THE UNIVERSITY OF CHICAGO

# FAST AND EFFECTIVE COMPRESSION FOR IOT SYSTEMS

A DISSERTATION PROPOSAL SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
CHUNWEI LIU

CHICAGO, ILLINOIS
JANUARY 2022

# Table of Contents

# List of Figures

# List of Tables

# ACKNOWLEDGMENTS

TODO

# ABSTRACT

Nowadays, the Internet of Things (IoT) is compelling as it enables connections of trillions of sensors and data collection for connectivity and analytics. The amount of IoT-generated data has exploded due to the rapid growth of interconnected IoT devices from a wide range of IoT applications. IoT systems need an effective solution to ingest, store, and analyze the exploding IoT data to make the best of the limited resources in IoT systems, such as network, storage, energy, and computation power. Due to the extensive usage of IoT across diverse applications, IoT data includes dynamic data streaming with various data types and changing data statistics. Given the special features of IoT data: endless, heterogeneous, and dynamic, we introduce new compression techniques to handle those special data features: MOP handles the endlessness of IoT data by pre-allocating encoding space for the dynamic incoming data and enables in-situ queries in the encoded domain for fast query execution. BUFF addresses the heterogeneity of IoT data by applying decomposed but compact encoding space for the IoT numeric data with different statistics and achieves efficient query execution support according to the host's hardware. In addition, the dynamic IoT data imposes challenges to the traditional compression selection strategies as there is no one-size-fits-all solution for IoT systems with varying data statistics, different workloads, and constrained hardware resources. We, therefore, propose AdaEdge as a hardware-conscious encoding selection framework for resource-constrained devices.

# CHAPTER 1

# THESIS STATEMENT

Modern applications and systems are generating vast amounts of data every day. It is expected that the amount of data generated in the future will continue to grow and largely be driven by IoT devices. IoT systems need an effective solution to ingest, store and analyze the exploding IoT data to make the best of the limited resources in IoT systems. To solve this problem, we propose new compression techniques that compress the endless and heterogeneous IoT signal data quickly and effectively while supporting efficient query execution according to the hardware specifics. These compression techniques, together with resource-aware compression selection strategy, can adaptively select the optimal compression approach to meet the edge resource constraints and workload requirements.

- **Endless streaming data** The Compression for IoT data should handle the endlessness of the signal streaming data and adapt to the data distribution change. The compression should provide efficient in-situ query execution in the encoded space without heavy decoding overhead.

- **Heterogeneous numeric data** The compression approach has to handle the heterogeneity of IoT data, including a wide range of data types, varying data statistics such as distribution, range, precision. The compression should also provide efficient query execution support for both simple edge devices and advanced cloud servers.

- **Compression selection for dynamic data** The compression selection strategy should be aware of changing data statistics, workload and the limited edge resource in different IoT scenarios, including varying network bandwidth and connectivity, limited local storage and power support.

# CHAPTER 2

# INTRODUCTION

With the rapid growth of interconnected IoT devices including consumer electronics such as smartphones, smart home devices, autonomous vehicles, wearables and connected healthcare as well as in industrial applications with the rise of industrial IoT. it is becoming common to collect a large amount of sensor-generated data series before any downstream analytic are performed. Recently, the International Data Corporation predicted that the global amount of data will reach $175ZB$ by 2025 [80]. Billions of IoT devices will be a significant driver of this growth, which contribute more than 50% of the data volume. To keep up with the storage demands stemming from all this data creation, a new effective data storage layout and compression technique is necessary in addition to relying on increasing capacity supplied from the storage industry.

Many IoT systems use the cloud as a backend for data and query processing. Depending on the IoT system design, the sensor sends data to its hub node - an edge device or a router. The data is ingested and assembled into segments before sending to the cloud. Then the following data processing steps like compression, indexing, archiving, query are all processed on the cloud side. However, the cloud supported IoT framework has the following problem: Data transmission to the cloud is costly, especially for cases with unstable network connections or weak power supplies. Additionally, all raw data is uploaded to the cloud. Some of those data are useless and may never be touched in the future, which wastes a lot of network and storage resources. Query execution on the cloud causes long query latency compared with in-situ query execution. Besides, the query execution depends on the network, and the system will be down if there is no network connection.

As edge devices are getting more computational power and edge computing is becoming more popular, pushing down compression, query, and analysis on the edge side is an appealing option. Those pushing down make it possible to apply compression on the ingested data in the earlier phase. It is beneficial to many aspects of the IoT system: minimizing the

