P}| &= \{1, 2, 3, 5, 8, 13, 21, 34, 55\} \\ |\mathbb{S}| &= \{1, 2, 3, 5, 8, 13, 21, 34, 55\} \\ \mathbb{R} &= \{64K, 128K, 256K, 512K, 1M, 2M, 4M, 8M, 16M, 32M\} \\ \mathbb{DMA} &= \{\text{PreDict}, \text{Deflate}, \textit{uncompressed}\} \\ \mathbb{EV} &= |\mathbb{P}| \times |\mathbb{S}| \times \mathbb{DMA} \times \mathbb{R} \end{aligned} \tag{5.5.2}$$

First, we start the Pulsar broker on a VM backed by a node with a spinning disk. All topics are persistet to disk, hence the drive speed is also important. Since messages are compressed and have a smaller size, we expect less I
O. Listing [**lst:broker benchmark**] shows the benchmark results of the disk. Then we copy the dataset to the publishers. After the dataset is copied, we set up the bandwidth limitations between the broker and clients. Listing 5.5.2 shows the outgoing bandwidth limitations enacted specifically to a client with a specific ip adress. Listing 5.5.3 shows the outgoing bandwidth limitations from the client to the broker.

Listing 5.5.1: Broker disk benchmark

```
time sh -c "dd if=/dev/zero of=testfile bs=512 count=10000 oflag=direct && sync"
57.720s and 89.1kB/s

time sh -c "dd if=/dev/zero of=testfile bs=1000k count=1k && sync"
5.893s and 182 MB/s
```

Listing 5.5.2: Bandwidth limitations at broker

```
ubuntu@bigenv-pulsar-broker-13:\$ tcshow eth0
```

```
{
  "eth0": {
    "outgoing": {
      "dst-network=172.subscriber.ip/32, protocol=ip": {
        "filter_id": "800::803",
        "delay'': "50.0ms",
        "rate'': "64K",
      }
    },
      "incoming": { }
    }
}
```

Listing 5.5.3: Bandwidth limitations publisher or subscriber

```
ubuntu@bigenv-pulsar-client-724:\$ tcshow eth0
{
  "eth0": {
    "outgoing": {
      "dst-network=172.broker.ip/32, protocol=ip": {
         "filter_id": "800::803",
         "delay": "50.0ms",
         "rate": "64K"
       }
     },
      "incoming": { }
   }
}
```

Once the datasets are copied to the publishers, the bandwidth limitations are enacted, the controller publishes a message on a specific topic. When the message is received, each publishers starts to publish $\frac{20.000}{|\mathbb{P}|}$ messages and each subscriber waits until 20.000 messages are received. We measure the timespan between start and finish. Each message is sent syncronously, hence the broker confirms the reception of the message after it has been persisted in the log. Each message received by the subscriber is acknowledged to the broker, this advances the cursor in the log to the next message.

Slow network connections often lead to timeouts in the evaluation. If we run into a timeout, we cancel the benchmark and rerun it. At 1Mbps, we had to rerun the benchmark on average 5 times until we had a usable result. Figure 5.5.3 shows the performance improvements for consumers when PREDICT is used in a bandwidth limited scenario using the Twitter dataset. The bandwidth is limited to 1Mbps for all clients, which

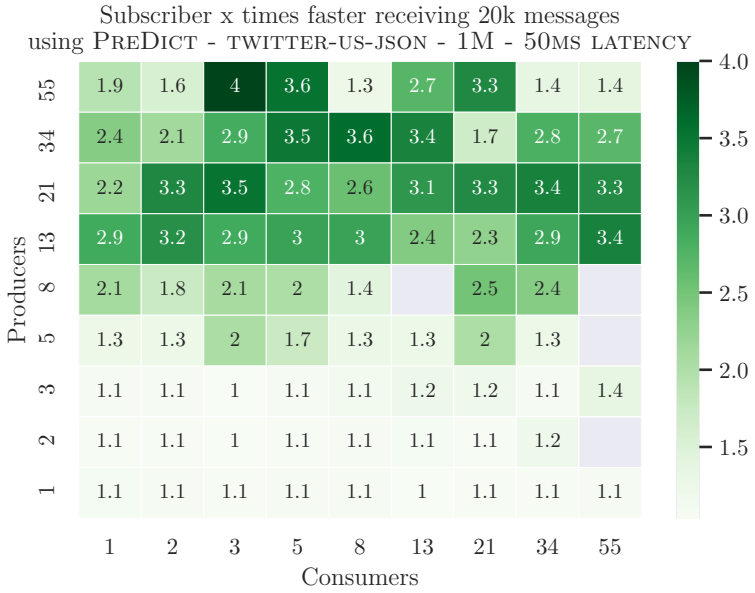


Figure 5.5.3: Consumers faster

corresponds to a rural 3G connection. All producers are started at the same time, so in case 3 producers send messages, each producer sends $\frac{20000}{3}$ messages. We can see that when 5 publishers start sending at the same time, we hit the bandwidth limitions on the subscriber side, hence compression of messages pays off. The more publishers, the faster messages arrive at the broker, the higher the speedup. The maximum speedup was 4 for the Twitter dataset which using PreDict achieves 84.4% bandwidth reductions.

Figure 5.5.4 shows the performance when bandwidth is limited in both directions to 4Mbps which corresponds roughly to a HSDPA mobile connections. We see that the connections to the consumers become a bottleneck once we have 34 publishers sending messages, which is expected since in the 1Mbps experiment we hit the bottleneck at around 5 publishers.

Overall we can see that when bandwidth is not constraint, we achieve around 1.1× speedup. The reason is that messages are persisted faster to disk on the broker. Reducing the size of the messages using SDC also improves the performance of disk based message queues.

Additional results are available in the Appendix C. We show additionally the subscriber

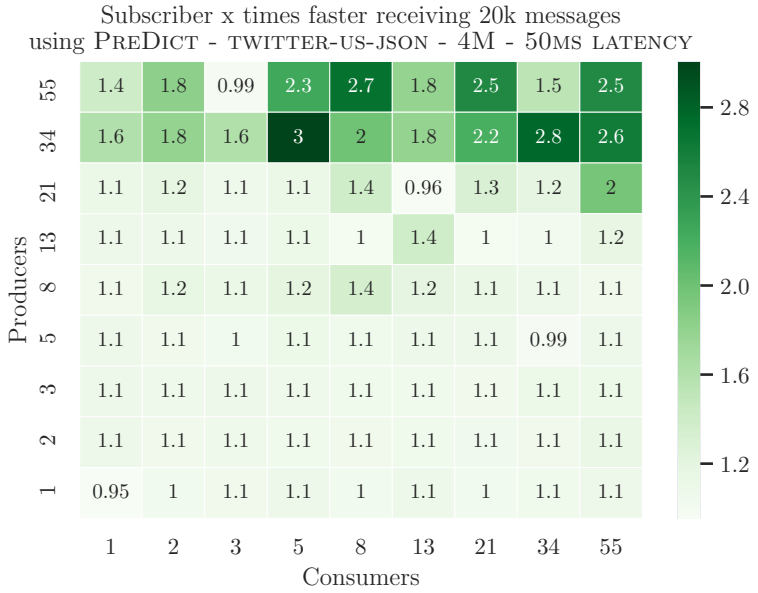


Figure 5.5.4: Consumers faster

speedup and also publisher speedup at different bandwidth constraints.

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.48	0.56	0.51	0.42	1.09	1.32	0.31	0.53	0.31	0.54	0.34	1.49	0.75	2.25	1.29	0.34	0.33	0.24	1.12	2.22	0.8/0.5/0.6
VCDIFF	0.43	0.50	0.43	0.25	1.17	3.08	0.25	0.56	0.26	0.47	0.23	2.73	1.14	4.04	2.79	0.33	0.29	0.20	1.73	3.41	1.2/0.5/1.2
FIXEDRATE*	0.39	0.54	0.37	0.32	0.99	2.76	0.33	0.45	0.25	0.51	0.23	5.49	0.93	7.54	1.34	0.29	0.18	0.15	0.84	2.73	1.3/0.5/1.9
SINGLEDICT*	1.08	1.43	0.97	0.44	7.01	3.90	0.60	2.34	0.51	1.54	0.44	18.52	3.59	21.40	7.20	0.39	0.39	0.36	3.39	20.35	4.8/1.5/6.7
ADAPTIVE*	0.53	0.96	0.51	0.27	6.54	4.61	0.63	1.19	0.58	0.78	0.40	8.08	2.41	13.80	5.52	0.39	0.24	0.33	1.76	6.90	2.8/0.9/3.6
PREDICT	0.69	1.02	0.60	0.39	2.15	3.24	0.46	1.10	0.39	0.85	0.54	5.05	2.00	7.41	4.29	0.47	0.36	0.30	1.45	6.51	2.0/0.9/2.1
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.17	0.18	0.15	0.15	0.37	0.50	0.13	0.18	0.07	0.17	0.14	0.64	0.25	0.81	0.46	0.18	0.11	0.08	0.36	0.88	0.3/0.2/0.2
VCDIFF	0.23	0.21	0.21	0.15	0.75	0.85	0.10	0.25	0.11	0.29	0.13	1.11	0.40	1.13	0.77	0.17	0.13	0.09	0.70	1.38	0.5/0.2/0.4
FIXEDRATE*	0.15	0.23	0.16	0.11	0.15	0.70	0.09	0.15	0.12	0.21	0.09	0.68	0.42	1.25	0.52	0.11	0.08	0.06	0.34	0.96	0.3/0.2/0.3
SINGLEDICT*	0.12	0.17	0.12	0.09	0.33	1.06	0.13	0.20	0.22	0.24	0.10	1.09	0.51	1.86	0.65	0.14	0.08	0.14	0.50	1.39	0.5/0.2/0.5
ADAPTIVE*	0.12	0.19	0.12	0.08	0.33	1.12	0.13	0.15	0.22	0.18	0.10	0.92	0.52	1.90	0.69	0.15	0.10	0.12	0.42	1.25	0.4/0.2/0.5
PREDICT	0.12	0.18	0.11	0.08	0.19	0.91	0.13	0.19	0.15	0.18	0.10	0.75	0.48	1.48	0.57	0.15	0.10	0.10	0.32	1.03	0.4/0.2/0.4
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.23	0.09	0.36	0.25	464.41	61.92	17.25	0.23	16.74	0.19	1.10	2718.15	0.64	1870.97	5.02	31.18	0.09	0.02	61.14	2.31	262.6/1.7/697.7
SINGLEDICT*	2.03	3808.02	2.17	0.67	22.00	2.43	0.14	1.06	0.14	1.88	0.39	227.36	5.28	151.00	19.31	0.33	0.36	0.06	1355.44	35.68	281.8/2.1/861.0
ADAPTIVE*	0.29	0.91	1.44	0.33	417.56	1.73	0.63	1.29	0.23	1.41	0.09	165.84	6.35	174.42	31.91	0.43	0.11	0.03	723.15	17.92	77.3/1.4/178.5
PREDICT	0.13	0.06	0.05	0.09	0.58	0.20	0.03	0.09	0.03	0.15	0.08	2.30	0.16	1.29	0.65	0.09	0.08	0.03	18.88	1.04	1.3/0.1/4.1
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	37.2	32.0	35.9	32.5	68.4	47.3	34.2	45.4	7.7	53.3	26.6	76.5	48.2	53.2	55.7	23.4	20.0	9.4	56.0	61.8	41.2/41.2/41.2
VCDIFF	54.1	25.9	53.1	53.1	76.5	21.7	0.7	40.7	0.8	66.0	32.1	56.3	24.8	24.1	15.4	30.1	8.8	0.0	54.0	54.6	34.6/34.6/34.6
FIXEDRATE*	71.2	62.8	71.3	70.1	92.6	62.3	60.8	77.1	50.3	77.1	74.1	84.4	66.8	66.5	69.0	67.0	63.3	45.1	73.8	77.9	69.2/-69.1/-1314.4
SINGLEDICT*	86.2	82.3	85.3	86.3	92.0	61.8	57.2	81.3	46.9	85.4	76.9	84.5	72.9	66.8	76.4	61.6	74.3	58.2	83.0	83.2	75.1/67.7/1.3
ADAPTIVE*	86.8	82.1	85.2	87.5	93.8	62.7	59.1	82.9	48.4	86.4	77.4	85.6	73.1	67.4	76.8	65.0	74.8	55.9	82.9	84.2	75.9/65.3/-30.3
PREDICT	87.0	82.1	86.7	87.3	91.6	62.8	59.4	82.5	49.0	86.8	76.6	85.1	72.8	67.0	76.0	69.8	74.9	56.7	82.7	84.4	76.1/72.0/44.2

* Best of all permutations in terms of bandwidth reduction

Table 5.4.4: Result summary (including all overhead) for batchsize 1

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.33	0.44	0.30	0.26	0.51	0.88	0.19	0.37	0.24	0.32	0.22	1.26	0.64	2.01	0.86	0.26	0.24	0.22	0.52	1.94	0.6/0.4/0.5
VCDIFF	0.36	0.49	0.31	0.24	0.49	1.77	0.28	0.38	0.27	0.44	0.24	1.79	1.05	3.18	1.86	0.32	0.24	0.18	0.84	2.81	0.9/0.4/0.9
FIXEDRATE*	0.35	0.49	0.25	0.24	1.11	3.37	0.27	0.40	0.20	0.50	0.24	6.60	0.94	7.79	1.36	0.28	0.17	0.16	0.88	2.59	1.4/0.4/2.1
SINGLEDICT*	1.18	1.39	0.95	0.51	6.53	4.78	0.60	2.30	0.44	1.54	0.76	8.51	3.69	20.49	9.42	0.38	0.40	0.22	3.40	14.30	4.1/1.5/5.3
ADAPTIVE*	0.39	0.71	0.43	0.29	3.58	4.07	0.49	1.03	0.31	0.75	0.39	7.27	2.44	16.04	3.98	0.41	0.26	0.44	2.65	7.01	2.6/0.7/3.7
PREDICT	0.57	0.91	0.49	0.29	2.74	2.55	0.41	0.98	0.35	0.72	0.45	4.92	1.87	7.93	4.84	0.36	0.31	0.25	1.24	6.05	1.9/0.8/2.2
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.09	0.14	0.08	0.08	0.15	0.23	0.06	0.12	0.09	0.13	0.07	0.24	0.17	0.40	0.21	0.10	0.06	0.08	0.15	0.38	0.2/0.1/0.1
VCDIFF	0.20	0.20	0.17	0.14	0.48	0.53	0.09	0.20	0.09	0.30	0.13	0.99	0.37	0.87	0.52	0.15	0.10	0.06	0.42	1.21	0.4/0.2/0.3
FIXEDRATE*	0.13	0.20	0.11	0.09	0.14	0.80	0.11	0.15	0.11	0.18	0.07	0.65	0.46	1.19	0.50	0.09	0.08	0.07	0.31	0.90	0.3/0.1/0.3
SINGLEDICT*	0.13	0.18	0.12	0.08	0.31	1.13	0.11	0.19	0.31	0.19	0.09	0.90	0.51	1.89	0.81	0.13	0.08	0.08	0.50	1.24	0.4/0.2/0.5
ADAPTIVE*	0.12	0.21	0.10	0.07	0.27	1.09	0.13	0.17	0.15	0.17	0.08	0.93	0.56	1.75	0.65	0.13	0.08	0.09	0.43	1.23	0.4/0.2/0.5
PREDICT	0.11	0.17	0.12	0.07	0.19	0.85	0.13	0.16	0.13	0.19	0.09	0.76	0.47	1.49	0.63	0.14	0.09	0.08	0.28	0.99	0.4/0.2/0.4
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.08	0.19	0.06	0.06	368.56	85.75	0.26	0.10	0.32	0.29	1.84	2623.69	0.33	2080.70	9.57	30.14	0.05	0.01	1238.21	3.60	322.2/0.3/735.3
SINGLEDICT*	4.54	3483.52	4.09	1.45	40.81	3.70	0.30	2.17	0.33	3.76	0.72	178.18	9.25	96.04	31.17	0.96	0.76	0.10	1311.03	73.35	262.3/3.9/791.4
ADAPTIVE*	0.32	0.24	1.16	0.62	30.69	1.17	0.41	0.57	0.03	0.68	0.23	95.10	7.78	188.13	20.44	3.07	0.46	0.04	1641.41	24.70	100.9/0.9/356.2
PREDICT	0.02	0.04	0.02	0.01	0.37	0.09	0.01	0.02	0.01	0.03	0.01	1.92	0.12	0.87	0.44	0.02	0.02	0.00	18.46	0.96	1.2/0.0/4.0
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	61.3	53.3	60.7	59.2	79.7	53.8	47.2	62.8	25.7	68.9	51.3	79.4	58.8	59.8	64.2	43.7	44.8	27.5	69.0	71.6	57.1/57.1/57.1
VCDIFF	57.9	35.3	56.6	56.8	84.8	23.3	1.6	49.1	1.8	71.4	46.3	59.6	30.2	28.3	20.2	41.2	15.2	0.0	57.2	59.5	39.8/39.8/39.8
FIXEDRATE*	79.5	69.7	79.6	78.7	93.3	62.7	60.3	78.1	50.5	82.1	78.4	85.1	68.7	67.9	71.2	71.5	68.2	48.3	80.2	80.0	72.7/-37.3/-1028.7
SINGLEDICT*	87.1	82.7	86.6	88.0	92.5	62.2	60.0	82.6	49.7	86.5	80.1	84.9	73.5	68.1	76.5	68.3	75.1	59.9	83.2	83.5	76.5/69.1/2.5
ADAPTIVE*	87.4	81.3	87.9	88.8	93.5	63.2	61.1	83.6	50.8	87.6	81.3	86.2	73.7	68.8	76.2	75.6	76.1	58.2	84.7	84.5	77.5/68.6/-11.4
PREDICT	87.6	82.7	87.7	88.8	93.1	62.6	61.1	83.5	50.8	87.4	80.5	85.1	73.4	68.3	76.8	74.6	75.9	58.2	84.0	84.5	77.3/73.8/46.5

* Best of all permutations in terms of bandwidth reduction

Table 5.5.1: Result summary (including all overhead) for batchsize 2

Chapter 6

TaPD: Topology-aware PreDict

Many pub/sub systems are distributed and the individual brokers are connected through an overlay network. Publishers and subscribers connect to the brokers and messages are routed along a specific path from the publishers to the subscribers through the overlay.

The challenge with Shared Dictionary Compression (SDC) in pub/sub systems is that on links with a low amount of messages, the introduced overhead of dictionary compression cannot be amortized. In this chapter, we propse TaPD, an approach to make PreDict aware of the underlying topology. TaPD defaults to compression algorithms which do not introduce overhead or prolonging dictionaries even longer to reduce overhead at the cost of lower compression ratios. Furthermore, we use brokers to re-code messages in the network to keep the overall bandwidth reductions high. The goal is that no overhead, which is inpossible to amortize, is imposed for all clients.

6.1 Overview

In TaPD, we address the problem of high overhead by adapting to the specifics of a graph topology. We extend our previous approach PreDict, with brokers that have the ability to re-code messages. As an example, in case a publisher sends a message

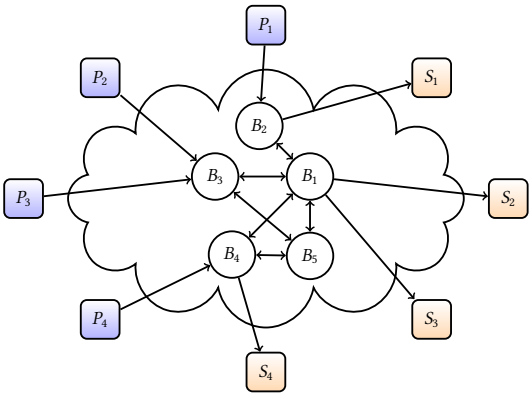


Figure 6.1.1: Clients connections connected to overlay

in DEFLATE, the broker can uncompress the message and then compress using a new dictionary and forward it using dictionary compression. Furthermore, we allow multiple dictionaries to be active at the same time. This flexibility allows us to keep dictionaries longer active in subparts of the overlay and amortize the overhead of the dictionary despite the degradation in compression ratio.

The system model we assume for TAPD is a distributed pub/sub overlay, see Figure 6.1.1. (Compare to the system model of standalone PREDICT, see Section 5.1) Each publisher and each subscriber is connected to one broker. The brokers are connected to a certain degree with other brokers.

6.2 Topology-aware compression in publish/subscribe

First, we introduce the overlay we assume for the evaluation of TAPD. Then, we explain the evaluation metrics and formulate bandwith reduction in the graph as an optimization problem. Finally, we introduce TAPD, which makes our previous approach PREDICT topology aware. The system model we assume is that of a distributed pub/sub systems. Pub/sub systems exhibiting similiar properties are Padres [6], Scribe [64], PastryStrings [65] or Siena [5, 24]. Additionally we assume that the system includes a leader election system such as Paxos [66] or Raft [67] and a consistent key-value store such as Zookeeper [68].

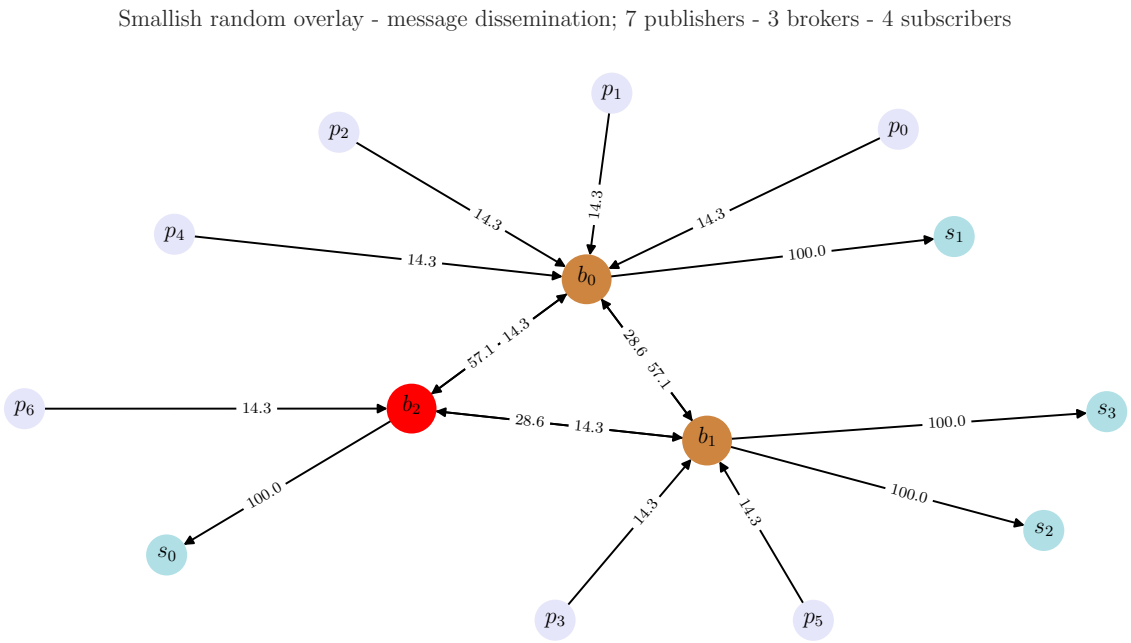


Figure 6.2.1: Overlay for message dissemination

6.2.1 Topology

Each publisher and subscriber is connected to a single broker. Each broker is connected to a certain number of other brokers. One of the brokers is selected to also be the Sampling Broker (SB) of the topic. Our assumption is that each message is routed along the shortest path through the overlay. Furthermore, the dictionary is shared through the overlay and routed along the shortest path to each publisher and subscriber by the sampling broker.

The overlay resembles a graph, see Figure 6.2.1. Each publisher (P_i) is connected to one broker (B_i). Messages flow from publishers towards subscribers (S_i). Each subscriber receives 100% of the messages. The publishers in this example publish each the same amount of messages. The messages are routed along the shortest path to the subscribers. The connections from publishers to brokers and from brokers to subscribers are directed. The connections between the brokers are undirected. The values towards the target on the edges denote percentages of total messages published on a topic.

A similiar graph is formed for the dissemination of dictionaries, see Figure 6.2.2. One broker acts as the SB, colored in red in the figure. Each client (publishers and subscribers)

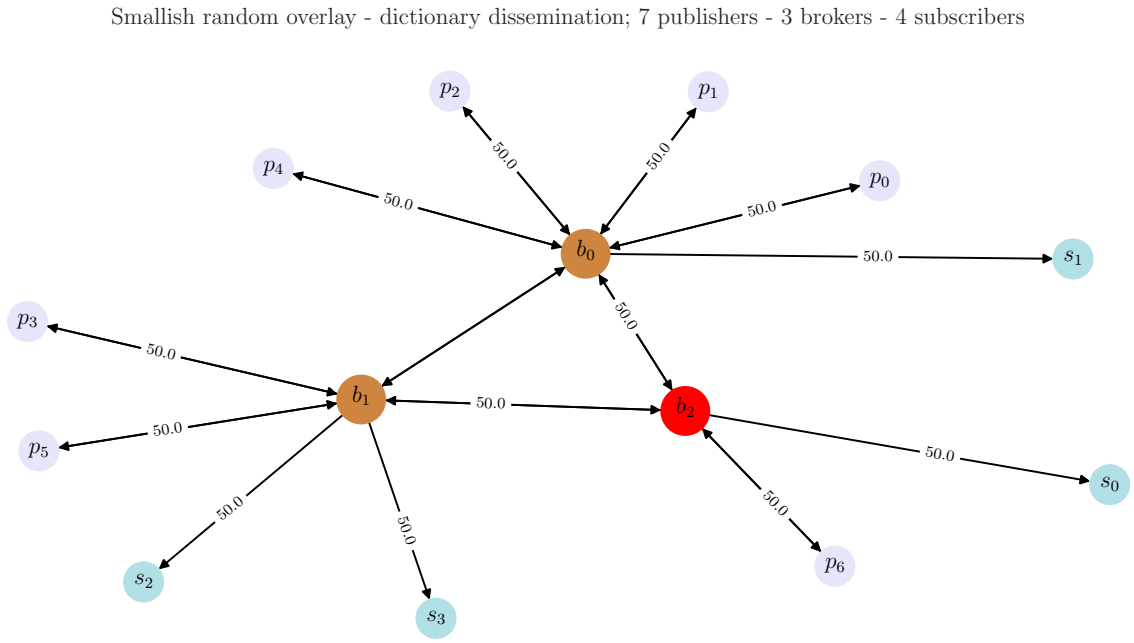


Figure 6.2.2: Overlay for dictionary dissemination

receives the dictionary. The values on the edges towards the target indicate the size of the dictionary. In this exeample, we use dictionaries that are 50kB. Because dictionaries are sent along the shortest path, they may take different routes than messages. On the edge betwenn B_0 and B_1 no dictionary is sent. The overhead from sharing a dictionary may not amortize on the individual edges between the brokers. In that case, since the dictionaries are shared through other links, the overall bandwidth reductions on other links is higher. This is not a concern since other connections to the broker are not impacted by overhead.

6.2.2 Evaluation metrics

We model the pub/sub overlay as graph (G) with the clients and brokers as vertices (V) and edges ($\mathbb{E}(i,j)$) as a link from V_i to V_j. The bandwidth reduction of the individual links is calculated the following way, see Equation 6.2.1. The size of the uncompressed messages. To calculate the bandwidth reduction on a edge $\mathbb{E}_{(i,j)}.br$ between two vertices

V, we need

$$
\begin{aligned}
\mathbb{V} &= \mathbb{P} \cup \mathbb{S} \cup \mathbb{B} \\
\mathbb{E}_{(i,j)}.u &= |\mathbb{M}_{(i,j)}.u| \\
\mathbb{E}_{(i,j)}.c &= \begin{cases} |\mathbb{M}_{(i,j)}.u| \times \mathbb{C}.dc + |DC_{(i,j)}| & \text{if SDC} \\ |\mathbb{M}_{(i,j)}.u| \times \mathbb{C}.d & \text{if DEFLATE compression} \\ |\mathbb{M}_{(i,j)}.u| & \text{if No compression} \end{cases} \\
\mathbb{E}_{(i,j)}.br &= 100.0 - (100/B_{(i,j)}.u \times B_{(i,j)}.c)
\end{aligned}
\tag{6.2.1}
$$

The total bandwidth reduction (BR) represents the total overall bandwidth using compression compared to using no compression.

$$
\begin{aligned}
BG.c &= \sum_{i=1,j=1}^{i=i.max,j=j.max} \mathbb{E}_{(i,j)}.c \\
BG.u &= \sum_{i=1,j=1}^{i=i.max,j=j.max} \mathbb{E}_{(i,j)}.u \\
BR &= 100 - \frac{100\ BG.c}{BG.u}
\end{aligned}
\tag{6.2.2}
$$

6.2.3 Optimization problem

We can formulate the goal of TAPD as an optimization problem. We classify the edges of the graph into three different categories. Edges which are between publishers and brokers, subscribers and brokers and between two brokers, see Equation 6.2.3.

$$
E_{(i,j)} = \begin{cases} P\mathbb{E}_{(i,j)}, \text{Publisher edge} & \text{if } \mathbb{V}_i \in \mathbb{P} \text{ or } V_j \in \mathbb{P} \\ S\mathbb{E}_{(i,j)}, \text{Subscriber edge} & \text{if } \mathbb{V}_i \in \mathbb{S} \text{ or } V_j \in \mathbb{S} \\ B\mathbb{E}_{(i,j)}, \text{Broker edge} & \text{if } \mathbb{V}_i \in \mathbb{B} \text{ and } V_j \in \mathbb{B} \end{cases}
\tag{6.2.3}
$$

Our goal for TAPD is to maximise the *BR* subject to the following constraints, see Equation 6.2.4. Edges between publishers and brokers and brokers and subscribers should have a lower bandwidth than DEFLATE. The variables TAPD can choose is on which edges to use SDC vs. DEFLATE, expressed as a integer variable ($E_{(i,j)}.usedc$. When SDC is used, how much the dictionary should be prolonged (dictionary degradation $E_{(i,j)}.degrade$). $CR.deflate$ denotes the compression ratio when DEFLATE is used. $CR.dc$ when SDC is used.

$$
\begin{aligned}
\underset{(E_{(i,j)}.usedc, E_{(i,j)}.degrade)}{\text{maximise}} \quad & 100 - \frac{100 \times (\sum \mathbb{E}_{(i,j)}.c + \sum \mathbb{E}_{(i,j)}.d)}{\sum \mathbb{E}_{(i,j)}.u} \\
\text{subject to} \quad & (|PE_{(i,j)}.u| \times CR.dc \times E_{(i,j)}.degrade + |DC_{(i,j)}.d|) \times E_{(i,j)}.usedc + \\
& (|PE_{(i,j)}.u| \times CR.deflate) \times (E_{(i,j)}.usedc - 1) \leq PE_{(i,j)}.u) \times CR.deflate \\
& \\
& (|SE_{(i,j)}.u| \times CR.dc \times E_{(i,j)}.degrade + |DC_{(i,j)}.d|) \times E_{(i,j)}.usedc + \\
& (|SE_{(i,j)}.u| \times CR.deflate) \times (E_{(i,j)}.usedc - 1) \leq SE_{(i,j)}.u) \times CR.deflate \\
& \\
& E_{(i,j)}.usedc = \{0, 1\} \\
& E_{(i,j)}.degrade \leq 1 \\
& E_{(i,j)}.degrade > 0
\end{aligned}
\tag{6.2.4}
$$

TAPD decides wheter an edge should use a dictionary ($E_{(i,j)}.usedc$) and how much the dictionary is degraded ($E_{(i,j)}.degrade$).

To solve this using a solver, the following additional information would be needed upfront: How many messages are published per publisher and the overall message size ($|PE_{(i,j)}.u|$). And how many messages in total are published ($|SE_{(i,j)}.u|$). Furthermore a model is needed for dictionary degradation ($E_{(i,j)}.degrade$). The amount of messages and how much the dictionary has degraded at the point of publication of the message is only available posterior, hence we decided to build a simulation and a model for the dictionary degradation.

6.2.4 Dictionary degradation analysis

TaPD extends the dictionaries way longer than their initially planned lifetime, we need a model for the degradation to simulate our approach. PreDict observes the bandwidth reductions and when additional reductions can be obtained, a new dictionary is being shared in the overlay. Edge brokers would measure the degradation and can react.

Dictionaries typically contain two parts, the schema overhead and short term biases. Short time biases get outdated soon while schema overhead stays the same over a longer time. We observed this parts in PreDict where we use the ratio of these values to predict the active time the dictionary should be used. The ratio depends on the dataset. In some datasets, the dictionary degrades very fast, others stay roughly the same. The reason is that either the short term biases change too fast and these parts of the dictionary get outdated quickly or that there are no biases in the stream.

Table 6.2.1 shows the results we use for the model. In this experiment, we create a single dictionary using PreDict but instead of continously maintaing and refreshing the dictionary, we observe how the first dictionary degrades. In some datasets the dictionary degraded by up to 17.3% while for other datasets the compression ratio did not degrade. Additionally, we looked at the swing of the compression ratio. Every 10k messages we compress 100 messages using the dictionary and record the compression ratio. We take the maximum and minium compression ratio and calculate the difference that is shown in the column SDC swing. We can see swings in terms of compression ratio of up to 43.8% when no dictionary maintenance is active.

6.3 Dictionary maintenance algorithm

We assume that PreDict is executed on the sampling broker. PreDict has also ways to take the overlay into account but for a system-model of cloud-based pub/sub. We assume PreDict is configured as the topology would only contain a single publisher and subscriber. With TaPD, the Dictionary Maintenance Algorithm (DMA) itself does not need to take the overlay into account since we allow re-coding and degradation of

dictionaries in the overlay. TAPD tackles adjustment to the overlay on a different level.

6.4 Evaluation

We evaluated TAPD using a simulation. Simulations allow us to evaluate large topologies. The simulation is implemented in Python [69] and uses the results from the dictionary degradation as an input.

6.4.1 Simulation setup

First we generate a based on a distribution random numbers. This random numbers denote how many messages each publisher sends during the simulation. Equation 6.4.1 shows all the different distributions we simulate. The Zipf distribution is often used for generating workloads for pub/sub systems [56]. We also included Rayleigh and Gamma which we assume are realistic for Internet of Things (IoT) workloads. We also include *Equal*, which assumes that each publisher sends the same amount of messages. We think this is also a realistic scenario for example for IoT, where each sensor sends messages at a fixed rate, e.g., every hour. In that case, each publisher sends approximately the same amount of messages. Further we include *Exponential* for scenarios where a single publisher sends most messages while other publishers send only occational messages.

$$\mathbb{D} = \{\text{Zipf}, \text{Gamma}, \text{Rayleigh}, \text{Exponential}, \text{Normal}, \text{Equal}\} \tag{6.4.1}$$

Second, we construct a graph. The graph is constructed based on 3 configuration variables. How many publishers $\mathbb{P}$, how many subscribers $\mathbb{S}$, how many brokers $\mathbb{B}$ and the connectivity within the broker overlay $\mathbb{C}$. Algorihtm 6.1 shows how we create the graph from these parameters. First we connect broker to random other brokers with links in both directions. Then we connect each publisher to a random broker and each subscriber to a random broker.

Algorithm 6.1: Connect message dissemination graph

```
Data: List of ℙ, List of 𝕊, List of 𝔹, List of ℙ and the connectivity between brokers as C
Result: Overlay as a graph
1  Procedure connect_vertices (ℙ, 𝕊, 𝔹, ℙ, C)
2      g ← 𝔾                                              // Initialize empty graph
       // Connect brokers to a certain degree at random
3      foreach B in 𝔹 do
4          foreach i in {1, 2..C} do
5              RB ← chooseRandom(𝔹)
6              addEdge(g, B, RB)                          // Both directions
7              addEdge(g, RB, B)
8      foreach P in ℙ do
9          RB ← chooseRandom(𝔹)
10         addEdge(g, P, RB)
11     foreach S in 𝕊 do
12         RB ← chooseRandom(𝔹)
13         addEdge(g, RB, S)
14     return g
```

Then we setup all path between publishers and subscribers for message dissemination. Algorihtm 6.2 shows how we setup the shortest path between publishers and subscriber and between the sampling broker and the other brokers. Since the shortest path calculation is expensive, we want to cache the shortest paths in a lookup table.

Algorithm 6.4 shows the main procedure of the simulation. Once we have setup to graph and calculated the routes, we begin generating the workload based on the defined distribution. The wokload contains a sequence of which publisher sends a message. Based on the Table 6.2.1 we extract how many dictionaries were created for the specific dataset. Then we insert for each dictionary dissemination a marker in the sequence of messages.

We begin iterating trough the generated sequence, see Line 6.4.7. In case it marker represents a dictionary, we spread the dictionary to all brokers, see Line 6.4.10. In case a message is received at a publisher, we begin iterating through the edges of the dissemination graph of the publisher and account for the overhead. The first edge is goes from the publisher to first broker. The first thing we check is how many messages

Algorithm 6.2: Initialize simulation

Data: $\mathbb{OL}$ describing the overlay, $\mathbb{S}$ all subscribers, $\mathbb{P}$ all publishers
Result: Mesage and dictionary dissemination graph using shortest path

Procedure *initialize_simulation*($\mathbb{S}, \mathbb{B}, \mathbb{P}, C$)
- $\mathbb{DD} \leftarrow set()$ // Holds dictionary dissemination edges
- $SB \leftarrow chooseSB(\mathbb{B})$
- $\mathbb{G} \leftarrow connect\_vertices(\mathbb{S}, \mathbb{B}, \mathbb{P}, C)$ // Directed graph
- **foreach** $P \in \mathbb{B}$ **do**
  - $sp \leftarrow shortestpath(SB, B, \mathbb{G})$ // Dictionary path SB and P
  - $\texttt{Union}(\mathbb{DD}, sp)$
- $MD \leftarrow dict()$
- **foreach** $P \in \mathbb{P}$ **do**
  - $sp \leftarrow set()$
  - **foreach** $S \in \mathbb{S}$ **do**
    - $tempsp \leftarrow shortestpath(P, S, \mathbb{G})$ // Message paths $\mathbb{P}$ and $\mathbb{S}$
    - $\texttt{Union}(sp, tempsp)$
  - $MD[P] \leftarrow sp$
- **return** $\mathbb{DD}$, MD

Algorithm 6.3: Calculate message size

Data: Current version of dictionary, most recent created by SB and p as parameters
Result: Messagesize in kb

Procedure *calculate_msgsize (currentDict, recentDict, p)*
- // Parameters according to extracted variables, see Table 6.2.1
- $normalizedMsgSize \leftarrow \frac{p.average\_messagesizekb}{100} * 100$
- **if** *currentDict* **then**
  - // Dictionary is available
  - **if** *currentDict = recentDict* **then**
    - // Newest dictionary ⇒ highest compression
    - **return** $normalizedMsgSize * p.sdccompression$
  - **else**
    - // Outdated dictionary ⇒ degraded compression ratio
    - **return** $normalizedMsgSize * p.sdcmin$
- **else**
  - // No dictionary ⇒ use DEFLATE
  - **return** $normalizedMsgSize * p.deflate$

Algorithm 6.4: Simulation

```
Data: 𝕄 representing historic messages
Result: Dictionary size in bytes
Procedure simulatePubsub (𝕆𝕃, 𝕊, 𝔻𝕊, D, DR, p)
    pubstate ← initPublisher(ℙ, p.dictvalidtime); substate ← initPublisher(𝕊)
    brokerstate ← initPublisher(𝔹); 𝔻𝔻, MD ← initializeSimulation(...)
    𝕊𝔼𝕆 ← messageDistribution(D, DR)
    foreach SEQ in 𝕊𝔼𝕆 do
        if SEQ = D then
            // A new dictionary is spread through the overlay foreach E_(i,j) ∈ 𝔻𝔻 do
            if (i ∈ 𝔹 and j ∈ 𝔹) then
                // New dictionary version for each broker
                𝔹_i.dictversion ← newversion
                𝔹_j.dictversion ← newversion
                E_(i,j).d += 1
        else
            𝔼 ← MD[SEQ]                                   // Get cached edges
            foreach E_(i,j) ∈ 𝔼 do
                // A message is sent from publihser to first broker
                if i ∈ ℙ and j ∈ 𝔹 then
                    if ℙ_i.pubcnt ≥ ℙ_i.nextdictmsg then
                        // Treshold for getting new dictionary from 𝔹
                        ℙ_i.dictversion ← 𝔹_j.dictversion
                        E_(i,j).d += 1
                        ℙ_i.nextdictmsg += p.dictvalidtime
                    E_(i,j).u += p.average_messagesize
                    E_(i,j).c += calcMessageSize(ℙ_i.dictversion, p)
                // Message from broker to subscriber
                if i ∈ 𝔹 and j ∈ 𝕊 then
                    // Check if subscriber needs a new dictionary
                    if 𝔹_i.dictversion ≠ 𝕊_j.dictversion then
                        𝕊_j.dictversion ← 𝔹_i.dictversion
                        E_(i,j).d += 1
                    E_(i,j).u += p.average_messagesize
                    E_(i,j).c += calcMessageSize(𝔹_i.dictversion, p)
                // Message from broker to broker
                if i ∈ 𝔹 and j ∈ 𝔹 then
                    // Check if subscriber needs a new dictionary
                    E_(i,j).u += p.average_messagesize
                    E_(i,j).c += calcMessageSize(𝔹_i.dictversion, p)
```

the publisher has already sent. If the publisher has already published enough messages, hence a dictionary is already amortized, the publisher gets a new dictionary. We account for the overhead on the edge.

We check how many messages the publisher has already sent. If the publisher has already sent enough messages, then the dictionary has already amortized or enough bandwidth was saved using DEFLATE and then aquires the latest dictionary available at the broker. As long as the dictionary is the latest, we assume the normal compression ratio, once it gets outdated we assume the degraded compression ratio, see Algorithm 6.3.