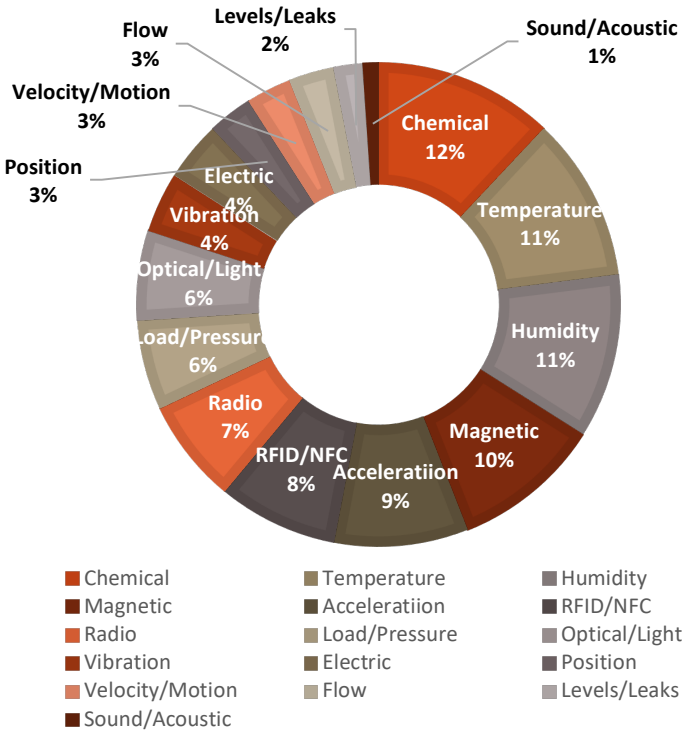



Figure 2.1: Various categories of IoT sensors

data size needed to send, saving network bandwidth, saving the edge /cloud resources for storing data, and speeding up the query with some encoding techniques. The pushing down also allows some query being processed by edge directly, which could significantly decrease query latency and make the system more robust to the network disconnection. However, the edge is usually a resource-constrained device compared with a more advanced cloud server. The limited resource will bring some challenges to compression design and query execution. the compression for IoT data should consider three unique features of real-time IoT data:**heterogeneous**, **endless** and **dynamic**. IoT system generates heterogeneous data because of the variety of data acquisition devices [57], as is shown in Figure 2.1. The heterogeneous data here does not only mean the data with many different data types but also the data with varying data statistics, including range, precision, cardinality, distributions, etc. IoT data is very different from the historical dataset. The historical dataset is static such that you can get the whole statistics for your data processing task. In contrast, the IoT

system generates endless streaming, and its statistics change over time.

To compress IoT data efficiently, we need to build dedicated compression approaches for IoT data and a compression selection framework to handle the changing data features. We propose MOP [61] to address the endless of IoT data by providing a fast single pass, order-preserving dictionary encoding for streaming data. We introduce BUFF [60] to deal with the heterogeneous data from IoT systems by enabling range and precision aware decomposed float compression for IoT numeric data. With extensive evaluation on many compression for IoT data, we find there is no one-size-fits-all solution for IoT data compression. An adaptive compression selection strategy is needed to choose the best compression fit for the incoming IoT data based on the data statistics and workload. We here give each system a high level overview as following.

**MOP**: Dictionary encoding, or domain encoding, is an important form of compression that uses a bijective mapping to replace attributes from a large domain (i.e. strings) with a finite domain (i.e. 32 bit integers). This encoding both reduces data storage and allows for more efficient query execution. Traditional dictionary encoding only supports efficient equality queries, while range queries require that encoded values are decoded for evaluating the predicates. An order preserving dictionary allows for range queries without decoding by ensuring that encoded keys follow the same order as the values in the dictionary. While this approach enables efficient queries it requires that the full set of values is known to create the mappings. In this work we bridge this gap by introducing mostly ordered dictionaries that use a best effort dictionary generation based on sampling the input dataset. Query evaluation on a mostly ordered dictionary avoids decoding when possible and gracefully degrades performance as the ratio of ordered values decreases.

**BUFF**: Modern data-intensive applications often generate large amounts of low precision float data with a limited range of values. Despite the prevalence of such data, there is a lack of an effective solution to ingest, store, and analyze bounded, low-precision, numeric data. To address this gap, we propose Buff, a new compression technique that uses a decomposed

columnar storage and encoding methods to provide effective compression, fast ingestion, and high-speed in-situ adaptive query operators with SIMD support. Compared to the state-of-the-art float compression techniques, Buff achieves up to 50× speedup for filtering and aggregations, and 1.5× improvement for ingestion with a single thread, while offering comparable compression sizes to the state-of-the-art approaches.

In the rest of this dissertation proposal, we first present the aforementioned mostly dictionary encoding designed for endless streaming data in Chapter 3, and range and precision aware decomposed float compression in 4. Then we discuss the current progress and research plan of the next project in Chapter 5.

# CHAPTER 3

# MOSTLY ORDER PRESERVING DICTIONARIES

## 3.1 Introduction

Database compression is critical for current data-intensive systems where the rate of data growth is outstripping growth of processing speeds, memory capacity, and I/O bandwidth. Columnar databases enable efficient compression in two primary ways. First, the ability to organize data by attributes reduces entropy and thereby improves compression effectiveness. Second, through use of columnar encoding [9] (i.e. run-length encodings, delta encoding, and dictionary encoding) the database supports efficient in-situ query processing, whereas prior use of byte-oriented compression (i.e. gzip, snappy) requires decompression as a blocking step before query execution [37], which can be CPU-intensive [28].