In a similiar way, we calculate the bandwidth between brokers. But brokers are assured to have the latest dictionary, hence always the best compression ratio is applied. The computational cost of decompressing of a message compressed using DEFLATE or a degraded dictionary is imposed on the edge broker. Publishers with high message rates will be updated to the latest dictionary version since higher compression ratios will anyway amortize the new dictionary. Publishers with low message rates will be very like to be re-coded from either deflate or degraded dictionaries. Hence, in topologies with Zipf or Exponential distributions, the re-coding overhead will be low.

6.4.2 Results

Table 6.4.2 shows the overall bandwidth reduction results of TAPD at various distributions of publications per publisher. In the experiment, we use 5000 individual publishers, each publishing at different rates. The broker overlay contains 50 brokers which were interconnected with 3 random connections to each other. Additionally, we added 20 subscribers. In total, we sent 1M messages and for the input of the simulations we use the results of the Table 6.2.1.

$$BR.overall = 100.0 - \left(\frac{100 \times \left(\sum \mathbb{M}_{(i,j)}.c + \sum \mathbb{OH}_{(i,j)}\right)}{\sum \mathbb{M}_{(i,j)}.u}\right. \tag{6.4.2}$$

Equation 6.4.2 shows how we calculate the overall compression. First, we classify if the edge is from a publisher to a broker, broker to subscriber or a broker to broker,

see Equation 6.2.3. For each category of links we sum up the uncompressed message size $\mathbb{M}_e.u$ when no compression would be employed and relate this to the sum of the compressed messages $\mathbb{M}_e.c$ including the overhead ($\mathbb{OH}$).

$$BR.mean = \frac{\sum(100 - (\mathbb{M}_{(i,j)}.u \times (\mathbb{M}_{(i,j)}.c + \mathbb{M}_e.dc)))}{|\mathbb{M}_{(i,j)}|} \tag{6.4.3}$$

Table 6.4.1 shows the mean bandwidth reduction per category according to Equation 6.4.3.

In case each publisher sends an equal amount of messages, $BR.mean = BR.overall$.

Table 6.2.1 shows that the mean bandwidth reduction using DEFLATE for *debs2015.json* dataset was around 45%. Figure 6.4.1 shows the boxplots for each of the evaluated distributions. In the Equal case, we see a narrow boxplot showing a bandwidth reduction of around 60% for publishers. The slight deviations are caused when dictionary was already outdated and the publisher continued sending message, hence the degradation is imposed. All subscribers use all dictionaries available and also always the most recent versions, hence the distribution is very small and the bandwidth reductions are around 85%. PREDICT alone, see Table 5.4.4 achieved the bandwidth reductions of 82.5% when only one publisher and one subscriber are present in the topology.

Another interesting observation in Table 6.4.2 is comparing the results of different distrubions. The *debs-json* dataset achieves the bandwidth reduction of 60.2% for publishers when each publisher sends the same amount of messages. In case the amount of messages is Zipf distributed, the bandwidth reductions are as high as 81.5%. This is because a few publishers send most messages, hence they can amortize always fresh dictionaries easily and reach high compression ratios.

When using the Zipf distribution, only a few publishers are responsible for nearly all messages. For these publishers, the dictionary imposed overhead easily amortizes. However, the other publishers which do not send a lot of messages won't ever reach the state where it pays off to use a dictionary. In case a dictionary compression is chosen for this publisher, then we keep the dictionary active for a longer time but at the cost of lower compression ratios. When we look at the overall compression, see Table 6.4.2,

we can see that high bandwidth reductions can be achieved for publishers. The many publishers which send only a few messages and fallback to either DEFLATE or use a degraded dictionary, do not impact the overall bandwidth reductions. Table 6.4.1 shows the mean bandwidth reduction per edge. This metric shows that most links have similar performance as DEFLATEin the case of Zipf distribution.

Figure 6.4.1 shows the distribution of the bandwidth reduction on the individual links on the exemplary *DEBS-json* dataset. We can see that in the broker overlay, since the message dissemination graph differs from the dictionary dissemination graph, single edges may not amortize and even exhibt negative compression ratios. This would be edges which are only used for dictionary dissemination. But other edges would amortize higher since no overhead is imposed.

We show additional results in the Appendix D.1.

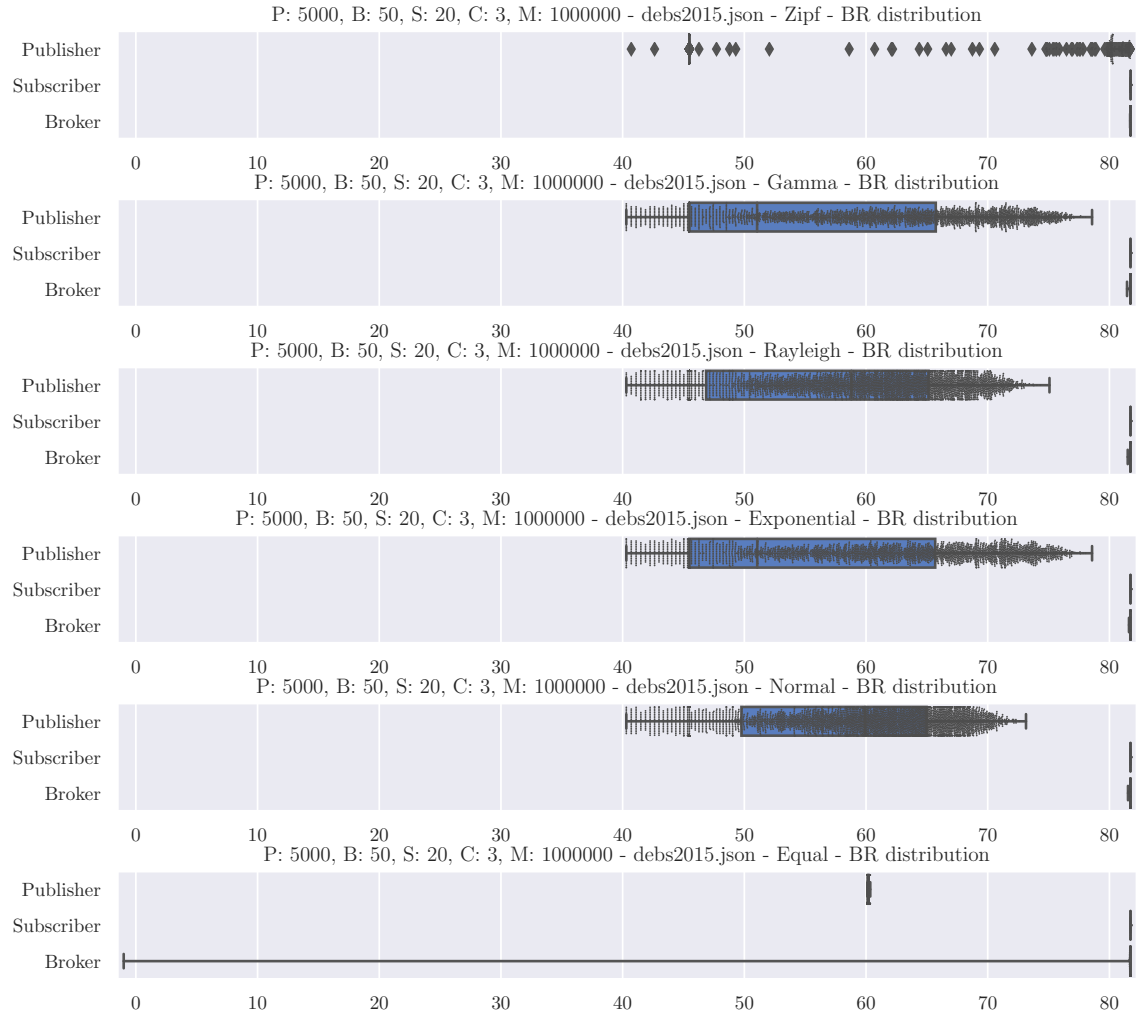


Figure 6.4.1: Debs2015 Json, Bandwidth reduction distribution

Dataset	Ext.	Mean size byte	Mean size deflate byte	Mean size sdc byte	Deflate begin	Deflate degradation	SDC begin	SDC end	SDC degradation	SDC swing	Dict size kB
debs2015	json	529.90	288.88	96.76	45.81	0.33	85.23	81.74	3.49	4.92	2.97
debs2015	pb	174.96	161.54	95.62	9.51	1.84	57.69	45.35	12.35	15.09	2.92
debs2015	csv	190.90	126.28	83.75	35.46	1.61	65.80	56.13	9.67	11.56	2.63
debs2015	xml	710.90	331.58	98.14	53.64	0.28	89.09	86.19	2.89	3.69	3.08
twitter-us	json	3078.41	1221.67	477.70	62.03	1.72	84.88	84.48	0.40	1.96	5.24
satellites	json	211.59	168.90	53.95	19.95	-0.22	77.52	74.50	3.02	3.34	2.72
github	json	3299.38	646.79	407.26	78.24	-2.16	87.22	87.66	-0.44	13.53	4.36
meetup-rsvps	json	1439.24	628.13	340.54	56.08	-0.28	78.81	76.34	2.47	5.92	5.33
Edmonton-Vehicle-Position	json	343.67	231.97	44.53	32.60	0.10	90.43	87.04	3.39	4.44	2.84
NYC-Bike-Live-Station	json	414.42	266.15	52.58	35.79	0.01	89.52	87.31	2.21	3.84	3.13
Chicago-Bike-Stations	json	470.94	295.96	62.88	37.28	0.12	88.40	86.65	1.75	2.67	3.01
tpch-lineitem	tbl	124.64	112.12	51.54	8.54	-1.50	63.44	58.65	4.79	6.13	3.13
meetup-events	json	2199.79	1027.42	671.17	53.12	-0.17	78.55	69.49	9.06	21.77	15.08
METAR-AWC-US	json	516.31	350.20	90.98	32.14	-0.03	83.17	82.38	0.79	2.69	3.52
bartarrivalsschedule	json	1930.51	474.07	148.94	68.64	-6.80	94.76	92.28	2.47	27.11	3.25
meetup-comments	json	968.06	509.89	264.10	47.84	0.51	73.34	72.72	0.62	5.66	4.55
bitcoin-transactions	json	1530.30	826.89	595.25	41.54	-4.43	62.15	61.10	1.05	18.39	2.92
neutron	log	279.26	220.75	116.05	24.74	3.79	75.73	58.44	17.29	43.76	3.22
gdax	json	258.77	189.60	82.27	25.84	-0.89	82.15	68.21	13.94	16.20	2.84
twitch-events	json	1492.06	648.33	229.65	56.69	0.14	85.80	84.61	1.19	28.48	4.16

Table 6.2.1: SDC compression, degradation and swing compared to DEFLATE, Dictionary size according to PREDICT and compressed using DEFLATE, 1M messages

Distribution	Equal				Exponential				Gamma				Normal				Rayleigh				Zipf			
Dataset	𝔹	𝕆	ℙ	𝕊	𝔹	𝕆	ℙ	𝕊	𝔹	𝕆	ℙ	𝕊	𝔹	𝕆	ℙ	𝕊	𝔹	𝕆	ℙ	𝕊	𝔹	𝕆	ℙ	𝕊
DEBS-json	81.7	81.3	60.2	81.7	81.7	81.4	65.0	81.7	81.7	81.4	65.0	81.7	81.7	81.3	61.7	81.7	81.7	81.3	61.7	81.7	81.7	81.7	81.5	81.7
DEBS-pb	45.3	44.6	16.9	45.3	45.3	44.8	24.0	45.3	45.3	44.8	24.0	45.3	45.3	44.7	19.2	45.3	45.3	44.8	19.4	45.3	45.3	45.3	45.1	45.3
DEBS-csv	56.1	55.7	37.2	56.1	56.1	55.8	42.1	56.1	56.1	55.8	42.1	56.1	56.1	55.7	38.9	56.1	56.1	55.7	39.1	56.1	56.1	56.1	55.9	56.1
DEBS-xml	86.2	85.8	67.2	86.2	86.2	85.9	71.5	86.2	86.2	85.9	71.5	86.2	86.2	85.8	68.6	86.2	86.2	85.8	68.6	86.2	86.2	86.2	86.1	86.2
MET-json	82.4	81.7	53.3	82.4	82.4	81.9	59.9	82.4	82.4	81.9	59.9	82.4	82.4	81.7	55.4	82.4	82.4	81.8	55.4	82.4	82.4	82.4	82.1	82.4
twitch-Eve-json	84.6	84.0	56.4	84.6	84.6	84.0	56.9	84.6	84.6	84.0	56.9	84.6	84.6	84.0	56.6	84.6	84.6	84.0	56.6	84.6	84.6	84.6	82.7	84.6
events-json	69.5	69.1	52.6	69.5	69.5	69.2	54.4	69.5	69.5	69.1	54.4	69.5	69.5	69.1	53.3	69.5	69.5	69.1	53.3	69.5	69.5	69.5	68.3	69.5
BTC-json	61.1	60.8	50.0	61.1	61.1	60.9	51.3	61.1	61.1	60.9	51.3	61.1	61.1	60.9	50.4	61.1	61.1	60.9	50.4	61.1	61.1	61.1	60.6	61.1
sat-json	74.5	73.8	40.9	74.5	74.5	73.9	49.2	74.5	74.5	73.9	49.2	74.5	74.5	73.8	43.5	74.5	74.5	73.8	43.6	74.5	74.5	74.5	74.4	74.5
None	84.5	84.2	70.9	84.5	84.5	84.2	73.7	84.5	84.5	84.2	73.7	84.5	84.5	84.2	71.8	84.5	84.5	84.2	71.8	84.5	84.5	84.5	84.4	84.5
comments-json	72.7	72.4	56.3	72.7	72.7	72.4	59.5	72.7	72.7	72.4	59.5	72.7	72.7	72.4	57.3	72.7	72.7	72.4	57.3	72.7	72.7	72.7	72.3	72.7
Gdax-json	68.2	67.6	41.0	68.2	68.2	67.7	47.3	68.2	68.2	67.7	47.3	68.2	68.2	67.6	43.0	68.2	68.2	67.6	43.1	68.2	68.2	68.2	68.0	68.2
rsvps-json	76.3	76.0	63.1	76.3	76.3	76.1	65.6	76.3	76.3	76.1	65.6	76.3	76.3	76.1	63.9	76.3	76.3	76.0	63.9	76.3	76.3	76.3	75.8	76.3
NYC-B-json	87.3	86.6	57.1	87.3	87.3	86.8	64.0	87.3	87.3	86.8	64.0	87.3	87.3	86.6	59.2	87.3	87.3	86.7	59.3	87.3	87.3	87.3	87.0	87.3
Chi-B-json	86.6	86.0	58.3	86.6	86.6	86.1	64.8	86.6	86.6	86.2	64.8	86.6	86.6	86.1	60.3	86.6	86.6	86.0	60.4	86.6	86.6	86.6	86.5	86.6
Neut-log	58.4	57.5	21.3	58.4	58.4	57.6	24.7	58.4	58.4	57.6	24.7	58.4	58.4	57.6	22.5	58.4	58.4	57.5	22.7	58.4	58.4	58.4	56.8	58.4
GTFS-E-json	87.0	86.3	55.2	87.0	87.0	86.5	62.6	87.0	87.0	86.5	62.6	87.0	87.0	86.4	57.5	87.0	87.0	86.4	57.5	87.0	87.0	87.0	86.9	87.0
GH-json	87.7	87.5	79.0	87.7	87.7	87.5	79.0	87.7	87.7	87.5	79.0	87.7	87.7	87.5	79.0	87.7	87.7	87.5	79.1	87.7	87.7	87.6	86.9	87.7
tpch-tbl-tbl	58.6	57.8	21.5	58.6	58.6	58.1	31.4	58.6	58.6	58.0	31.4	58.6	58.6	57.8	24.7	58.6	58.6	57.9	25.0	58.6	58.6	58.6	58.3	58.6
Bart-json	92.3	91.8	71.2	92.3	92.3	91.8	70.6	92.3	92.3	91.8	70.6	92.3	92.3	91.8	71.1	92.3	92.3	91.8	71.1	92.3	92.3	92.3	91.0	92.3
Mean	75.0	74.5	51.5	75.1	75.0	74.6	55.9	75.1	75.0	74.6	55.9	75.1	75.0	74.6	52.9	75.1	75.0	74.6	53.0	75.1	75.1	75.1	74.5	75.1

Table 6.4.1: Overall bandwith reduction, 5k publisher, 50 broker, 20 subscriber, 3 connections between broker, 1M messages

Distribution	Equal				Exponential				Gamma				Normal				Rayleigh				Zipf			
Dataset	𝔹	𝕆	ℙ	$	𝔹	𝕆	ℙ	$	𝔹	𝕆	ℙ	$	𝔹	𝕆	ℙ	$	𝔹	𝕆	ℙ	$	𝔹	𝕆	ℙ	$
DEBS-json	81.1	60.8	60.2	81.7	81.7	56.1	55.3	81.7	81.7	56.1	55.3	81.7	81.7	58.6	57.9	81.7	81.7	58.0	57.3	81.7	81.7	47.6	46.6	81.7
DEBS-pb	45.3	17.8	16.9	45.3	44.9	15.4	14.5	45.3	44.9	15.4	14.5	45.3	45.3	16.4	15.4	45.3	45.3	15.8	14.9	45.3	45.3	9.9	8.7	45.3
DEBS-csv	55.6	37.8	37.2	56.1	56.0	37.3	36.7	56.1	56.1	37.3	36.7	56.1	56.1	37.3	36.7	56.1	56.0	37.0	36.4	56.1	55.9	35.1	34.5	56.1
DEBS-xml	86.2	67.8	67.2	86.2	86.2	63.3	62.6	86.2	86.2	63.3	62.6	86.2	86.2	65.7	65.1	86.2	86.2	65.1	64.5	86.2	85.9	55.3	54.3	86.2
MET-json	82.4	54.2	53.3	82.4	82.4	47.4	46.2	82.4	82.4	47.3	46.2	82.4	82.4	51.1	50.1	82.4	82.4	50.2	49.2	82.4	82.4	35.2	33.7	82.4
twitch-Eve-json	84.6	57.2	56.4	84.6	84.6	57.4	56.5	84.6	84.6	57.4	56.5	84.6	84.6	57.3	56.4	84.6	84.6	57.3	56.4	84.6	84.0	58.0	57.1	84.6
events-json	69.5	53.1	52.6	69.5	69.5	53.7	53.2	69.5	68.9	53.7	53.2	69.5	69.5	53.2	52.7	69.5	69.5	53.2	52.7	69.5	69.5	54.1	53.6	69.5
BTC-json	61.1	50.3	50.0	61.1	61.1	49.0	48.6	61.1	61.1	49.0	48.6	61.1	61.1	49.7	49.3	61.1	61.1	49.5	49.2	61.1	60.0	46.8	46.4	61.1
sat-json	74.5	42.0	40.9	74.5	74.0	35.5	34.2	74.5	74.5	35.4	34.2	74.5	74.5	38.8	37.6	74.5	74.5	37.8	36.7	74.5	74.0	23.4	21.8	74.5
None	83.9	71.3	70.9	84.5	84.5	67.8	67.3	84.5	83.8	67.8	67.3	84.5	84.5	69.7	69.3	84.5	84.5	69.3	68.8	84.5	84.5	61.8	61.0	84.5
comments-json	72.7	56.8	56.3	72.7	72.7	53.9	53.4	72.7	71.6	53.9	53.3	72.7	72.2	55.4	54.9	72.7	72.7	55.1	54.5	72.7	72.7	48.8	48.0	72.7
Gdax-json	68.2	41.8	41.0	68.2	68.2	37.5	36.5	68.2	68.2	37.5	36.5	68.2	68.2	39.6	38.7	68.2	68.2	39.0	38.1	68.2	68.1	29.2	27.9	68.2
rsvps-json	76.3	63.5	63.1	76.3	76.3	61.4	60.9	76.3	76.3	61.4	60.9	76.3	76.3	62.5	62.1	76.3	76.3	62.2	61.8	76.3	76.3	57.5	56.9	76.3
NYC-B-json	87.3	58.0	57.1	87.3	87.3	51.1	50.0	87.3	87.3	51.2	50.0	87.3	87.3	54.8	53.8	87.3	87.3	53.9	52.9	87.3	87.3	38.8	37.3	87.3
Chi-B-json	86.6	59.2	58.3	86.6	86.6	52.3	51.2	86.6	86.6	52.3	51.2	86.6	86.6	56.0	55.0	86.6	86.6	55.1	54.1	86.6	86.6	40.1	38.6	86.6
Neut-log	58.3	22.5	21.3	58.4	58.3	22.9	21.7	58.4	58.3	22.9	21.7	58.4	58.3	22.4	21.3	58.4	58.3	22.3	21.1	58.4	58.4	22.9	21.8	58.4
GTFS-E-json	87.0	56.2	55.2	87.0	87.0	48.8	47.6	87.0	87.0	48.8	47.6	87.0	87.0	52.7	51.6	87.0	87.0	51.8	50.7	87.0	86.9	35.8	34.1	87.0
GH-json	87.6	79.3	79.0	87.7	87.6	79.8	79.6	87.7	87.0	79.8	79.6	87.7	87.6	79.5	79.2	87.7	87.6	79.5	79.3	87.7	87.6	80.7	80.5	87.7
tpch-tbl-tbl	58.6	22.6	21.5	58.6	58.6	19.8	18.6	58.6	58.6	19.8	18.6	58.6	58.6	20.8	19.6	58.6	57.7	20.1	18.9	58.6	58.6	12.9	11.4	58.6
Bart-json	92.3	71.9	71.2	92.3	92.3	73.4	72.8	92.3	92.3	73.4	72.8	92.3	92.3	72.5	71.9	92.3	92.3	72.7	72.0	92.3	92.3	76.2	75.7	92.3
Mean	75.0	52.2	51.5	75.1	75.0	49.2	48.4	75.1	74.9	49.2	48.4	75.1	75.0	50.7	49.9	75.1	75.0	50.3	49.5	75.1	74.9	43.5	42.5	75.1

Table 6.4.2: Average bandwidth reduction per link, 5k publisher, 50 broker, 20 subscriber, 3 connections between broker, 1M messages

Chapter 7

Conclusions

Bandwidth usage is a significant concern, despite network expansion and the availability of high data rates. Mobile internet connections are still metered. Data plans for mobile phones typically have a fixed amount of data included, exceeding the data introduces costs. Also in many Industry 4.0 scenarios, bandwidth limitations are already hit, hence reducing the required bandwidth enables higher sampling rates.

We have shown that by employing Shared Dictionary Compression (SDC) in pub/sub systems, we can significantly reduce the bandwidth compared to off-the-shelf compression such as DEFLATE. Furthermore, we have shown that this approach can be implemented in a user hidden way. All parameters regarding the compression can be automatically determined using polynomial fitting and machine learning. Moreover, it is possible to adapt to varying degrees of publication rates and to complex graph overlays while achieving higher bandwidth reductions than off-the-shelf compression in most of the scenarios.

We think that our results can have a significant impact on the design of future pub/sub systems. To achieve high bandwidth reductions, compression cannot be seen as an independent technique. Compression has to become an integral part of the design in pub/sub systems. Our simulations, evaluations and a prototypical implementation on top of Apache Pulsar show that it is possible to hide all the complexity behind abstractions

and without any additional configuration. We do not see any obstacles to adopt these techniques at the core of pub/sub systems.

An interesting observation is that there is nearly no difference in the bandwidth usage between XML, JSON and Protobuf encoded messages when SDC is used. This is not the case when DEFLATE is used. The schema overhead and common attribute combinations are promoted in the dictionary, which is similar in all schema variations. Hence, the developer can choose the message format which is most convenient without worrying about data size or binary encodings.

The analysis of the actual usage of the dictionary has shown to be very valuable. The most surprising discovery was that there is an upper bound on the cpu usage for compression with large dictionaries which is dependent on the content. The reason for this behavior is that the additional entries tend to be less used. Nevertheless the bandwidth reductions still increases the bigger the dictionary is and these huge dictionaries are actually needed when the dictionary is used for a longer time. It is also important that these dictionaries are sampled over a huge quantity of messages. In the evaluation we have shown that having a big dictionary and creating this dictionary over many messages can still compete regarding overall bandwidth reduction. This can be a favorable scenario from an implementation standpoint since a dictionary can be preshared without any extensions to the pub/sub.

Current limitations and extensions needed for real world usage

Message count requires a central entity which is not possible in a completely distributed pub/sub systems. But counting the messages exactly is not needed, we estimate that a number +/ – 10% will still perform well. We propose an extension of the protocol between the brokers which shares how many messages were observed on which topic. The overhead could be potentially further reduced using counting bloomfilters.

The sampling broker is a role which is executed per topic in the overlay. Choosing the optimal place is subject to several constraints and costs. Constraints are for example the available CPU at the broker and the cost of propagating the dictionaries depends on the position in the graph.

In many pub/sub systems, subscribers are moving. In the case state is needed at the edge brokers, such as proposed by TAPD, the state has to move along the publishers and subscribers. This requires additional entities replicating the state and providing fault tolerance. In the end this is a tradeoff between bandwidth reductions and implementation complexity in the brokers.

We see future work in the following areas:

Delta encoding of dictionaries – Currently we always assume a complete new dictionary. The new dictionary can be compressed using the old dictionary and the size reduction is around 30%. To reduce the size further, the merging of the substrings could be adapted to ignore minor changes in ranks of the dictionary. This would potentially reduce the compression only slightly but would on the other side reduce the size of the delta substantially.

Extend for content-based pub/sub – Our work assumes topic-based pub/sub with "focused channels" and a Dictionary Maintenance Algorithm (DMA) per topic. We believe that SDC for pub/sub can also be extended to content-based pub/sub. In content-based pub/sub subscriptions are expressed as queries and messages are matched against these queries. To cover a wide variety of messages, large dictionaries would be needed. But large dictionaries additionally impose bandwidth overhead when sharing the dictionary, CPU load on the publishers when compressing the messages. Hence, a vast dictionary covering all possible content biases should be avoided. We think that upfront clustering or matching on the type of the message, e.g., a hash of the fields present in the message, could reveal a "focused channel" and subsequently a smaller dictionary. Topic detection techniques could be used [70] to identify "focused channels" and to build custom small dictionaries for higher compression ratios.

Event specific dictionaries – Sensor networks in the field often measure continously the environment on a continous basis. Sudden events like wildfires impose a high load in terms of messages and the messages have to be disseminated within a short timeframe. The message content during such events might be very specific. Hence also a specific dictionary for such events could be beneficial to reduce bandwidth exactly at times when high message rates are needed.

Content-based pub/sub matching – Content-based pub/sub could be extended to match notifications without decompressing. Lightweight compression algorithms are used in databases which allow queries to be executed on top of compressed data. Adapting such a compression scheme to work on streaming data could improve matching speed and reduce bandwidth at the same time.

Tree-aware dictionary dissemination – In TAPD we assume messages are routed along the shortest path. Many pub/sub systems form a tree within the overlay to disseminate the messages. Using a tree it is possible to fullfill ordering guarantees. We think that TAPD could be adapted to trees with various stages of recoding and different lifetimes of dictionaries along the tree branches.

In network batching – When multiple messages can be batched together DEFLATE or other compression algorithms work well. The challenge in pub/sub systems is that the guarantees (e.g., ordering, delivery) are typically enforced on a message level. Batches are often implemented as container on top of messages. For example, in Apache Pulsar, a batch is committed as single message to the log. Compressing and uncompressing messages always needs a dictionary when SDC is used. Hence for logs or latency insesitive workloads, batching reduces the operational complexity at the cost of increased latency. If this tradeoff can be taken is highly dependent on the scenario. As an example, if for the user a latency of 10 seconds is acceptable and around 5 messages a second are sent, the batchsize is already in the region where DEFLATE can perform similar to SDC.

Reduction of I/O for persistent queues – Our implementation on top of Apache Pulsar has shown that I/O is reduced which has a significant effect on spinning disks. Benchmarks on SSDs hinted the potential for a significant improvement of throughput once the bandwidth limitations are hit. We think that an extensive evaluation could reveal additional benefits of SDC especially for write-ahead logs.

Privacy preserving dictionary creation – Messages in pub/sub systems are often exchanged encrypted. Creating dictionaries from encrypted messages is not possible since typically random padding is prepended. A trusted intermediary which has access to the content of the messages could create the dictionary. Secure enclaves, e.g., Intel SGX, could be used as a way to implement such a trusted intermediary.

New compression libraries – New compression libraries emerged recently that support dictionary-based compression. Brotli [41] and ZStandard [71] are notable examples. We believe that both Brotli and ZStandard could reduce the computational overhead for publishers and subscribers significantly because of their highly optimized implementations. Preliminary evaluation has shown not necessarly higher bandwidth reductions, but more evalutions . We believe that our approaches for DMA are sufficiently general to also work with these libraries.

Online optimization – The topologies of pub/sub systems constantly change due to new brokers joining or faults. Also network conditions tend to change and clients are mobile. Hence, assuming a static topology for optimization is not feasible. Changing and transforming the message dissemination topology or changing rolas has a certain cost attached to it. Hence we propose a new way of online optimization of the topology by leveraging deep reinforcement learning on a fitted simulation model. Once a simulation model can be derived from the current system, many scenarios can be simulated by a deep learning agent. Examples would be for example that additional delay is applied to a replicating broker, maybe it makes sense to migrate to a different broker or change the replication to a different broker. Another example would be that suddenly a subscriber is increasing the messaging rate to a point where persisting to disc becomes the bottleneck. In such a scenario the pub/sub system could react by introducing load balancing on this topic or multiple persistent queues on different disks. Another way could be creating a new dictionary to reduce the message sizes to a point where the brokers can keep up with this load. This technique is similar to reinforcement learning agents that are trained to play video games [72]. Once an agent is trained with all the scenarios and adaptations to the system, it can react very quickly to changes and adapt the pub/sub system to varying load.

List of Acronyms

SDC Shared Dictionary Compression 97

Pub/sub Publish/Subscribe

DMA Dictionary Maintenance Algorithm 99

SB Sampling Broker 81

LCS Longest Common Substrings 49

IoT Internet of Things 86

MQTT Message Queuing Telemetry Transport 21

JSON JavaScript Object Notation 6

XML eXtensible Markup Language 6

CSV Comma-separated values 29

M2M Machine-to-Machine 4

LTE Long-Term Evolution 4

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "Design and Evaluation of a Wide-Area Event Notification Service." In: *ACM Transactions on Computer Systems* 19.3 (Aug. 2001), pp. 332–383.

[2] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. "SCRIBE: The design of a large-scale event notification infrastructure." In: *In Networked Group Communication*. 2001, pp. 30–43.

[3] P. R. Pietzuch and J. M. Bacon. "Hermes: a distributed event-based middleware architecture." In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. 2002, pp. 611–618. DOI: 10.1109/ICDCSW.2002.1030837.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe." In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.

[5] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. "Efficient Publish/Subscribe Through a Self-Organizing Broker Overlay and Its Application to SIENA." In: *Comput. J.* 50.4 (July 2007), pp. 444–459. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxm002.

[6] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovskii. "The PADRES Distributed Publish/Subscribe System." In: *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05)* (July 2005), pp. 12–30.

[7] A.-M. Kermarrec and P. Triantafillou. "XL Peer-to-peer Pub/Sub Systems." In: *ACM Comput. Surv.* 46.2 (Nov. 2013), 16:1–16:45. ISSN: 0360-0300. DOI: 10.1145/2543581.2543583.

[8] H.-A. Jacobsen, A. Cheung, G. Li, et al. "The PADRES Publish/Subscribe System." In: *Principles and Applications of Distributed Event-Based Systems*. IGI Global, 2010, pp. 164–205.

[9] F. Rahimian, S. Girdzijauskas, A. H. Payberah, and S. Haridi. "Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks." In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 746–757. DOI: 10.1109/IPDPS.2011.75.

[10] M.-M. Wang, J.-N. Cao, J. Li, and S. K. Dasi. "Middleware for Wireless Sensor Networks: A Survey." In: *Journal of Computer Science and Technology* 23.3 (2008), pp. 305–326. ISSN: 1860-4749. DOI: 10.1007/s11390-008-9135-x.

[11] K. McLaughlin. *Gaps in 4G network hinder high-tech agriculture*. Accessed May 1, 2018. July 2016. URL: http://www.bendbulletin.com/home/4535283-151/gaps-in-4g-network-hinder-high-tech-agriculture.

[12] V. Setty, G. Kreitz, R. Vitenberg, et al. "The Hidden Pub/Sub of Spotify: (Industry Article)." In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 231–240. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488273.

[13] *Facebook Messenger*. Accessed May 1, 2018. Aug. 2012. URL: https://www.facebook.com/notes/10150259350998920.

[14] *Eurostat*. Accessed Nov 1, 2018. Nov. 2018. URL: https://ec.europa.eu/eurostat.

[15] *Pakistan Telecommumincation Authority*. Accessed Nov 1, 2018. Nov. 2018. URL: https://www.pta.gov.pk/en/telecom-indicators.

[16] C. on Energy and C. o. t. U. S. Commerce. *Letter to Niantic regarding Pokemon Go data usage*. Accessed May 1, 2018. July 2016. URL: https://democrats-energycommerce.house.gov/sites/democrats.energycommerce.house.gov/files/JohnHanke.Niantic.%20Pokemon%20Go%20Letter.2016.07.19.pdf.

[17] D. Vasisht, Z. Kapetanovic, J. Won, et al. "FarmBeats: An IoT Platform for Data-Driven Agriculture." In: *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 2017, pp. 515–529.

[18] *RFC1951 - DEFLATE Compressed Data Format Specification version 1.3*. Accessed May 1, 2018. May 1996. URL: https://tools.ietf.org/html/rfc1951.

[19] C. Doblander, T. Ghinaiya, K. Zhang, and H.-A. Jacobsen. "Shared Dictionary Compression in Publish/Subscribe Systems." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 117–124. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933308.

[20] C. Doblander, A. Khatayee, and H.-A. Jacobsen. "PreDict: Predictive Dictionary Maintenance for Message Compression in Publish/Subscribe." In: Middleware '18. Rennes, France: ACM, 2018. ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274822.

[21] C. Doblander, K. Zhang, and H. A. Jacobsen. "Publish/Subscribe for Mobile Applications Using Shared Dictionary Compression." In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. June 2016, pp. 775–776. DOI: 10.1109/ICDCS.2016.70.

[22] C. Doblander, S. Zimmermann, K. Zhang, and H.-A. Jacobsen. "Demo Abstract: MOS: A Bandwidth-Efficient Cross-Platform Middleware for Publish/Subscribe." In: *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference*. Middleware Posters and Demos '16. Trento, Italy: ACM, 2016, pp. 27–28. ISBN: 978-1-4503-4666-5. DOI: 10.1145/3007592.3007607.

[23] Y. Sharma, P. Ajoux, P. Ang, et al. "Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services." In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 351–366. ISBN: 978-1-931971-218.

[24] R. Baldoni, R. Beraldi, V. Quema, et al. "TERA: Topic-based Event Routing for Peer-to-peer Architectures." In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada:

ACM, 2007, pp. 2–13. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266898.

[25] M. Kapritsos and P. Triantafillou. "BAD: Bandwidth Adaptive Dissemination or (the Case for BAD Trees)." In: *Proceedings of the 2007 ACM/IFIP/USENIX International Conference on Middleware Companion*. MC '07. Newport Beach, California: ACM, 2007, 25:1–25:6. ISBN: 978-1-59593-935-7. DOI: 10.1145/1377943.1377963.

[26] S. Ji, C. Ye, J. Wei, and H. Jacobsen. "Towards Scalable Publish/Subscribe Systems." In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. June 2015, pp. 784–785. DOI: 10.1109/ICDCS.2015.108.

[27] D. Dedousis, N. Zacheilas, and V. Kalogeraki. "On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. July 2018, pp. 76–86. DOI: 10.1109/ICDCS.2018.00018.

[28] I. Aekaterinidis and P. Triantafillou. "Pyracanthus: A Scalable Solution for DHT-independent Content-based Publish/Subscribe Data Networks." In: *Inf. Syst.* 36.3 (May 2011), pp. 655–674. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.11.002.

[29] *RFC1952 - GZIP file format specification version 4.3*. Accessed May 1, 2018. Apr. 2018. URL: https://www.ietf.org/rfc/rfc1952.txt.

[30] H. White. "Printed english compression by dictionary encoding." In: *Proceedings of the IEEE* 55.3 (Mar. 1967), pp. 390–396. ISSN: 0018-9219. DOI: 10.1109/PROC.1967.5496.

[31] J. Bentley and D. McIlroy. "Data compression using long common strings." In: *Data Compression Conference, 1999. Proceedings. DCC '99*. Mar. 1999, pp. 287–295. DOI: 10.1109/DCC.1999.755678.

[32] J. Bentley and D. McIlroy. "Data compression with long repeated strings." In: *Information Sciences* 135.1–2 (2001). Dictionary Based Compression, pp. 1–11. ISSN: 0020-0255. DOI: http://dx.doi.org/10.1016/S0020-0255(01)00097-4.

[33] *FemtoZip compression library*. Accessed May 1, 2018. Apr. 2018. URL: https://github.com/chrido/femtozip.

[34] E. Skjervold and M. Skjegstad. "REAP: Delta Compression for Publish/Subscribe Web Services in MANETs." In: *Military Communications Conference, MILCOM 2013 - 2013 IEEE*. Nov. 2013, pp. 1488–1496. DOI: 10.1109/MILCOM.2013.251.

[35] L. Xu, A. Pavlo, S. Sengupta, et al. "Reducing Replication Bandwidth for Distributed Document Databases." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 222–235. ISBN: 978-1-4503-3651-2. DOI: 10.1145/2806777.2806840.

[36] P. Alves and P. Ferreira. "Radiator - efficient message propagation in context-aware systems." In: *Journal of Internet Services and Applications* 5.1 (Apr. 2014), p. 4. ISSN: 1869-0238. DOI: 10.1186/1869-0238-5-4.

[37] S. Ji, C. Ye, J. Wei, and H.-A. Jacobsen. "MERC: Match at Edge and Route intra–Cluster for Content-based Publish/Subscribe Systems." In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 13–24. ISBN: 978-1-4503-3618-5. DOI: 10.1145/2814576.2814801.

[38] J. J. Hunt, K.-P. Vo, and W. F. Tichy. "Delta Algorithms: An Empirical Analysis." In: *ACM Trans. Softw. Eng. Methodol.* 7.2 (Apr. 1998), pp. 192–214. ISSN: 1049-331X. DOI: 10.1145/279310.279321.

[39] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter. *A Proposal for Shared Dictionary Compression over HTTP*. https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf. 2008.

[40] *Shared Dictionary Compression for HTTP at LinkedIn*. Apr. 2017. URL: https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin.

[41] J. Alakuijala and Z. Szabadka. *RFC 7932: Brotli Compressed Data Format*. https://tools.ietf.org/html/rfc7932. 2015.

[42] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. "The Implementation and Performance of Compressed Databases." In: *SIGMOD Rec.* 29.3 (Sept. 2000), pp. 55–67. ISSN: 0163-5808. DOI: 10.1145/362084.362137.

[43] *Percona Server for MySQL, Compressed Columns with Dictionaries*. Accessed May 1, 2018. May 2018. URL: %7Bhttps://www.percona.com/doc/percona-server/5.7/flexibility/compressed_columns.html%7D.

[44] I. Müller, C. Ratsch, and F. Färber. "Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems." In: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. 2014, pp. 283–294. DOI: 10.5441/002/edbt.2014.27.

[45] *Apache HBase*. Accessed May 1, 2018. May 2018. URL: https://hbase.apache.org/.

[46] F. Chang, J. Dean, S. Ghemawat, et al. "Bigtable: A Distributed Storage System for Structured Data." In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. OSDI '06. Seattle, WA: USENIX Association, 2006, pp. 15–15.

[47] C. Binnig, S. Hildenbrand, and F. Färber. "Dictionary-based Order-preserving String Compression for Main Memory Column Stores." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 283–296. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559877.

[48] Z. Jerzak and H. Ziekow. "The DEBS 2015 Grand Challenge." In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. Oslo, Norway: ACM, 2015, pp. 266–268. ISBN: 978-1-4503-3286-6. DOI: 10.1145/2675743.2772598.

[49] D. Karger, E. Lehman, T. Leighton, et al. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660.

[50] *MQTT*. http://mqtt.org/.

[51] *Google Protocol Buffers*. https://developers.google.com/protocol-buffers/. Accessed May 1, 2018. Apr. 2018.

[52] *NYC OpenData*. https://data.cityofnewyork.us/.

[53] *Moquette*. https://github.com/andsel/moquette. Apr. 2017.

[54] P. Salehi, K. Zhang, and H. Jacobsen. "Incremental Topology Transformation for Publish/Subscribe Systems Using Integer Programming." In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. June 2017, pp. 80–91. DOI: 10.1109/ICDCS.2017.17.

[55] M. Rotaru, F. Olariu, E. Onica, and E. Rivière. "Reliable Messaging to Millions of Users with Migratorydata." In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. Middleware '17. Las Vegas, Nevada: ACM, 2017, pp. 1–7. ISBN: 978-1-4503-5200-0. DOI: 10.1145/3154448.3154449.

[56] P. Salehi, C. Doblander, and H.-A. Jacobsen. "Highly-available Content-based Publish/Subscribe via Gossiping." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 93–104. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933303.

[57] A. Pal and K. Kant. "IoT-Based Sensing and Communications Infrastructure for the Fresh Food Supply Chain." In: *Computer* 51.2 (Feb. 2018), pp. 76–80. ISSN: 0018-9162. DOI: 10.1109/MC.2018.1451665.