While all columnar encodings offer compression benefits over unencoded data, query benefits can be limited given the value domain and data distribution [9]. For non-numeric data types, such as dates and strings, dictionary encoding can translate a large and near infinite domain to a smaller finite and dense domain, often integers [82]. Dictionary encoding, or domain encoding, creates a bijective mapping from an arbitrary infinite source domain (*value domain*) to a fixed size target domain (*code domain*). The translated codes as well as the mapping are stored as the encoded result. Since the domain is smaller, it allows values further compression using methods such as bit-packed encoding (e.g. truncating unnecessary bits, such as using 3 bits to represent integers 0-7) [82, 44], as well as supporting fast query execution via efficient hardware instructions [44]. As a result, dictionary encoding is widely supported in many analytic platforms systems [86, 12, 82, 1, 62, 38], and offers significant benefits for string data types that are common in many domains, such as enterprise data [29] and open data initiatives [44]. Note that we limit our focus to global dictionaries that maintain a single mapping for an entire column, instead of a local per-block dictionary.

For query filtering with equality predicates, dictionary encoding allows the query to

translate the predicate value(s) to the code domain and evaluate the predicates directly on the encoded data. Prior work studies variations of dictionary encoding algorithms on their trade-off between compression ratios and decoding speed [65]. However, for query range predicates, the system must decode encoded values to evaluate the predicates. To avoid this expensive translation, the use of order-preserving dictionaries can allow range predicates to be evaluated without decoding [13]. Here, order-preserving dictionaries work by ensuring that encoded keys maintain the same order as values (e.g. for mappings $k_1 = v_1$ and $k_2 = v_2$, $k_1 < k_2$ iff $v_1 < v_2$). Note that we refer to order-preserving dictionaries as *OP* for short.

Generating the mapping to support OP dictionaries comes at a cost. If the code domain is a finite domain, such as 32-bit integers, then the entire set of values must be known and fixed before the dictionary encoding can occur, as preserving the order requires sorting the values first [13]. If the code domain is an infinite domain, such as double values, an order preserving scheme can be generated for a set of unknown values using a Dewey Decimal style coding, where new values can be inserted using increasing precision [87]. While this approach works when the domain is unknown or not-fixed, it suffers from reduced compression (e.g. larger code domain values and no bit-packing), less efficient CPU operations, and lacks the ability for dense SIMD operations that depend on small key values [44].

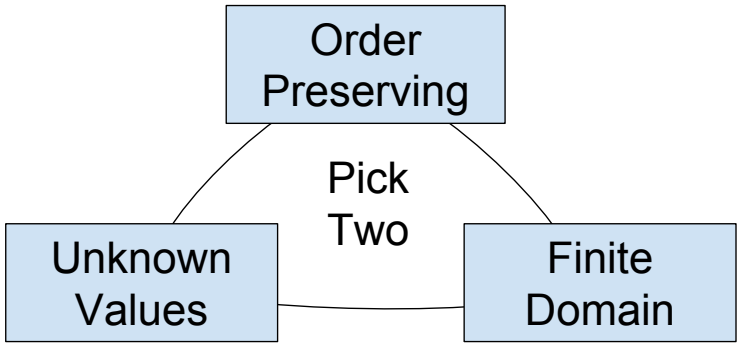



Figure 3.1: Tension between three properties of OP dictionary

Therefore, in Figure 3.1, we propose a conjecture for dictionary encodings that it is impossible to simultaneously provide more than two out of the three properties: the dictionary is order preserving (i.e. OP), the encoded keys have a finite domain (i.e. integer codes), and if the encoding tolerates an unknown value domain (i.e. unknown values). For example, if we support an unknown value domain and OP, the integer code domain property can be violated when there is no more "code space" for new values, and a more complex code domain must be used to keep the OP property. Otherwise, if we require OP and integer codes, recoding (e.g. redoing the encoding with a known domain) is needed when there is no valid code space to maintain ordering.

While many applications know the domain of values to encode a priori, several important scenarios exist where the values are unknown when the encoding occurs. First, for data that is continuously being loaded into a data warehouse or lake, the entire set of values may not ever be known or known before analysis must begin – either of which could prevent use of an OP dictionary if integer codes are desired. Second, for massive static datasets, generating an OP dictionary requires taking a full pass on the dataset before encoding to learn the full value set. Given the potential costs of scanning and parsing the dataset, this extra pass may be prohibitive for scenarios that want to begin analysis quickly.

Observe that if a dictionary encoding supports unknown values and an integer code domain, we may achieve the OP goal by pre-allocating a huge code space for a dictionary, but at the cost of larger bit representation for each value. Despite this, one cannot guarantee there will be no code space conflicts in the future, especially in the presence of skewed data distributions. This inability to guarantee ordering though, leads to the key research questions of this paper which are *can a database system leverage a dictionary encoding that is partially ordered* and *given the ability to sample a dataset, what is the ideal pre-allocation key allocation?*

In this paper, we present a best-effort order preserving dictionary encoding: a mostly order preserving dictionary encoding (*MOP*). It pre-allocates a code space for an order

preserving dictionary based on estimated statistics, such as record count and cardinality. A backup code space is reserved to handle potential code space conflicts, which we explore both with unordered and cascading ordered variations. For distributed generation, MOP leverages a leader protocol to synchronize and update local dictionary views of its writers. For a MOP that has 90% of the keys ordered, we are able to reduce query filtering latency by up to 47% compared to decoding a standard dictionary, and 9% slower than a order preserving dictionary according to our experiments. In addition we propose a nested MOP, *Cascade-MOP or C-MOP*, that minimizes the amount of disordered data at the cost of more complex query evaluation.