[58] A. Zanella, N. Bui, A. Castellani, et al. "Internet of Things for Smart Cities." In: *IEEE Internet of Things Journal* 1.1 (Feb. 2014), pp. 22–32. ISSN: 2327-4662. DOI: 10.1109/JIOT.2014.2306328.

[59] *Java port of Google's open-vcdiff vcdiff (RFC3284) implementation*. Accessed Nov 1, 2018. Nov. 2018. URL: https://github.com/ehrmann/vcdiff-java.

[60] *RFC 3284: VCDIFF*. Accessed Nov 1, 2018. Nov. 2018. URL: https://tools.ietf.org/html/rfc3284.

[61] E. W. Myers. "An O(ND) Difference Algorithm and Its Variations." In: *Algorithmica* 1 (1986), pp. 251–266.

[62] *Apache Kafka*. Apr. 2017. URL: https://kafka.apache.org/.

[63] *Pulsar*. Accessed Nov 1, 2018. Nov. 2018. URL: http://pulsar.apache.org/.

[64] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron. "Scribe: a large-scale and decentralized application-level multicast infrastructure." In: *IEEE Journal on Selected Areas in Communications* 20.8 (Oct. 2002), pp. 1489–1499. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.803069.

[65] I. Aekaterinidis and P. Triantafillou. "PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network." In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. July 2006, pp. 23–23. DOI: 10.1109/ICDCS.2006.63.

[66] L. Lamport. "The Part-time Parliament." In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[67] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[68] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851.

[69] *Python*. Accessed Nov 1, 2018. Nov. 2018. URL: https://www.python.org/.

[70] L. Chen, H. Zhang, J. M. Jose, et al. "Topic detection and tracking on heterogeneous information." In: *Journal of Intelligent Information Systems* 51.1 (Aug. 2018), pp. 115–137. ISSN: 1573-7675. DOI: 10.1007/s10844-017-0487-y.

[71] *ZStandard*. Nov. 2018. URL: https://github.com/facebook/zstd.

[72] G. Brockman, V. Cheung, L. Pettersson, et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

Appendices

Appendix A

Datasets

A.1 DEBS 2015 Taxitrip datasets

Listing A.1.1: DEBS 2015 dataset in csv

```
07290D3599E7A0D62097A346EFCC1FB5,E7750A37CAB07D0DFF0AF7E3573AC141,2013-01-01 00:00:00,2013-01-01
    00:02:00,120,0.44,-73.956528,40.716976,-73.962440,40.715008,CSH,3.50,0.50,0.50,0.00,0.00,4.50

22D70BF00EEB0ADC83BA8177BB861991,3FF2709163DE7036FCAA4E5A3324E4BF,2013-01-01 00:02:00,2013-01-01
    00:02:00,0,0.00,0.000000,0.000000,0.000000,0.000000,CSH,27.00,0.00,0.50,0.00,0.00,27.50

0EC22AAF491A8BD91F279350C2B010FD,778C92B26AE78A9EBDF96B49C67E4007,2013-01-01 00:01:00,2013-01-01
    00:03:00,120,0.71,-73.973145,40.752827,-73.965897,40.760445,CSH,4.00,0.50,0.50,0.00,0.00,5.00

1390FB380189DF6BBFDA4DC847CAD14F,BE317B986700F63C43438482792C8654,2013-01-01 00:01:00,2013-01-01
    00:03:00,120,0.48,-74.004173,40.720947,-74.003838,40.726189,CSH,4.00,0.50,0.50,0.00,0.00,5.00
```

Listing A.1.2: DEBS 2015 dataset in json

```
{"tip_amount": "0.00", "payment_type": "CSH", "dropoff_latitude": "40.715008", "dropoff_datetime": "2013-01-01
    00:02:00", "medallion": "07290D3599E7A0D62097A346EFCC1FB5", "pickup_datetime": "2013-01-01 00:00:00", "mta_tax":
     "0.50", "total_amount": "4.50", "tolls_amount": "0.00", "surcharge": "0.50", "hack_license": "
    E7750A37CAB07D0DFF0AF7E3573AC141", "fare_amount": "3.50", "pickup_longitude": "-73.956528", "pickup_latitude":
    "40.716976", "trip_time_in_secs": "120","dropoff_longitude": "-73.962440", "trip_distance": "0.44"}

{"tip_amount": "0.00", "payment_type": "CSH", "dropoff_latitude": "0.000000", "dropoff_datetime": "2013-01-01
    00:02:00", "medallion": "22D70BF00EEB0ADC83BA8177BB861991", "pickup_datetime": "2013-01-01 00:02:00", "mta_tax":
     "0.50", "total_amount": "27.50", "tolls_amount": "0.00", "surcharge": "0.00", "hack_license": "3
    FF2709163DE7036FCAA4E5A3324E4BF", "fare_amount": "27.00", "pickup_longitude": "0.000000", "pickup_latitude":
    "0.000000", "trip_time_in_secs": "0", "dropoff_longitude": "0.000000", "trip_distance": "0.00"}
```

Listing A.1.3: DEBS 2015 dataset in xml

```
<trip>
  <tip_amount>0.00</tip_amount>
  <payment_type>CSH</payment_type>
  <dropoff_latitude>40.715008</dropoff_latitude>
  <dropoff_datetime>2013-01-01 00:02:00</dropoff_datetime>
  <medallion>07290D3599E7A0D62097A346EFCC1FB5</medallion>
  <pickup_datetime>2013-01-01 00:00:00</pickup_datetime>
  <mta_tax>0.50</mta_tax>
  <total_amount>4.50</total_amount>
  <tolls_amount>0.00</tolls_amount>
  <surcharge>0.50</surcharge>
  <hack_license>E7750A37CAB07D0DFF0AF7E3573AC141</hack_license>
  <fare_amount>3.50</fare_amount>
  <pickup_longitude>-73.956528</pickup_longitude>
  <pickup_latitude>40.716976</pickup_latitude>
  <trip_time_in_secs>120</trip_time_in_secs>
  <dropoff_longitude>-73.962440</dropoff_longitude>
  <trip_distance>0.44</trip_distance>
</trip>
```

Listing A.1.4: DEBS 2015 dataset - Protobuf message format

```
option java_package = "de.tum.i13.pb";
option java_outer_classname = "Debs2015Protos";

message Taxitrip {
  required string medallion = 1;
  required string hack_license = 2;
  required string pickup_datetime       = 3;
  required string dropoff_datetime = 4;
  required uint32 trip_time_in_secs = 5;
  required float trip_distance = 6;
  required float pickup_longitude = 7;
  required float pickup_latitude = 8;
  required float dropoff_longitude = 9;
  required float dropoff_latitude = 10;
  required string payment_type = 11;
  required float fare_amount = 12;
  required float surcharge = 13;
  required float mta_tax = 14;
  required float tip_amount = 15;
  required float tolls_amount = 16;
  required float total_amount = 17;
}
```

A.2 Social media datasets

Listing A.2.1: Twitter US json dataset

{"created_at":"Thu Dec 08 15:17:16 +0000 2016","id":806880331445964806, "id_str":"806880331445964806", "text":"I've told my mom numerous times that I have a sleeping problem yet everytime I'm up all night she yells at me and asks why I'm not asleep??", "source":"u003ca href="http://twitter.com/download/iphone" rel="nofollow"Twitter for iPhone/a", "truncated":false, "in_reply_to_status_id":null, "in_reply_to_status_id_str":null, "in_reply_to_user_id":null, "in_reply_to_user_id_str":null, "in_reply_to_screen_name":null, "user":{"id":512655145, "id_str":"512655145", "name":"", "screen_name":"ColeFigley3", "location":"O-H", "url":"http://Instagram.com/cole_fig3", "description":"\#AWOLSC: cfigley3", "protected":false, "verified":false, "followers_count":941, "friends_count":370, "listed_count":3, "favourites_count":22074, "statuses_count":24013, "created_at":"Sat Mar 03 01:36:10 +0000 2012", "utc_offset":-28800, "time_zone":"Pacific Time (US \& Canada)", "geo_enabled":true, "lang":"en", "contributors_enabled":false, "is_translator":false, "profile_background_color":"000000", "profile_background_image_url":"http://abs.twimg.com/images/themes/theme1/bg.png", "profile_background_image_url_https":"https://abs.twimg.com/images/themes/theme1/bg.png", "profile_background_tile":false, "profile_link_color":"DD2E44", "profile_sidebar_border_color":"000000", "profile_sidebar_fill_color":"000000", "profile_text_color":"000000", "profile_use_background_image":false, "profile_image_url":"http://pbs.twimg.com/profile_images/804938214670761984/lfrvedZ1_normal.jpg", "profile_image_url_https":"https://pbs.twimg.com/profile_images/804938214670761984/lfrvedZ1_normal.jpg","profile_banner_url": "https://pbs.twimg.com/profile_banners/512655145/1481062764", "default_profile":false, "default_profile_image":false, "following":null, "follow_request_sent":null, "notifications":null}, "geo":null, "coordinates":null, "place":{"id":"de599025180e2ee7", "url":"https://api.twitter.com/1.1/geo/id/de599025180e2ee7.json", "place_type":"admin", "name":"Ohio", "full_name":"Ohio, USA", "country_code":"US", "country":"United States", "bounding_box":{"type":"Polygon", "coordinates":[[[-84.820309,38.403186], [-84.820309,42.327133], [-80.518626,42.327133], [-80.518626,38.403186]]]}, "attributes":{}}, "contributors":null, "is_quote_status":false, "retweet_count":0, "favorite_count":0, "entities":{"hashtags":[], "urls":[], "user_mentions":[], "symbols":[]}, "favorited":false, "retweeted":false, "filter_level":"low", "lang":"en", "timestamp_ms":"1481210236785"}

Listing A.2.2: Meetup comments dataset

{"visibility":"public", "member":{"member_id":122075502, "photo":"http://photos1.meetupstatic.com/photos/member/8/8/c/d/thumb_247295021.jpeg", "member_name":"Pairic"}, "comment":" Booked Seat H15. ", "id":472395031, "mtime":1478608243000, "event":{"event_name":"Watch \"Nocturnal Animals\" Cineworld", "event_id":"235412299"},"group":{"join_mode":"open", "country":"ie", "city":"Dublin", "name":"Dublin Horror Society", "group_lon":-6.25, "id":13332502, "urlname":"Dublin-Horror-Society", "category":{"name":"movies/film", "id":20, "shortname":"movies-film"}, "group_photo":{"highres_link":"http://photos4.meetupstatic.com/photos/event/9/5/8/0/highres_343478272.jpeg", "photo_link": "http://photos2.meetupstatic.com/photos/event/9/5/8/0/600_343478272.jpeg", "photo_id":343478272, "thumb_link":"http://photos4.meetupstatic.com/photos/event/9/5/8/0/thumb_343478272.jpeg"}, "group_lat":53.33}, "status":"active"}

Listing A.2.3: Meetup events dataset

{"utc_offset":39600000, "venue":{"country":"AU", "city":"Erskineville", "address_1":"Sydney Park Rd (Opp. Mitchell Rd)", "name":"Sydney Park", "lon":151.18959,"lat":-33.90789}, "rsvp_limit":0,"venue_visibility":"public", "visibility":"public", "maybe_rsvp_count":0, "description":" Join, connect, socialise, and network with other creatives from across Sydney at a gorgeous, hidden, harborside park. Arts, music, fashion, film, communication, performing arts, design and more - all creatives are welcome. Join the Facebook event for details:
https://www.facebook.com/events/105466003266724/ Plus, attendees will go into the running to win a \$1,000 exposure package. Eat, drink, and mingle with Sydney's incredible creative community. This is a free social gathering - attend by yourself or with fellow creatives. ","mtime":1478608286592, "event_url":"http://www.meetup.com/Sydney-Creatives/events/235290638/", "yes_rsvp_count":29, "payment_required":"0", "name":"Sydney Creatives Picnic", "id":"235290638", "time":1484964000000, "group":{"join_mode":"open", "country":"au", "city":"Sydney", "name":"Sydney Creatives", "group_lon":151.21, "id":2782292, "urlname":"Sydney-Creatives", "category":{"name":"career/business", "id":2, "shortname":"career-business"}, "group_photo":{"highres_link":"http://photos2.meetupstatic.com/photos/event/b/e/4/c/highres_449148716.jpeg", "photo_link":"http://photos2.meetupstatic.com/photos/event/b/e/4/c/600_449148716.jpeg", "photo_id":449148716, "thumb_link":"http://photos4.meetupstatic.com/photos/event/b/e/4/c/thumb_449148716.jpeg"},"group_lat":-33.87}, "status":"upcoming"}

Listing A.2.4: Meetup RSVPs dataset

{"venue": {"venue_name":"Baltenswilerstrasse 5", "lon":8.628714, "lat":47.443462, "venue_id":24848436}, "visibility":"public", "response":"no", "guests":0, "member":{"member_id":195090863, "photo":"http://photos4.meetupstatic.com/photos/member/3/2/7/4/thumb_251112916.jpeg", "member_name":"Nadia" }, "rsvp_id":1638527340, "mtime":1478608271889, "event":{"event_name":"Show up to meet people", "event_id":"234891743", "time":1479135600000, "event_url":"http://www.meetup.com/Bassersdorf-Language-Culture-Meetup/events/234891743/"}, "group":{"group_topics": [{"urlkey":"esl", "topic_name":"English as a Second Language"}, {"urlkey":"esl-practice", "topic_name":"ESL practice"}, {"urlkey":"language-exchange", "topic_name":"Language Exchange"}, {"urlkey":"english-language", "topic_name":"English Language"}, {"urlkey":"culture", "topic_name":"Culture"}, {"urlkey":"motivation", "topic_name":"Motivation"}, {"urlkey":"conversation", "topic_name":"Conversation"}, {"urlkey":"socialnetwork", "topic_name":"Social Networking"}, {"urlkey":"culture-exchange", "topic_name":"Culture Exchange"}, {"urlkey":"language", "topic_name":"Language \& Culture"}, {"urlkey":"learning", "topic_name":"Learning"}, {"urlkey":"courses-and-workshops", "topic_name":"Courses and Workshops"}], "group_city":"Bassersdorf", "group_country":"ch", "group_id":20752133, "group_name":"Bassersdorf Language and Culture Meetup", "group_lon":8.61, "group_urlname":"Bassersdorf-Language-Culture-Meetup", "group_lat":47.45}}

A.3 Financial datasets

Listing A.3.1: EPEX Spot market

"2012-01-01 10:45:00+01";"2012-01-01 11:00:00+01";"15min";"germany/austria";;;;;;;
"2012-01-01 11:00:00+01";"2012-01-01 12:00:00+01";"hourly";"france";10;15;10;13.01;13.01;86;86
"2012-01-01 11:00:00+01";"2012-01-01 12:00:00+01";"hourly";"germany/austria";5;25;5;15.24;15.23;1174.3;1174.3
"2012-01-01 11:00:00+01";"2012-01-01 11:15:00+01";"15min";"germany/austria";;;;;;;

Listing A.3.2: GDAX Cryptocurrency exchange

{"type": "open", "price": "709.65000000", "remaining_size": "0.01000011", "sequence": 1707596156, "side": "buy", "order_id": "71a7db5f-9b02-49d8-8c61-4321134985ec", "product_id": "BTC-USD", "time": "2016-11-15T10:21:57.978000Z"}
{"type": "done", "reason": "canceled", "price": "709.70000000", "sequence": 1707596157, "side": "buy", "order_id": "0873b924-8d18-422b-ae14-a23ce4e668e7", "remaining_size": "0.01000321", "product_id": "BTC-USD", "time": "2016-11-15T10:21:58.270000Z"}
{"size": "0.01000321", "type": "received", "time": "2016-11-15T10:21:58.350000Z", "price": "709.65000000", "order_type": "limit", "side": "buy", "order_id": "707873fc-e7eb-453f-804f-bd1818495f41", "product_id": "BTC-USD", "sequence": 1707596158}
{"type": "open", "price": "709.65000000", "remaining_size": "0.01000321", "sequence": 1707596159, "side": "buy", "order_id": "707873fc-e7eb-453f-804f-bd1818495f41", "product_id": "BTC-USD", "time": "2016-11-15T10:21:58.350000Z"}
{"type": "done", "reason": "canceled", "price": "709.65000000", "sequence": 1707596160, "side": "buy", "order_id": "71a7db5f-9b02-49d8-8c61-4321134985ec", "remaining_size": "0.01000011", "product_id": "BTC-USD", "time": "2016-11-15T10:21:59.281000Z"}

A.4 Other datasets

Listing A.4.1: Openstack Neutron Service

```
2016-11-14 06:25:11.949 89152 DEBUG oslo_messaging._drivers.amqpdriver [req-e6208ddc-913c-4075-bee4-aebd3c9bad51 - - - - -] sending reply msg_id: 4dc6083445f544a183cc8795172a6fa5 reply queue: reply_f128945ec2174505982655a88ceec960 time elapsed: 0.0424906117842s _send_reply /usr/lib/python2.7/dist-packages/oslo_messaging/_drivers/amqpdriver.py:74

2016-11-14 06:25:12.040 89128 INFO neutron.wsgi [req-3e6dbc63-7e6f-4006-84fc-7f2a90b5ff66 e2fec82efdf34f4283be778bca34c2c2 6666747c8d4b417990106639e4b54c11 - - -] 172.24.18.119 - - [14/Nov/2016 06:25:12] "GET /v2.0/ports.json?tenant_id=9c6d7868d17f4cfbbb3402755bcc0694 \&device_id=7f42e441-42f0-4a23-a3eb-857aba3abc67 HTTP/1.1" 200 1225 0.021415

2016-11-14 06:25:12.084 89128 INFO neutron.wsgi [req-66658959-dfbf-421c-9dea-bab14f3108b2 e2fec82efdf34f4283be778bca34c2c2 6666747c8d4b417990106639e4b54c11 - - -] 172.24.18.119 - - [14/Nov/2016 06:25:12] "GET /v2.0/networks.json?id=dd0e99f0-4112-458f-a30f-328b517ed627 HTTP/1.1" 200 903 0.040351

2016-11-14 06:25:12.097 89128 INFO neutron.wsgi [req-54fa4d30-7585-4be2-8b7a-b7a835acc981 e2fec82efdf34f4283be778bca34c2c2 6666747c8d4b417990106639e4b54c11 - - -] 172.24.18.119 - - [14/Nov/2016 06:25:12] "GET /v2.0/floatingips.json?fixed_ip_address=172.24.45.194 \&port_id=1f51ddf9-448d-469d-84d1-01628131c6bc HTTP/1.1" 200 232 0.008559

2016-11-14 06:25:12.138 89128 INFO neutron.wsgi [req-444be002-f188-46e1-8f55-0b054008c7fa e2fec82efdf34f4283be778bca34c2c2 6666747c8d4b417990106639e4b54c11 - - -] 172.24.18.119 - - [14/Nov/2016 06:25:12] "GET /v2.0/subnets.json?id=0bfb3997-0678-4ddb-9f03-20e9270f43cf HTTP/1.1" 200 994 0.038401
```

Listing A.4.2: TPCH lineitems

```
1|155190|7706|1|17|21168.23|0.04|0.02|N|O|1996-03-13|1996-02-12|1996-03-22|DELIVER IN PERSON|TRUCK|egular courts above the|

1|67310|7311|2|36|45983.16|0.09|0.06|N|O|1996-04-12|1996-02-28|1996-04-20|TAKE BACK RETURN|MAIL|ly final dependencies: slyly bold |

1|63700|3701|3|8|13309.60|0.10|0.02|N|O|1996-01-29|1996-03-05|1996-01-31|TAKE BACK RETURN|REG AIR|riously. regular, express dep|

1|2132|4633|4|28|28955.64|0.09|0.06|N|O|1996-04-21|1996-03-30|1996-05-16|NONE|AIR|lites. fluffily even de|

1|24027|1534|5|24|22824.48|0.10|0.04|N|O|1996-03-30|1996-03-14|1996-04-01|NONE|FOB| pending foxes. slyly re|
```