To evaluate the effectiveness of MOP, we implement a prototype within the open-source columnar format Parquet [12]. With this prototype we consider how to construct a MOP without knowing the value domain a priori and demonstrate how to leverage MOP to accelerate range queries and sort operators. The rest of the paper is organized as follows. Section 4.2 discusses related work. Section 4.3 overviews MOP and key definitions. Sections 3.6 and 3.7.2 cover MOP generation and query execution respectively. Section 3.7.3 evaluates MOP for both range filtering and generation. The paper concludes in Section 3.10.

While there is a large body of research on compression and encoding for columnar databases, we mainly focus on dictionary encoding and its query optimizing on encoded data. In this section, we review previous efforts on this topic. We first introduce dictionary encoding and review the state-of-the-art, then briefly discuss consensus protocols which our dictionary encoding is based on in a distributed environment.

## 3.2 Dictionary encoding

Dictionary encoding creates a bijective mapping between values of variable length to compact integer codes, and replaces the original data entries with corresponding codes. The dictionary used in the encoding process is then attached to the encoded data, as part of the encoding output. Dictionary encodings are information-lossless as described by Lempel and Ziv, and

such encodings can achieve the theoretical lower bound of compression ratio defined by Shannon entropy [55, 95]. A dictionary can be global or local. For global dictionaries, a single dictionary is used for the entire column; for local dictionaries multiple dictionaries exist, and each are valid only for some partition of the data (i.e. block, page, or row-group). Dictionaries may be random such that keys are assigned with no order, or may be order-preserving such that keys preserve an order of the values (e.g. for mappings $k_1 = v_1$ and $k_2 = v_2$, $k_1 < k_2$ iff $v_1 < v_2$). Note that while order-preserving dictionaries can be local or global, they primarily are used for global dictionaries for query performance benefits [13].

Compression techniques help reduce I/O operations and data storage, and therefore are prevalent for analytic systems. Many byte-oriented or block-oriented compression techniques require data to be decompressed before querying [43]. These compression techniques, such as GZip and Lempel-Ziv [96], typically require expensive CPU cycles. Columnar encoding schemes, such as run-length encoding or dictionary encoding, trade-off CPU overhead for larger storage and the ability to directly query on the encoded data [9].

Research has explored optimizing dictionary encoding for database systems. Chen et al. [21] proposes a hierarchical dictionary encoding scheme for string attributes, supporting encoding at different granularities (attribute level, word level, prefix level, etc.). Paradies et al. [74] demonstrates an entropy-based approach that is adaptive to user query patterns. Column-oriented databases, such as MonetDB [98] and C-Store [86], use dictionary encoding to allow arbitrary types to be converted to consecutive integer codes; this enabling other encoding schemes, such as bit-packing (i.e. truncating unnecessary bits), run-length encoding, and delta encoding to be applied, further reducing storage size.

In addition to the encoded data storage, the dictionary itself can have substantial contribution to storage space. Muller et al. [65] make a thorough comparison between various dictionary compression algorithms regarding decoding speed and space consumption, and propose an empirical decision tree based approach to select a dictionary algorithm that is either optimized for access speed or storage space on a given dataset. Zukowski et al. [99]

propose PDICT compression scheme that allows infrequent values to be exceptions from the dictionary in order to reduce dictionary size on skewed frequency distributions, thus better compression performance.

In addition to compression, research explores how dictionary encodings affect query performance. Chen et al. [21] design a compression-aware optimizer to estimate overhead brought by accessing compressed data. Ray et al. [79] and Abadi et al. [9] show that by rewriting query predicates, one can efficiently skip the decoding step and execute queries directly on encoded data, which is beneficial to query performance due to reduced I/O requests and leverage in MOP. Jiang et al. [44] demonstrates how a SIMD-based algorithm can filter up to 18 billions entries per second on dictionary encoded data that is bit-packed.

While standard dictionary encodings only support querying directly on encoded data for equality predicates, an order-preserving dictionary [10] can support direct evaluation for range queries. For non-supported queries, either the encoded values must be decoded using the dictionary and evaluated against the predicates, or the predicates must be evaluated against the entire dictionary and passing keys are recorded. Antoshenkov et al. [11] propose an order-preserving data structure for DBMS and Binnig et al. [13] adapt the idea for in-memory analytic databases. Order preserving dictionaries require that the entire value space is known before encoding the dataset.