Appendix B

PreDict at different batchsizes

B.1 Additional results for PreDict at different batch-sizes

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.19	0.23	0.20	0.14	0.35	0.70	0.14	0.23	0.17	0.27	0.14	0.82	0.45	1.16	0.53	0.15	0.12	0.13	0.42	1.64	0.4/0.2/0.4
VCDIFF	0.26	0.42	0.27	0.30	0.36	1.84	0.20	0.38	0.21	0.36	0.19	1.94	0.96	2.57	1.38	0.19	0.24	0.18	0.70	2.48	0.8/0.4/0.8
FIXEDRATE*	0.36	0.51	0.27	0.20	1.29	3.21	0.25	0.58	0.21	0.49	0.24	6.25	1.12	11.27	1.95	0.25	0.19	0.15	0.91	3.69	1.7/0.5/2.7
SINGLEDICT*	1.31	1.09	0.85	0.49	7.24	7.03	0.67	1.65	0.37	1.45	0.59	9.90	3.87	22.84	9.62	0.35	0.36	0.46	3.26	20.74	4.7/1.4/6.5
ADAPTIVE*	0.50	0.69	0.48	0.32	3.20	7.27	0.39	1.01	0.25	0.69	0.35	8.85	2.11	8.49	6.65	0.34	0.28	0.21	1.85	8.28	2.6/0.7/3.2
PREDICT	0.48	0.86	0.39	0.27	1.58	2.41	0.42	0.74	0.37	0.52	0.38	4.16	1.77	7.62	4.15	0.32	0.30	0.25	0.86	4.79	1.6/0.6/2.0
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.06	0.07	0.06	0.04	0.08	0.17	0.05	0.06	0.05	0.08	0.05	0.23	0.11	0.23	0.14	0.05	0.04	0.04	0.12	0.25	0.1/0.1/0.1
VCDIFF	0.16	0.17	0.15	0.15	0.43	0.57	0.06	0.19	0.07	0.25	0.10	0.97	0.36	0.87	0.46	0.10	0.09	0.04	0.38	1.37	0.3/0.2/0.3
FIXEDRATE*	0.11	0.20	0.09	0.07	0.17	1.02	0.10	0.15	0.11	0.17	0.09	0.68	0.45	1.21	0.55	0.10	0.08	0.07	0.32	1.17	0.3/0.2/0.4
SINGLEDICT*	0.14	0.13	0.10	0.07	0.32	1.30	0.12	0.17	0.14	0.17	0.07	0.87	0.58	1.54	0.75	0.10	0.08	0.21	0.43	1.27	0.4/0.2/0.5
ADAPTIVE*	0.12	0.18	0.10	0.07	0.27	1.41	0.12	0.15	0.14	0.16	0.09	0.91	0.49	1.64	0.77	0.12	0.09	0.16	0.43	1.34	0.4/0.2/0.5
PREDICT	0.11	0.17	0.10	0.07	0.18	0.98	0.12	0.15	0.15	0.23	0.08	0.78	0.45	1.44	0.57	0.12	0.09	0.09	0.24	0.99	0.4/0.2/0.4
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.17	0.07	0.10	0.04	109.68	73.13	0.07	0.15	0.07	0.22	0.21	1597.68	0.34	3347.02	21.24	25.58	0.03	0.03	212.41	4.00	269.6/0.2/786.3
SINGLEDICT*	4.98	3825.68	4.77	1.60	37.94	3.23	0.36	2.08	0.21	3.56	0.77	177.06	10.81	95.09	27.73	0.79	0.80	0.09	1359.35	79.33	281.8/4.2/864.4
ADAPTIVE*	0.54	0.37	1.57	0.41	481.15	4.22	0.09	1.17	0.06	0.55	0.23	72.43	8.34	70.42	63.62	0.76	0.11	0.04	731.14	16.99	72.7/1.0/183.7
PREDICT	0.02	0.02	0.02	0.01	0.27	0.09	0.01	0.02	0.01	0.02	0.01	1.40	0.10	0.86	0.34	0.01	0.01	0.00	18.22	0.65	1.1/0.0/3.9
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	77.4	69.5	77.2	77.7	88.3	58.1	56.4	74.9	42.2	79.8	70.7	82.3	67.1	65.5	70.9	61.4	63.6	43.1	78.2	78.9	69.2/69.2/69.2
VCDIFF	61.9	43.1	60.3	61.1	89.1	24.6	5.3	56.0	6.1	75.0	53.6	64.3	35.4	34.4	30.1	52.5	20.1	0.0	60.9	62.9	44.8/44.8/44.8
FIXEDRATE*	84.9	76.1	84.6	85.4	93.7	63.2	61.3	81.7	52.6	86.0	79.7	85.6	70.8	70.0	74.1	75.7	72.1	52.5	82.0	82.2	75.7/-35.8/-1040.3
SINGLEDICT*	87.9	83.0	87.7	89.4	93.5	62.4	61.9	83.5	52.8	87.2	82.9	85.3	74.1	69.5	77.2	74.0	76.7	62.7	83.6	83.8	78.0/70.5/3.5
ADAPTIVE*	88.6	82.4	88.8	90.4	94.3	63.6	62.6	84.9	53.6	88.6	83.8	86.8	74.1	69.2	77.8	77.6	78.0	59.7	84.3	84.8	78.7/71.8/9.3
PREDICT	88.5	83.0	88.5	90.3	92.7	60.2	62.6	84.4	53.3	81.5	82.9	82.3	73.8	69.6	77.0	78.0	77.4	60.3	84.0	84.3	77.7/74.6/48.7

* Best of all permutations in terms of bandwidth reduction

Table B.1.1: Result summary (including all overhead) for batchsize 5

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.17	0.20	0.17	0.10	0.31	0.71	0.10	0.18	0.15	0.20	0.12	0.95	0.39	1.24	0.59	0.12	0.10	0.11	0.36	1.25	0.4/0.2/0.4
VCDIFF	0.26	0.45	0.29	0.22	0.29	1.90	0.27	0.41	0.25	0.25	0.19	1.53	0.81	2.58	1.46	0.23	0.27	0.21	0.68	2.44	0.7/0.3/0.8
FIXEDRATE*	0.38	0.50	0.30	0.19	1.49	3.65	0.24	0.52	0.19	0.46	0.20	7.60	1.10	11.99	2.00	0.20	0.17	0.17	0.98	4.63	1.8/0.5/3.0
SINGLEDICT*	1.17	1.06	0.90	0.53	7.64	7.59	0.64	2.42	0.47	1.28	0.73	9.27	4.12	25.84	9.58	0.30	0.38	0.27	4.02	17.65	4.8/1.2/6.6
ADAPTIVE*	0.63	0.84	0.40	0.31	2.94	6.36	0.41	0.75	0.28	0.59	0.32	12.56	2.15	13.61	5.06	0.31	0.25	0.31	2.32	9.43	3.0/0.7/4.1
PREDICT	0.43	0.89	0.41	0.29	1.62	2.70	0.34	0.85	0.33	0.60	0.37	4.88	1.72	7.88	4.60	0.28	0.32	0.29	0.88	4.47	1.7/0.7/2.1
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.04	0.05	0.04	0.03	0.07	0.14	0.04	0.05	0.04	0.06	0.04	0.40	0.10	0.24	0.14	0.04	0.03	0.04	0.08	0.28	0.1/0.0/0.1
VCDIFF	0.17	0.18	0.16	0.13	0.37	0.52	0.08	0.21	0.07	0.21	0.10	0.84	0.31	0.87	0.48	0.12	0.08	0.04	0.39	1.25	0.3/0.2/0.3
FIXEDRATE*	0.11	0.15	0.09	0.06	0.16	0.84	0.10	0.13	0.12	0.13	0.06	0.68	0.45	1.25	0.51	0.09	0.07	0.09	0.37	0.98	0.3/0.1/0.3
SINGLEDICT*	0.15	0.12	0.10	0.08	0.23	1.21	0.11	0.23	0.17	0.19	0.09	0.73	0.52	1.54	0.67	0.11	0.08	0.09	0.49	1.03	0.4/0.2/0.4
ADAPTIVE*	0.12	0.18	0.09	0.07	0.25	1.25	0.12	0.16	0.15	0.15	0.08	0.88	0.51	1.54	0.68	0.11	0.08	0.09	0.46	1.22	0.4/0.2/0.4
PREDICT	0.14	0.18	0.10	0.09	0.21	0.96	0.10	0.15	0.14	0.20	0.07	0.79	0.45	1.39	0.62	0.10	0.09	0.09	0.23	1.18	0.4/0.2/0.4
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.17	0.07	0.06	0.03	95.40	78.74	0.04	0.07	0.01	0.10	0.15	2526.80	1.60	3100.81	23.88	14.05	0.02	0.01	272.89	2.25	305.9/0.2/843.2
SINGLEDICT*	4.85	3562.85	4.73	1.58	37.01	3.39	0.37	2.04	0.31	3.60	0.81	154.87	9.61	90.40	24.15	0.77	0.75	0.12	1438.82	64.46	270.3/4.2/816.7
ADAPTIVE*	2.04	3.31	0.28	0.69	55.52	7.98	0.08	0.20	0.02	0.28	0.16	272.67	7.59	266.22	31.69	3.12	0.15	0.06	1660.54	9.25	116.1/2.6/363.1
PREDICT	0.01	0.02	0.02	0.01	0.27	0.08	0.01	0.03	0.00	0.03	0.01	1.47	0.11	0.74	0.39	0.01	0.01	0.00	18.74	0.62	1.1/0.0/4.1
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	81.8	74.2	81.7	82.5	90.8	59.3	58.8	78.3	46.9	82.8	76.1	83.5	69.8	67.7	73.4	67.6	68.7	48.8	80.7	81.1	72.7/72.7/72.7
VCDIFF	63.6	46.3	62.0	63.2	90.3	25.2	9.2	58.3	10.4	76.3	57.5	66.5	37.9	38.5	36.6	57.4	22.1	0.0	62.3	64.2	47.4/47.4/47.4
FIXEDRATE*	85.8	76.9	85.5	87.3	94.2	63.2	62.1	82.9	53.4	87.0	81.8	85.9	71.9	70.9	75.6	77.4	73.7	55.0	83.3	82.9	76.8/5.2/-640.3
SINGLEDICT*	88.2	83.4	88.1	89.9	93.7	62.5	62.7	84.1	53.8	87.5	83.7	85.4	74.4	70.4	77.8	76.3	76.9	64.1	84.1	83.9	78.6/71.2/4.7
ADAPTIVE*	89.0	83.1	88.7	91.0	94.6	63.6	63.3	84.7	54.4	88.5	85.0	86.8	74.6	70.9	77.9	80.6	78.3	60.2	84.5	85.1	79.2/72.5/11.4
PREDICT	83.0	83.5	88.4	89.5	91.4	61.0	62.6	84.6	53.8	84.1	83.8	83.4	74.2	70.1	77.9	78.3	77.9	61.9	83.9	81.2	77.7/75.0/49.9

* Best of all permutations in terms of bandwidth reduction

Table B.1.2: Result summary (including all overhead) for batchsize 8

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.16	0.16	0.13	0.09	0.25	0.79	0.10	0.18	0.12	0.20	0.08	0.87	0.38	1.35	0.56	0.11	0.09	0.09	0.39	1.18	0.4/0.2/0.4
VCDIFF	0.27	0.39	0.23	0.24	0.33	2.40	0.23	0.41	0.20	0.26	0.15	1.97	0.92	2.70	1.66	0.17	0.19	0.14	0.79	3.00	0.8/0.3/0.9
FIXEDRATE*	0.44	0.54	0.36	0.25	1.85	5.52	0.30	0.63	0.18	0.56	0.26	9.72	1.43	11.48	2.36	0.22	0.24	0.22	0.93	5.64	2.2/0.6/3.2
SINGLEDICT*	1.32	0.98	1.05	0.51	10.00	9.22	0.63	2.69	0.45	1.60	0.71	13.14	5.07	28.08	10.21	0.32	0.42	0.33	4.66	22.17	5.7/1.5/7.6
ADAPTIVE*	0.54	1.08	0.52	0.33	3.68	5.38	0.43	1.23	0.33	0.77	0.39	10.24	2.21	19.63	7.27	0.30	0.30	0.28	2.19	12.31	3.5/0.9/5.1
PREDICT	0.47	1.03	0.37	0.29	2.13	3.03	0.42	0.69	0.32	0.63	0.32	5.77	1.84	7.72	4.16	0.28	0.30	0.27	0.89	5.80	1.8/0.7/2.2
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.04	0.04	0.03	0.03	0.07	0.15	0.03	0.05	0.03	0.06	0.02	0.15	0.08	0.23	0.11	0.03	0.03	0.02	0.08	0.26	0.1/0.0/0.1
VCDIFF	0.16	0.15	0.14	0.13	0.39	0.54	0.07	0.19	0.06	0.22	0.08	0.85	0.30	0.74	0.50	0.08	0.06	0.04	0.41	1.26	0.3/0.2/0.3
FIXEDRATE*	0.09	0.16	0.08	0.06	0.17	0.85	0.13	0.16	0.09	0.15	0.07	0.61	0.46	1.12	0.45	0.08	0.09	0.08	0.28	0.92	0.3/0.2/0.3
SINGLEDICT*	0.15	0.11	0.12	0.06	0.27	1.02	0.11	0.20	0.16	0.18	0.08	0.77	0.51	1.44	0.58	0.10	0.08	0.15	0.42	1.07	0.4/0.2/0.4
ADAPTIVE*	0.12	0.21	0.09	0.07	0.22	1.11	0.11	0.17	0.16	0.18	0.07	0.77	0.56	1.42	0.64	0.11	0.08	0.09	0.47	1.18	0.4/0.2/0.4
PREDICT	0.13	0.17	0.11	0.07	0.20	0.91	0.12	0.15	0.12	0.18	0.07	0.82	0.47	1.32	0.53	0.10	0.09	0.08	0.22	1.09	0.3/0.2/0.4
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.05	0.05	0.12	0.23	80.87	104.12	0.06	0.08	0.02	0.07	0.47	2357.68	1.81	2983.60	27.98	21.13	0.06	0.01	0.94	1.18	279.0/0.4/803.8
SINGLEDICT*	5.15	3147.06	4.79	1.49	37.33	3.65	0.36	2.15	0.28	3.67	0.72	156.14	10.53	90.94	26.21	0.71	0.70	0.14	1528.75	64.19	254.2/4.2/741.2
ADAPTIVE*	0.23	1998.99	1.83	0.56	141.57	4.44	0.52	1.50	0.54	0.35	0.37	165.57	2.60	320.76	52.20	4.71	0.39	0.02	1057.79	12.87	188.4/2.2/477.4
PREDICT	0.01	0.02	0.02	0.01	0.28	0.09	0.00	0.02	0.01	0.04	0.01	1.44	0.11	0.72	0.33	0.01	0.01	0.01	17.50	0.69	1.1/0.0/3.8
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	84.6	77.8	84.6	85.9	92.5	60.1	60.3	80.5	50.5	84.9	79.6	84.5	72.0	69.8	75.7	72.7	72.2	53.7	82.5	82.6	75.3/75.3/75.3
VCDIFF	65.1	49.3	63.7	65.5	91.1	25.8	13.7	60.5	15.6	77.4	60.1	68.4	40.4	43.3	42.9	60.7	22.8	0.0	63.5	65.2	49.7/49.7/49.7
FIXEDRATE*	86.6	79.1	87.1	88.4	94.3	63.1	62.6	83.3	54.0	87.3	83.4	86.4	73.2	72.0	77.4	79.8	74.8	56.4	83.0	83.5	77.8/-22.1/-921.8
SINGLEDICT*	88.5	83.6	88.5	90.2	93.8	62.4	63.3	84.4	54.5	87.8	84.3	85.6	74.8	71.4	78.6	78.2	77.8	64.6	84.1	84.0	79.0/71.5/3.8
ADAPTIVE*	88.6	83.8	89.7	90.8	95.3	63.4	63.9	85.7	55.4	88.4	85.3	87.0	75.0	72.4	79.4	80.5	78.5	61.8	84.7	85.3	79.7/73.1/13.1
PREDICT	85.5	83.6	85.5	89.5	93.0	61.9	63.4	82.2	55.0	85.9	84.2	84.5	74.7	71.4	78.6	79.7	77.5	61.9	84.2	82.8	78.2/75.5/51.0

* Best of all permutations in terms of bandwidth reduction

Table B.1.3: Result summary (including all overhead) for batchsize 13

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.13	0.16	0.10	0.09	0.24	0.95	0.11	0.17	0.10	0.17	0.10	0.86	0.45	1.56	0.59	0.10	0.09	0.09	0.34	1.52	0.4/0.2/0.5
VCDIFF	0.26	0.50	0.23	0.26	0.37	3.09	0.25	0.38	0.25	0.35	0.20	1.94	1.29	3.53	1.68	0.22	0.35	0.21	0.72	4.37	1.0/0.4/1.2
FIXEDRATE*	0.49	0.75	0.42	0.23	2.64	5.75	0.30	0.87	0.24	0.70	0.29	9.35	1.51	11.22	2.94	0.21	0.24	0.17	1.56	7.64	2.4/0.7/3.3
SINGLEDICT*	1.39	1.08	1.19	0.56	11.14	10.06	0.74	2.98	0.51	1.94	0.83	15.01	5.11	24.80	11.09	0.41	0.43	0.31	5.25	22.98	5.9/1.7/7.4
ADAPTIVE*	0.80	1.01	0.54	0.38	5.46	7.29	0.44	1.24	0.31	0.91	0.46	11.20	2.45	19.72	5.07	0.34	0.33	0.26	1.42	14.23	3.7/1.0/5.3
PREDICT	0.61	0.97	0.45	0.30	3.01	3.74	0.30	0.92	0.34	0.78	0.32	6.32	2.11	6.99	4.49	0.27	0.30	0.28	0.94	7.07	2.0/0.9/2.3
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.03	0.03	0.03	0.02	0.05	0.16	0.03	0.04	0.02	0.04	0.03	0.15	0.08	0.25	0.13	0.02	0.02	0.02	0.07	0.20	0.1/0.0/0.1
VCDIFF	0.16	0.20	0.13	0.13	0.39	0.63	0.07	0.17	0.07	0.25	0.10	0.73	0.36	0.87	0.45	0.11	0.10	0.04	0.36	1.46	0.3/0.2/0.3
FIXEDRATE*	0.10	0.19	0.09	0.05	0.15	0.78	0.09	0.16	0.13	0.16	0.06	0.56	0.39	1.03	0.46	0.07	0.08	0.06	0.38	0.87	0.3/0.2/0.3
SINGLEDICT*	0.15	0.11	0.13	0.08	0.22	0.91	0.13	0.19	0.19	0.17	0.09	0.77	0.45	1.36	0.57	0.12	0.08	0.07	0.40	1.11	0.4/0.2/0.4
ADAPTIVE*	0.14	0.18	0.12	0.07	0.21	1.06	0.11	0.16	0.13	0.17	0.08	0.74	0.47	1.34	0.55	0.10	0.08	0.08	0.28	1.09	0.4/0.2/0.4
PREDICT	0.12	0.17	0.10	0.07	0.19	0.97	0.10	0.16	0.12	0.20	0.07	0.63	0.50	1.17	0.53	0.09	0.08	0.08	0.21	0.99	0.3/0.2/0.3
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.09	0.12	0.08	0.05	30.36	96.00	0.03	0.25	0.04	0.14	0.66	3573.89	0.75	2837.17	17.63	27.47	0.01	0.02	1.60	4.56	329.5/0.5/966.0
SINGLEDICT*	5.26	2863.95	5.38	1.81	39.81	3.13	0.37	1.99	0.34	3.21	0.74	156.48	9.35	88.66	27.41	0.86	0.84	0.12	1666.52	62.02	246.9/4.2/700.0
ADAPTIVE*	5.99	1977.09	0.34	1.27	177.82	4.40	0.43	0.90	0.20	1.44	1.23	306.78	5.10	473.27	19.15	11.08	0.50	0.07	3199.83	10.11	309.8/4.8/793.0
PREDICT	0.02	0.03	0.02	0.01	0.27	0.09	0.01	0.02	0.01	0.04	0.01	1.67	0.09	0.70	0.28	0.01	0.01	0.00	16.68	0.62	1.0/0.0/3.6
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	86.5	80.2	86.5	88.0	93.6	60.5	61.2	82.0	52.9	86.2	81.9	85.3	73.7	71.5	77.8	76.6	74.3	57.4	83.8	83.6	77.2/77.2/77.2
VCDIFF	66.6	52.1	65.3	67.9	91.6	26.3	18.0	62.5	20.7	78.5	62.3	69.8	42.8	47.9	48.2	62.9	23.5	0.0	64.5	66.0	51.9/51.9/51.9
FIXEDRATE*	87.5	81.5	88.2	89.6	94.5	63.0	63.1	83.8	54.4	87.7	84.7	86.6	74.5	73.3	78.5	81.2	76.2	57.9	83.9	84.2	78.7/-10.6/-814.9
SINGLEDICT*	88.7	83.9	88.7	90.5	94.0	62.4	63.8	84.6	55.4	87.9	84.8	85.6	75.3	72.6	79.6	79.7	78.3	65.2	84.0	84.1	79.5/71.9/4.2
ADAPTIVE*	89.1	84.4	89.2	91.9	95.7	63.4	64.5	85.9	56.2	89.4	86.0	87.1	75.8	73.6	79.5	82.5	79.0	63.3	85.2	85.3	80.4/73.8/14.3
PREDICT	87.1	83.1	87.3	88.7	94.1	62.1	63.4	83.5	55.8	87.1	83.2	85.4	75.1	72.4	79.4	81.2	77.6	62.9	84.1	83.9	78.9/76.3/52.9