Dictionaries can be implemented using various data structures, such as an array, hash table, or trie [31] and different compression strategies, such as Huffman coding, Hu-Tucker coding [41], front coding [92], and RE-PAIR [54]. These compression strategies have full access to entire value corpus and thus can achieve better compression ratio, they all suffer from higher latency and low throughput [47], rendering them unsuitable for many database applications. In this paper, we focus our discussion on hash table-based dictionary with no compression strategy applied on the dictionary.

## 3.3 Consensus Protocol

When multiple distributed, shared-nothing nodes participate in the dictionary generation process in parallel, consensus protocols need to be employed to ensure a global unique key-value mapping. Consensus protocols are a family of algorithms for distributed systems that makes sure when multiple write attempts for a key occur, only one value is chosen for all participating nodes.

Many previous systems employ consensus protocols that provide eventual consistency to improve system throughput. Wuu et al. [93] presents using distributed logs to maintain unified dictionary views on each node. Amazon's Dynamo [24] designs a key-value store that employs a quorum and version based eventual-consistency algorithm to resolve value-conflict between nodes. These are closely related to our distributed-dictionary generation problem. However, in our scenario, each node encodes data stream and emits the results to output devices in real-time. Nodes only maintain local data copies of limited size, and are not allowed to re-encode data that has been written out. Eventual consistency does not fit in this case as there's no time bound on when a change will be propagated to nodes, which may cause incorrect data to be emitted. In this paper, we limit our discussion to consensus protocols that are able to provide a stronger consistency guarantee. We focus on protocols with a distinguished leader, such as GFS [34], Raft [66], and RAMCloud [67].

**Protocols with Distinguished Leader**

In this type of protocol, a unique node is designated([34, 67]) or elected([66]) to be leader, and all other nodes serve as followers. Followers who want to make write operations send requests to leader, who computes a final image based on the requests and broadcasts the results to all followers.

The distinguished leader protocol approach is easy to understand and simpler to implement [66]. In addition, the leader always maintains an up-to-date image of global dictionary, allowing easy access to the information. However, a single leader architecture is vulnerable to single node failure and requires a complicated process to recover. In addition, the leader

node can easily become performance bottleneck under high throughput.

In this paper, we implement distinguished leader protocol and for distributed dictionary generation.

## 3.4 Background

In this section, we review previous efforts on this topic. We first introduce dictionary encoding and review the state-of-the-art, then briefly discuss consensus protocols on which our global dictionary encoding is based.

### *3.4.1 Dictionary Encoding*

Dictionary encoding creates a bijective mapping (the dictionary) between input values of variable length to compact integer codes, and replaces the original data entries with corresponding codes. Dictionary encoding belongs to the information-lossless of finite order (ILF) encoder family described by Lempel and Ziv, and it is shown that such encoding can achieve the theoretical lower bound of compression ratio defined by Shannon entropy [55, 95].

As analytic database systems often involve heavy I/O operations, compression techniques help reduce data size, alleviating I/O bottlenecks. However, compression may require data to be decompressed on-the-fly upon access, and most general purpose compression algorithms, such as Lempel-Ziv [96], GZip [35], and Snappy [36], suffer from high CPU overhead. On the other hand, lightweight encoding schemes, such as dictionary encoding, provide a well-balanced trade-off between storage size reduction and CPU overhead [43].

Various algorithms have been designed to optimize dictionary encoding for database systems. Chen et al. [21] proposes a hierarchical dictionary encoding scheme for string attributes, supporting encoding on different levels (attribute level, word level, prefix level, etc.). Paradies et al. [74] demonstrates an entropy-based approach that is adaptive to user query patterns. Moreover, in column-oriented databases, such as MonetDB [98] and C-

Store [86], dictionary encodings allows arbitrary types to be converted to consecutive integer codes, enabling other encoding schemes, such as bit-packing (i.e. truncating unnecessary bits), run-length encoding, and delta encoding to be applied, further reducing storage size.

In large datasets, besides storage size used by encoded data, the storage size occupied by dictionary itself is non—negligible. Studies have been done on reducing dictionary size. Muller et al. [65] conduct a thorough study on the effect of various compression techniques and propose a decision tree based approach to select proper compression on a given dataset.

Aside from space benefits, research explores how dictionary encodings affects query performance. Chen et al. [21] design a compression-aware optimizer to estimate overhead brought by accessing compressed data. Ray et al. [79] and Abadi et al. [9] show that by rewriting query predicates, one can efficiently skip the decoding step and execute queries directly on encoded data, which is beneficial to query performance due to reduced I/O requests. While a trivial dictionary only supports query rewrite on equality predicates, with an order-preserving dictionary [10], one can further expand rewriting to range predicates. Antoshenkov et al. [11] proposes an order-preserving data structure for DBMS and Binnig et al. [13] adapt the idea to in-memory architecture. Hardware based algorithms are also invented to further speed up query performance on dictionary encoded data. Jiang et al. [44] demonstrates an innovative SIMD-based algorithm that is able to filter up to 18 billions entries per second on dictionary encoded data.

Dictionaries can be implemented using various data structures, such as an array, hash table, or trie [31]. It can also choose from different compression strategies, such as Huffman coding, Hu-Tucker coding [41], front coding [92], and RE-PAIR [54]. Although these compression strategies have full access to entire message corpus, and thus can achieve better compression ratio, they all suffer from higher latency and low throughput [47], rendering them unsuitable for database application scenario. In this paper, we focus our discussion on hash table-based dictionary with no compression strategy.