* Best of all permutations in terms of bandwidth reduction

Table B.1.4: Result summary (including all overhead) for batchsize 21

Dataset	Chi-B	MET	NYC-B	GTFS-E	Bart	BTC	DEBS	DEBS	DEBS	DEBS	Gdax	GH	Comm	Eve	RSVPS	Neut	sat	tpch	twitch	Twitter	aggregated
Format	json	json	json	json	json	json	csv	json	pb	xml	json	json	json	json	json	log	json	tbl	json	json	
Absolute CPU cost for all publishers in minutes																					mean/med/std
DEFLATE	0.14	0.21	0.12	0.10	0.28	0.95	0.08	0.16	0.12	0.17	0.08	1.00	0.54	1.57	0.67	0.08	0.08	0.08	0.37	1.58	0.4/0.2/0.5
VCDIFF	0.33	0.46	0.24	0.26	0.32	3.90	0.27	0.53	0.25	0.32	0.21	3.44	1.31	5.41	2.42	0.20	0.24	0.16	1.08	7.18	1.4/0.3/2.0
FIXEDRATE*	0.56	0.84	0.49	0.33	3.65	6.33	0.41	0.90	0.27	0.88	0.35	9.50	2.02	9.62	3.97	0.25	0.27	0.26	1.69	9.57	2.6/0.9/3.3
SINGLEDICT*	1.65	1.09	1.38	0.65	11.86	13.81	0.72	3.53	0.46	2.49	0.85	17.78	6.50	26.00	11.94	0.33	0.46	0.39	3.66	24.77	6.5/2.1/8.1
ADAPTIVE*	0.96	0.69	0.78	0.49	5.96	9.93	0.47	1.54	0.28	1.08	0.51	12.79	2.82	19.62	5.83	0.31	0.34	0.28	1.36	20.42	4.3/1.0/6.2
PREDICT	0.73	1.16	0.60	0.36	4.37	4.41	0.39	1.38	0.34	0.97	0.44	6.89	2.15	7.48	4.40	0.32	0.38	0.28	1.09	7.53	2.3/1.0/2.5
Absolute CPU cost per subscriber in minutes																					mean/med/std
DEFLATE	0.02	0.03	0.02	0.02	0.05	0.17	0.02	0.03	0.03	0.03	0.02	0.17	0.09	0.21	0.11	0.02	0.02	0.02	0.07	0.20	0.1/0.0/0.1
VCDIFF	0.17	0.18	0.13	0.14	0.40	0.54	0.08	0.23	0.07	0.24	0.09	1.14	0.32	0.82	0.48	0.10	0.07	0.04	0.39	1.22	0.3/0.2/0.3
FIXEDRATE*	0.10	0.14	0.09	0.05	0.15	0.71	0.09	0.13	0.10	0.14	0.07	0.53	0.41	0.89	0.48	0.08	0.07	0.08	0.28	0.86	0.3/0.1/0.3
SINGLEDICT*	0.13	0.10	0.13	0.07	0.21	0.89	0.12	0.18	0.15	0.18	0.08	0.75	0.45	1.35	0.51	0.09	0.09	0.09	0.28	1.11	0.3/0.2/0.4
ADAPTIVE*	0.14	0.09	0.11	0.07	0.17	0.91	0.12	0.18	0.13	0.19	0.07	0.69	0.48	1.25	0.53	0.10	0.08	0.08	0.19	1.17	0.3/0.2/0.4
PREDICT	0.11	0.18	0.10	0.07	0.16	0.88	0.11	0.18	0.14	0.18	0.08	0.59	0.44	1.01	0.47	0.09	0.08	0.07	0.22	0.86	0.3/0.2/0.3
Absolute CPU cost for SB in minutes																					mean/med/std
FIXEDRATE*	0.03	0.17	0.05	0.06	10.13	127.08	0.03	0.21	0.01	0.18	0.48	2638.93	1.51	1923.66	19.56	17.18	0.03	0.04	133.15	2.39	243.7/0.3/689.6
SINGLEDICT*	4.72	2530.45	4.54	1.22	37.35	3.33	0.35	1.90	0.31	3.21	0.64	151.82	9.98	85.30	31.86	0.71	0.73	0.14	1733.37	64.10	233.3/3.9/646.4
ADAPTIVE*	9.69	2529.29	5.35	3.01	143.70	12.47	0.35	0.22	0.11	0.34	0.44	35.95	1.26	450.34	10.47	4.76	0.90	0.03	3313.44	146.75	333.4/5.1/877.5
PREDICT	0.02	0.02	0.02	0.01	0.33	0.10	0.01	0.03	0.01	0.04	0.01	1.50	0.10	0.75	0.33	0.01	0.01	0.00	13.41	0.65	0.9/0.0/2.9
Total bandwidth reduction in % with different subscriber/publisher ratio 1:n																					1/1k/10k pub
DEFLATE	87.7	82.0	87.8	89.5	94.4	60.9	61.9	83.0	54.8	87.0	83.5	85.9	75.2	72.7	79.5	79.4	75.9	60.5	84.7	84.2	78.5/78.5/78.5
VCDIFF	68.2	55.0	67.6	70.0	95.3	27.0	21.8	64.3	25.2	79.4	64.6	70.7	45.0	51.9	52.7	64.6	35.6	0.0	65.3	66.8	54.6/54.6/54.6
FIXEDRATE*	88.2	82.3	88.5	90.2	94.7	62.7	63.4	84.2	55.3	87.8	84.9	86.6	75.5	74.4	79.9	81.7	76.8	59.6	84.6	84.5	79.3/-13.1/-845.5
SINGLEDICT*	88.8	84.2	88.9	90.8	94.4	62.3	64.4	84.7	56.3	88.0	85.1	85.7	75.9	73.9	80.6	81.0	78.9	65.7	84.6	84.2	79.9/72.3/4.0
ADAPTIVE*	89.8	85.0	90.2	92.1	95.8	63.2	65.1	85.3	57.0	88.9	86.6	86.8	76.2	74.9	80.8	82.4	79.6	63.7	85.7	85.7	80.7/74.7/20.4
PREDICT	88.3	83.5	88.5	90.1	94.8	62.3	64.2	84.4	56.5	88.0	84.4	86.2	75.5	74.4	80.3	81.9	78.5	64.7	84.6	84.7	79.8/77.2/54.2

* Best of all permutations in terms of bandwidth reduction

Table B.1.5: Result summary (including all overhead) for batchsize 34

Appendix C

Pulsar evaluation

C.1 Subscriber faster compared to no compression

Grey areas denote missing results because of connection timeouts.

C.2 Publisher faster compared to no compression

Grey areas denote missing results because of connection timeouts.