### *3.4.2 Consensus Protocol*

When multiple distributed, shared-nothing nodes participate in the dictionary generation process in parallel, consensus protocols need to be employed to ensure a global unique key-value mapping. Consensus protocols are a family of algorithms for distributed systems that makes sure when multiple write attempts for a key occur, only one value is chosen for all participating nodes.

Many previous systems employ consensus protocols that provide eventual consistency to improve system throughput. Wuu et al. [93] presents using distributed logs to maintain unified dictionary views on each node. Amazon's Dynamo [24] designs a key-value store that employs a quorum and version based eventual-consistency algorithm to resolve value-conflict between nodes. These are closely related to our distributed-dictionary generation problem. However, in our scenario, each node encodes data stream and emits the results to output devices in real-time. Nodes only maintain local data copies of limited size, and are not allowed to re-encode data that has been written out. Eventual consistency does not fit in this case as there's no time bound on when a change will be propagated to nodes, which may cause incorrect data to be emitted. In this paper, we limit our discussion to consensus protocols that are able to provide a stronger consistency guarantee. We focus on two families of consensus protocols: protocols with a distinguished leader, such as GFS [34], Raft [66], and RAMCloud [67], and decentralized protocols, such as Paxos [53].

**Protocols with Distinguished Leader** In this type of protocol, a unique node is designated([34, 67]) or elected([66]) to be leader, and all other nodes serve as followers. Followers who want to make write operations send requests to leader, who computes a final image based on the requests and broadcasts the results to all followers.

The distinguished leader protocol approach is easy to understand and simpler to implement [66]. In addition, the leader always maintains an up-to-date image of global dictionary, allowing easy access to the information. However, a single leader architecture is vulnerable to single node failure and requires a complicated process to recover. In addition, the leader

node can easily become performance bottleneck under high throughput.

**Decentralized Protocols** Two-phase commit (2PC) and Paxos are well-known decentralized protocols. These protocols allow any node in the system to propose a change. The nodes then vote to approve or reject the change. A unanimous agreement (2PC) or majority agreement (Paxos) is needed for the change to be approved. A decentralized protocol is more robust against single node failure, and allows the throughput to be evenly distributed among nodes in the cluster. But it may generate more network traffic when voting to reach an agreement. It can also suffer from competition of requests from different nodes and thus fail to make progress when trying to reach consensus. Bully algorithm [33] alleviates the blocking issue by always allowing the nodes with higher ID to move on.

In this paper, we implement both the distinguished leader protocol and decentralized protocol 2PC with bullying for distributed dictionary generation.

## 3.5 MOP Overview

Here we introduce the MOP organization, key space layout, necessary data structures, and a natural extension of MOP.

### *3.5.1 MOP Definitions*

A Mostly Order Preserving Dictionary (MOP) is a bijective mapping of keys $K$ and values $V$ consisting of two sections. As shown in Figure 3.2 the first section is order preserving and the second is not. Key space is pre-allocated for the ordered section and the disordered section grows as needed, such that after initialization the MOP specifies at most $m$ keys are ordered, while the dictionary can grow to $n$ keys (with $n \geq m$).

The initialization phase looks at a sample (or head) of records from the dataset to be loaded by collecting statistics and a set of values ($B$) to bootstrap the ordered section. These values are distributed evenly throughout the ordered section. Our *allocation strategy*

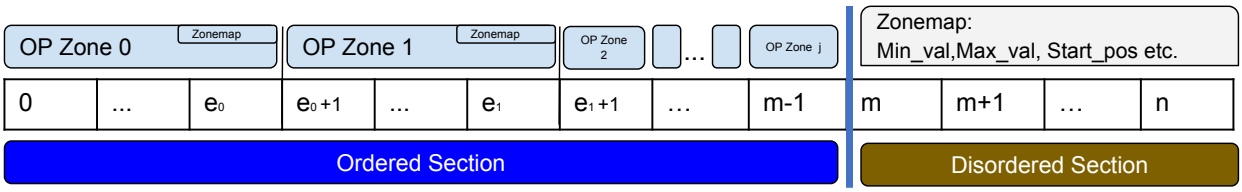



Figure 3.2: Key space for MOP dictionary

determines how much space to use for the ordered section and how keys should be spaced in the section. During the generation phase, when inserting a value $v \in V$ into the MOP to get its corresponding key $k$, we distribute the incoming items into the ordered section. When code space conflicts happen (e.g., cannot insert the value without violating ordering), the unsettled items will be appended to the end of code space in the disordered section sequentially (e.g., $v$'s key $k$ will be $m < k \leq n$). We refer to this as a *spillover*. Given a MOP dictionary, *ordered ratio* $r$ is defined to indicate the proportion of ordered entries in the dictionary:

$$r = 1 - \frac{n - m + 1}{|V|}$$

A *padding ratio* $p$ is defined to indicate the proportion of empty keys in the ordered space:

$$p = \frac{n - |V|}{n}$$

We use $r$ and $p$ for query and storage evaluation respectively. Therefore, the goal of MOP is to maximize the ordered ratio $r$ as much as possible, while trying to avoid excessive "bloat" of the key space by minimizing $p$. In Section 3.6 we discuss how MOP allocates keyspace and assigns keys. Note that unless bit-packing is specified prior to encoding, $n$ is not fixed and can grow. If bit-packing is used $n$ is fixed and the dictionary can only hold $2^n$ keys, and if additional keys are needed the dictionary must be entirely recoded (e.g. regenerate). For the rest of the paper we focus on the case where $n$ is dynamic, unless specified.