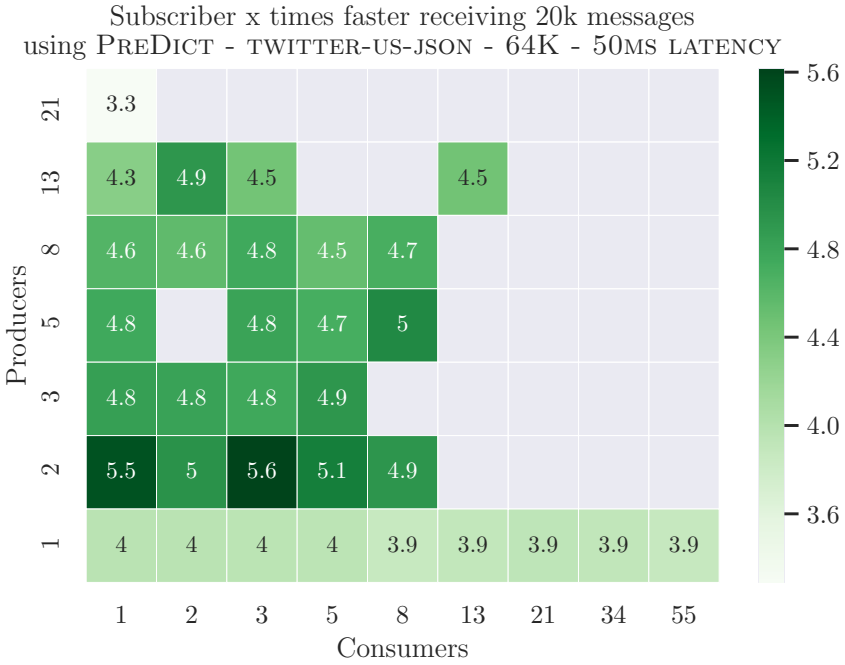


Figure C.1.1: Heatmap - Consumer faster 64K

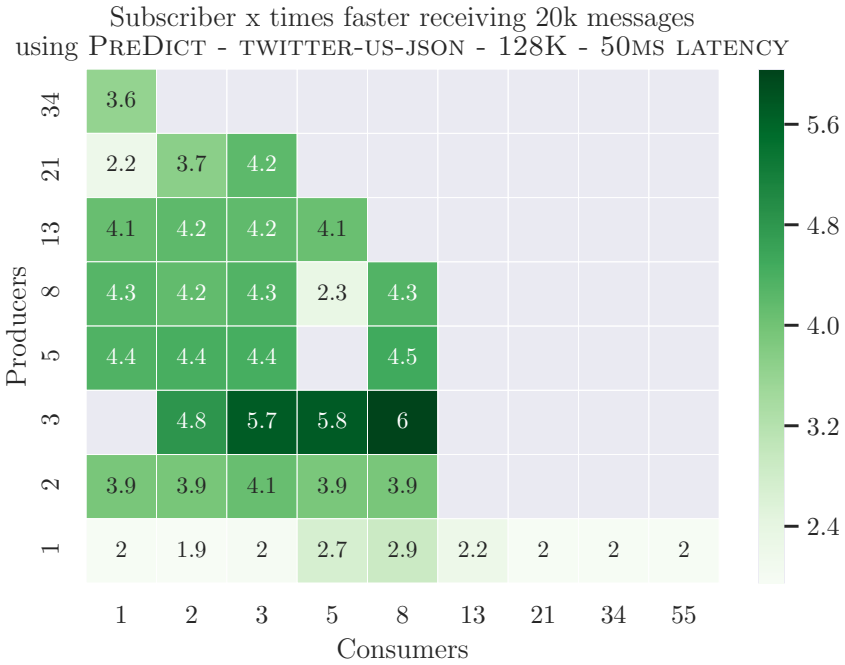


Figure C.1.2: Heatmap - Consumer faster 128K

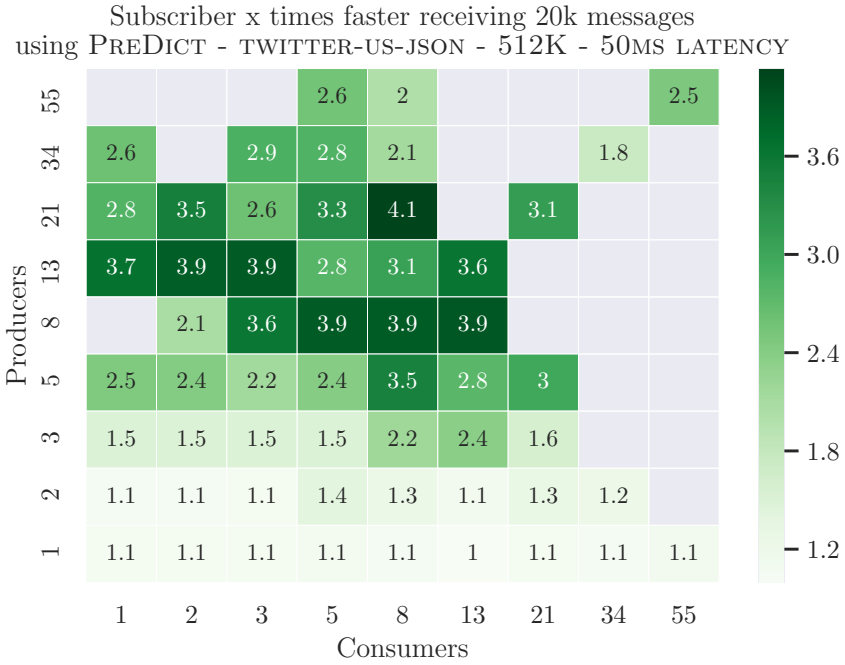


Figure C.1.3: Heatmap - Consumer faster 512K

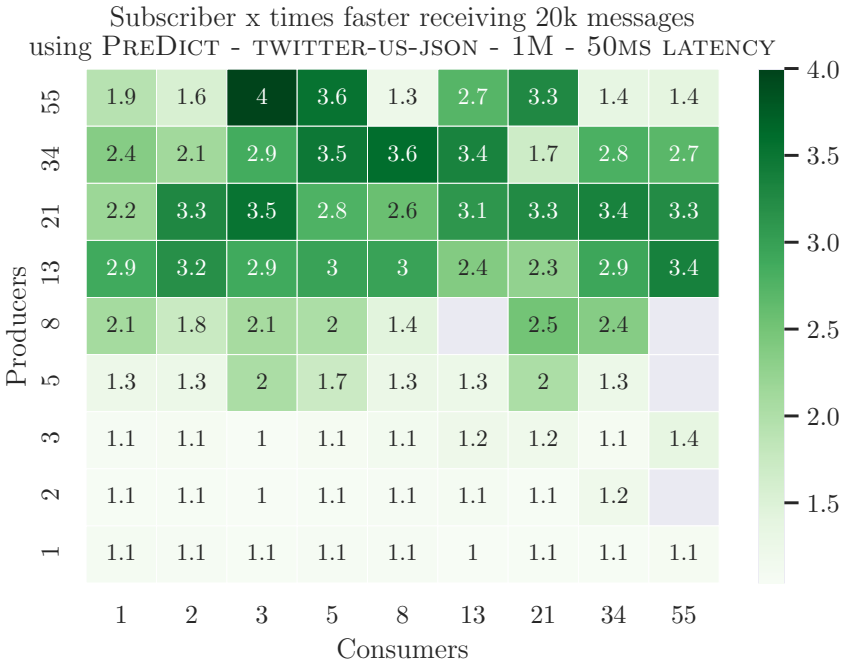


Figure C.1.4: Heatmap - Consumer faster 1M

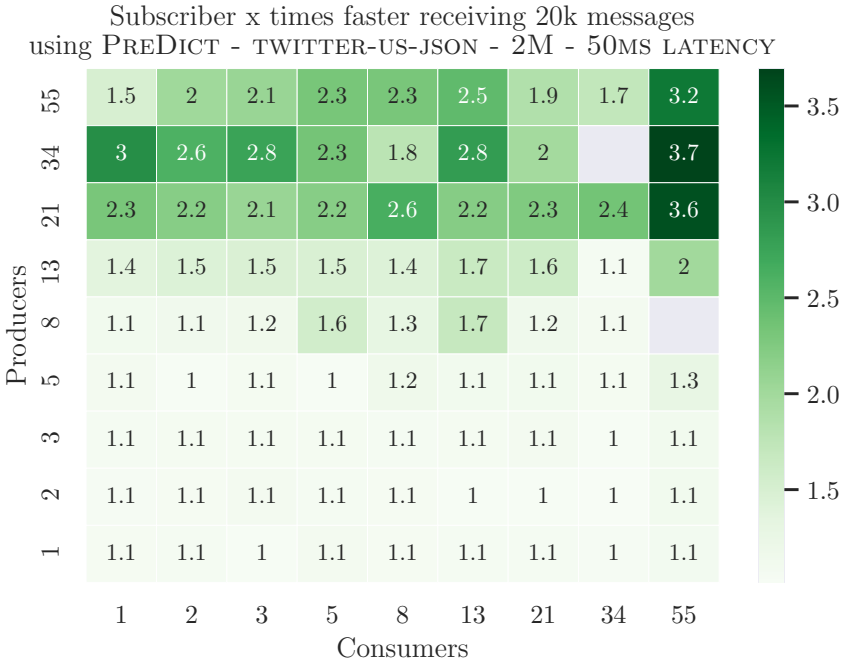


Figure C.1.5: Heatmap - Consumer faster 2M

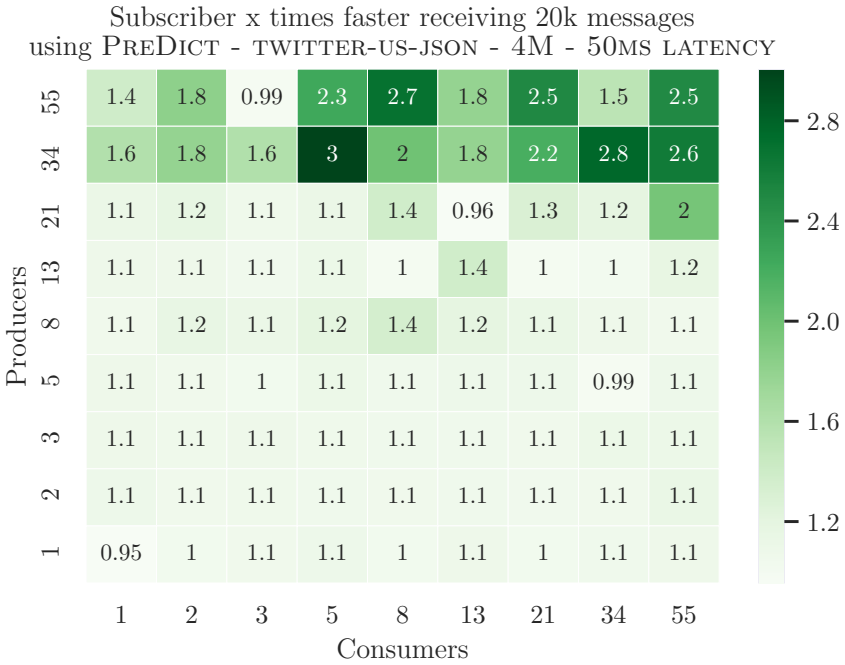


Figure C.1.6: Heatmap - Consumer faster 4M

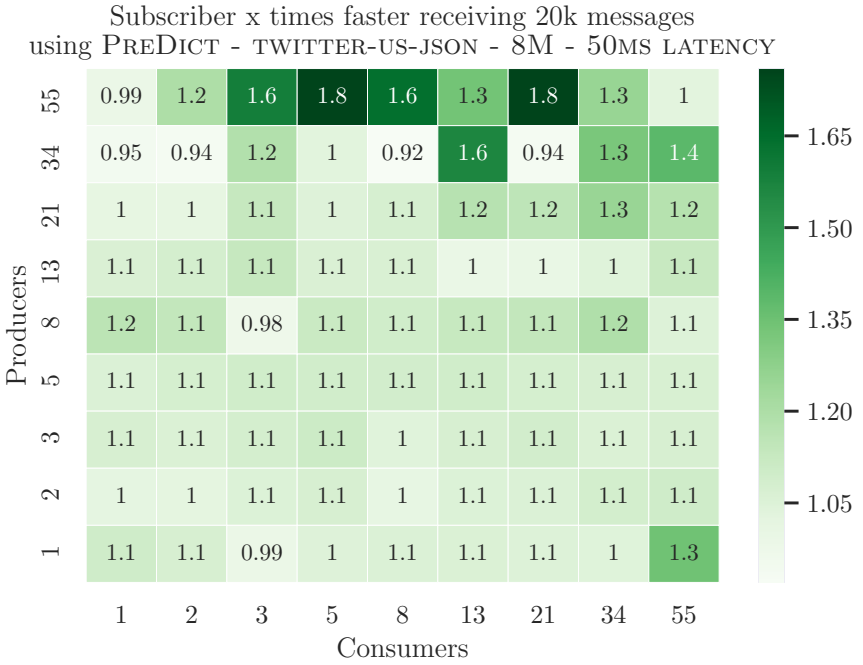


Figure C.1.7: Heatmap - Consumer faster 8M

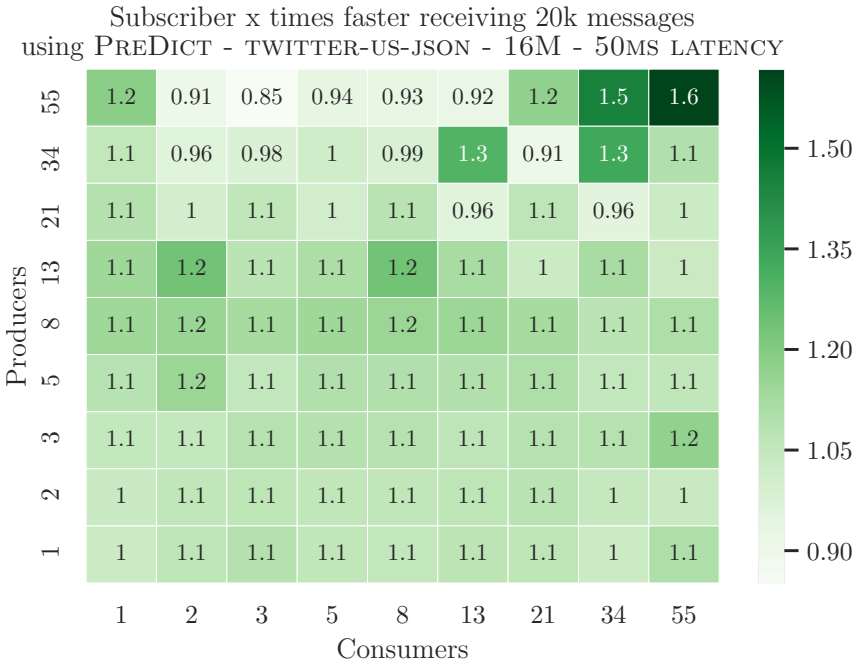


Figure C.1.8: Heatmap - Consumer faster 16M

Figure C.2.1: Heatmap - Producer faster 64K

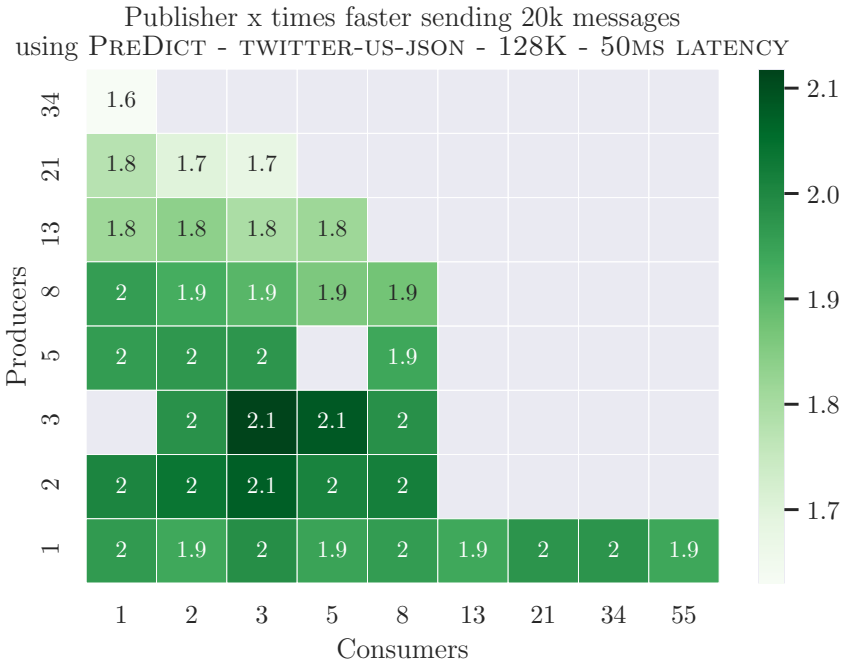


Figure C.2.2: Heatmap - Producer faster 128K

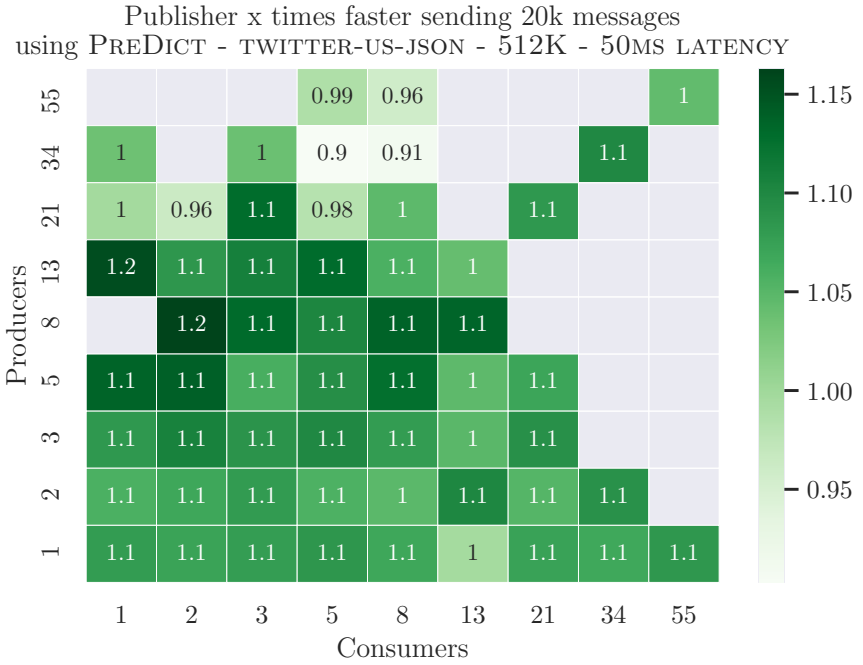


Figure C.2.3: Heatmap - Producer faster 512K

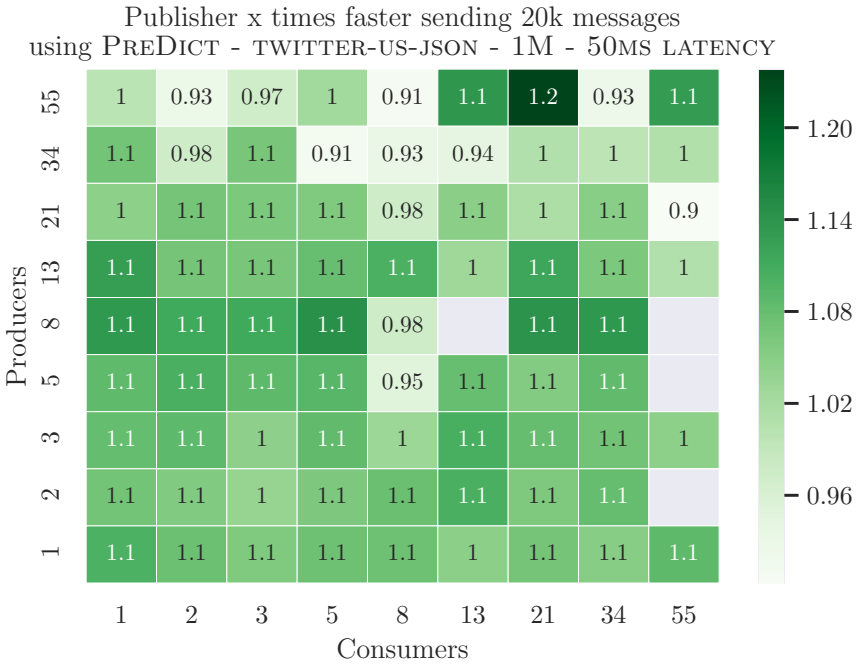


Figure C.2.4: Heatmap - Producer faster 1M

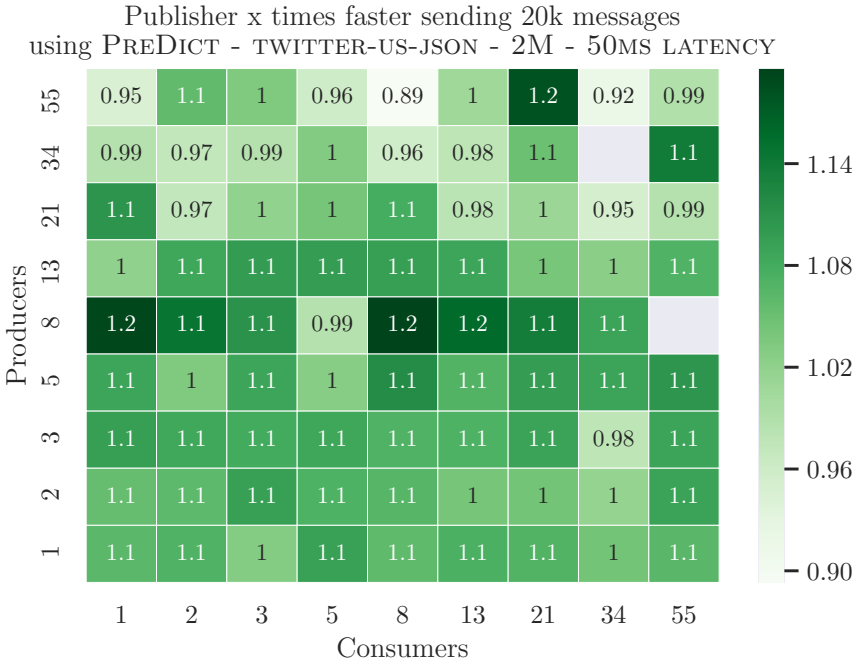


Figure C.2.5: Heatmap - Producer faster 2M

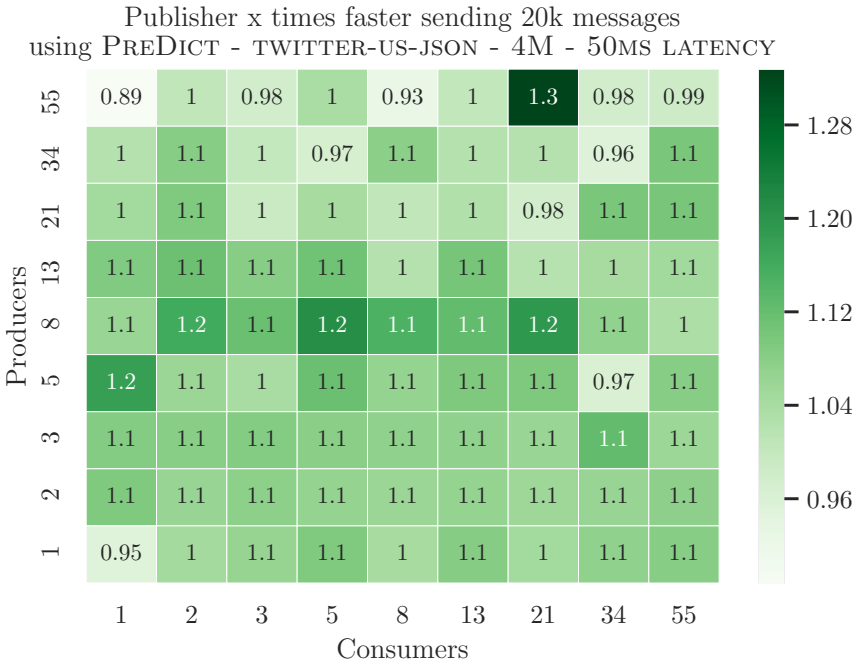


Figure C.2.6: Heatmap - Producer faster 4M

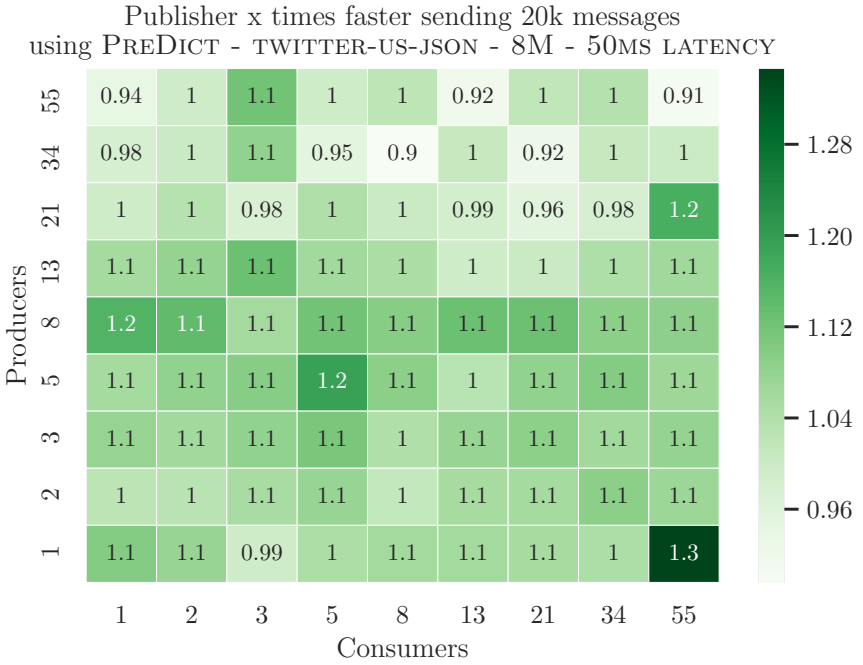


Figure C.2.7: Heatmap - Producer faster 8M

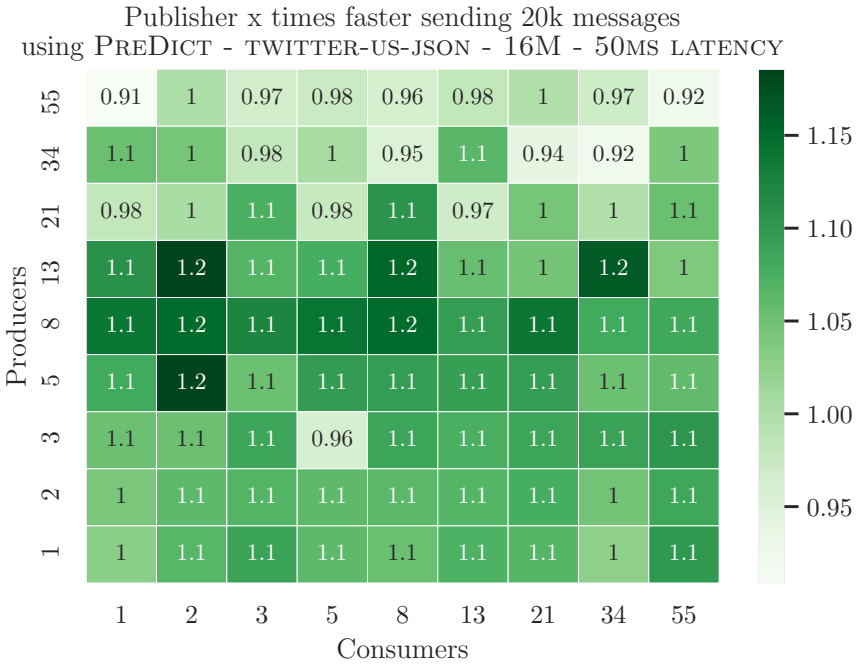


Figure C.2.8: Heatmap - Producer faster 16M

Appendix D

TAPD

D.1 Additional results for TaPD

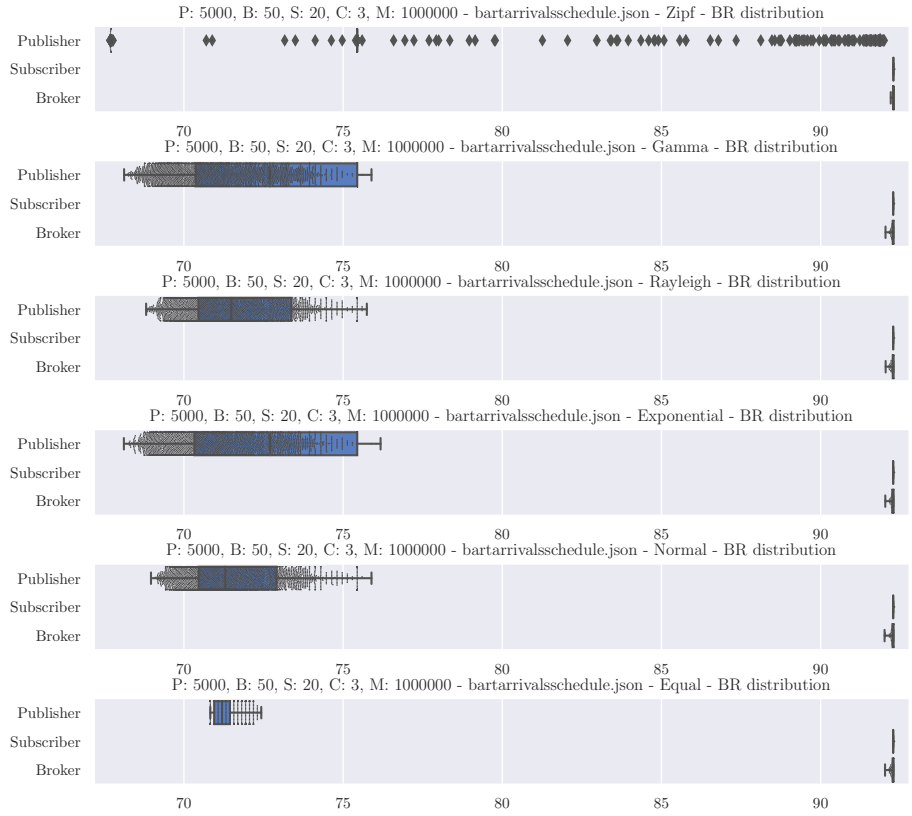


Figure D.1.1: Distribution for dataset Bart-json

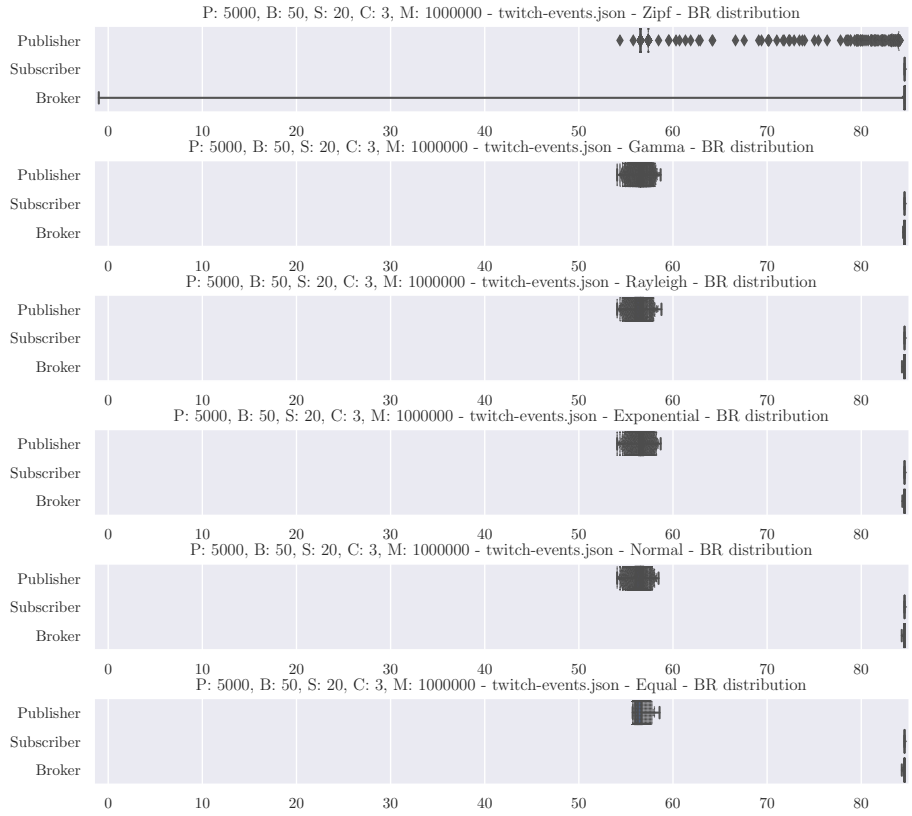


Figure D.1.2: Distribution for dataset twitch-Eve-json

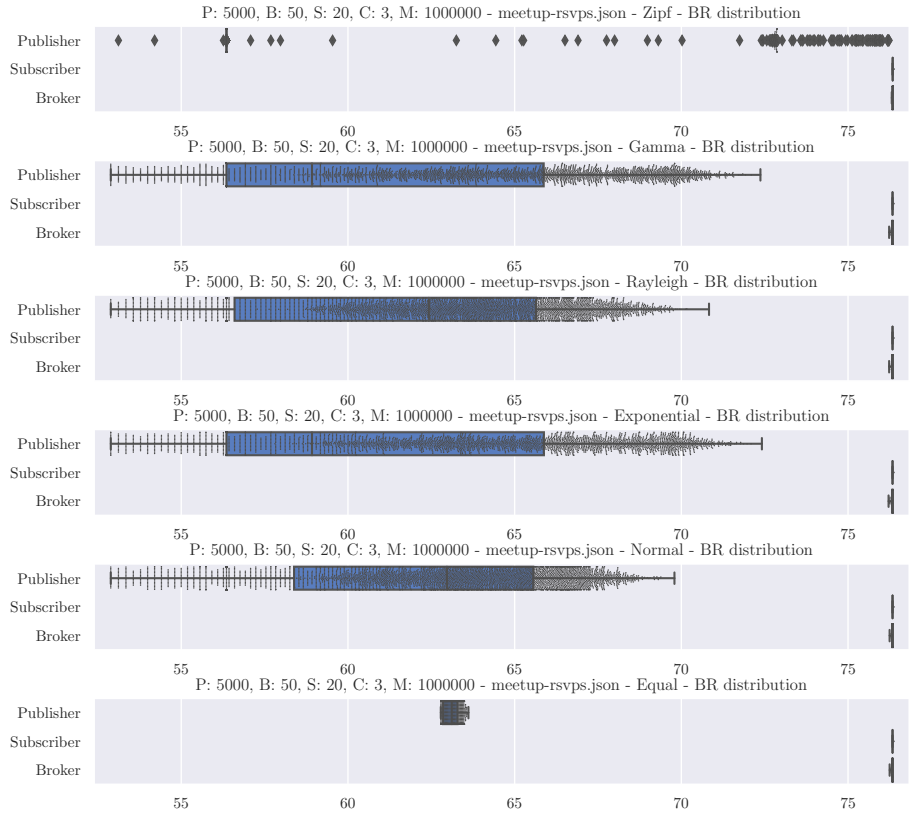


Figure D.1.3: Distribution for dataset rsvps-json