Given that it is impossible to guarantee that the ordered ratio of a MOP = 1, there will be some records that cannot fit into the ordered section. A natural extension to explore

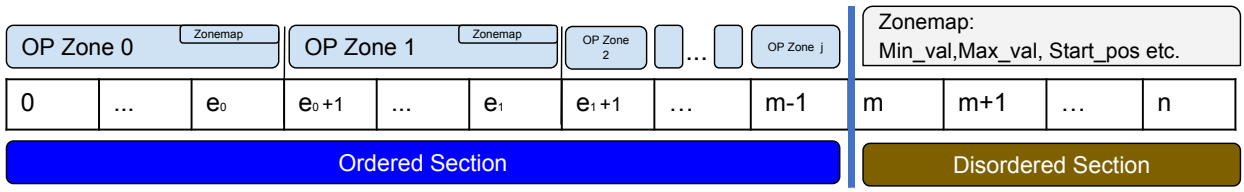



Figure 3.3: Key space for C-MOP dictionary

is instead of a single ordered section, is cascading ordered sections before defaulting to a disordered section.

### *3.5.2 Cascade-MOP*

With a *Cascade-MOP (C-MOP)*, we nest multiple levels of order-preserving sections before a single disordered-section. By default we limit nesting to eight levels, but this is configurable. Here, when spilling a value to the disordered section we instead treat it as a new order-preserving section. Rather than appending to the end of the key space, we allocate a new nested ordered key space for those unsettled items. For this new zone we allow for refinement in our allocation strategy, in particular how much space to allocate and how to bias the initial placement of spilled values. We call each order-preserving section an OP zone. Figure 3.3 shows the layout of a C-MOP with $k$ levels. The first zone runs from keys (0,$e_0$), the next from ($e_0 + 1$, $e_1$), and so on until the number of OP zones grows until it reaches the user-defined limit (8 by default). A final disordered section exists after the OP zones. In Section 3.6.2 we discuss determining the number of cascading levels.

### *3.5.3 Dictionary Generation Assumptions*

For this paper we assume there is a single coordinator (thread, process, or node) that is responsible for assigning keys to values. There are $n$ workers (threads, processes, or nodes) who are each reading a partition of a data file that contains records to write to a database. For attributes that are encoded using a dictionary, each worker maintains a local copy of the dictionary. When a value is to be encoded it first checks the local dictionary, and if

the value is not found it requests the key from the coordinator by providing the value. For performance reasons, the worker may batch requesting values together and the coordinator may batch requests from workers together – this is respectively referred to as worker batch size and coordinator batch size. When waiting for keys the workers block writing the current data records, and the leader broadcasts new keys to all workers, which is subsequently added to their local dictionaries when processed. Once the key for the value is found the worker writes out the encoded key for the record. We assume that once a worker writes an encoded value it cannot be changed, unless the process restarts (e.g. recoding). While we discuss the process for one attribute encoding for a data load, the process works for multiple attributes with separate metadata and attribute identifiers in the request.

## 3.6 Methodology

In this section we discuss the steps of MOP and C-MOP generation. This is broken into three main steps: initialization, encoding, and finalization. For C-MOP, we explain how spillover differs. For exposition we assume there is a coordinator managing the dictionary and multiple workers encoding values and requesting keys from the coordinator. We assume a shared-nothing architecture with workers starting with a distinct partition of the input dataset. Other approaches, such as threads with locks, are valid with minimal changes.

### *3.6.1 Initialization*

When generating a new dictionary all workers notify a coordinator about the attribute to encode, which may come from a file containing multiple attributes. This message includes the worker's coarse estimation about the data to encode (i.e. file size). The leader responds by requesting from each worker a sample of the dictionary to bootstrap the MOP. We refer to the percentage of records as *lookahead* ( i.e. a lookahead of 0.10 is a 10% a sample). The worker preprocesses the sample of records before sending out a sample of the dictionary,

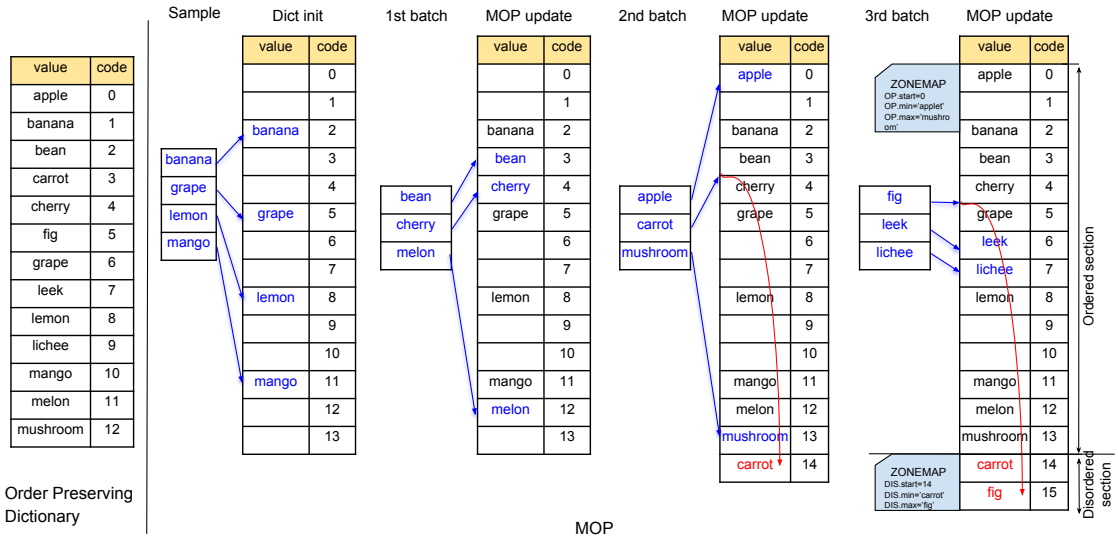



Figure 3.4: MOP generation example

which includes a set of records and several features (i.e. sortedness, estimated cardinality). These features help the coordinator understand the values to encode. By default, workers use the head of the file to sample unless otherwise specified. In rare cases, the leader will force workers to redo sampling using a uniform sampling strategy. This occurs if the features collected from workers indicate a high probability the attribute is sorted. To determine sortedness of an attribute, we use Kendall's $\tau$ which generates a real number in $[-1, 1]$. An output of 1 denotes a fully sorted attribute and $-1$ denotes a fully inverse-sorted attribute.

Once the sample is collected, the number of unique values in the sample data is counted and divided by the lookahead proportion to determine the estimated cardinality of the unique values in the data. The size of the order preserving section's key space is pre-allocated based on the estimated cardinality and a slack parameter which determines slack space between keys. The sample values are then uniformly distributed over the key space as shown in Figure 3.4. It is worth noting that this process could leverage more advanced sample-based distinct value estimators [20]; however, our experiments in Section 3.9.3 show these approaches add limited value to MOP performance. Given apriori distribution information, the sample values can be biased to the distributed key space, which we discuss in Section 3.9.3.

The space left between two ordered keys allows new input unique values to be ingested if encountered later in order to retain the overall ordering. We calculate this slack space

by dividing a controlled parameter *pitch* by the lookahead proportion. By adjusting pitch, the slack space can be scaled to allow for more or less key space available for new unique value batches. A higher pitch gives a larger initial dictionary with more space between the bootstrapped values. Too much slack may result in a dictionary to be sparse or padded (e.g. many empty cells), and too little slack results in the inability to order keys properly. As we later show, C-MOPs can achieve a high ordered ratio with a pitch of 1, and single layer MOPs can benefit from a pitch of 3, but this increases the key space. Once the initial bootstrapped dictionary is generated, the coordinator sends it to all workers, and the encoding process begins.

### *3.6.2 MOP Encoding*

In the encoding stage, the remaining unique values get inserted into the MOP in groups of values, or batches. Each worker can batch a configured number of values (worker batch) then send them to the coordinator, which in turn can batch a set of requests (coordinator batch) to insert into the MOP. In addition to amortizing message overhead, batching multiple values for insertion can help with proper spacing of values and reduce spillover.

For every value in each batch, if there is space in the order preserving section, the value is inserted there uniformly between the two already inserted ordered values that the new value's ordering falls between. If multiple values in this batch fall in the same range, they will be evenly spaced. Figure 3.4 shows a sample key insertion for the first batch values.

If no space is left for inserting a value in the ordered section, the value will be spillover either to a disordered section for MOPs or to a cascading ordered section for C-MOPs.

#### MOP Spillover

With MOP, when values spillover they are added sequentially to the end of the dictionary in the disordered section. For example, in Figure 3.4 in the second batch, no space exists for *carrot* between *bean* and *cherry*. The *carrot* value then gets appended to the end of the

ordered key space in the disordered section. This batching process repeats until all data has been processed.

### C-MOP Spillover

The C-MOP encoding process works in a similar fashion, with the difference being when there is no space left in an order preserving section for inserting new values, the spillover(s) may create another order preserving zone. Here, the process will add any pending requests to the current set of values to encode. All spilled-over values will be used to seed the next OP zone. Spillover can result in creating a final disordered section if we believe the encoding is nearing completion or we hit a prescribed maximum number of OP zones, which is the strategy we use.

When creating the next OP zone, we determine how large to make the zone based on the prior zone's size multiplied by a *MOP zone ratio*, which is set between 0.20 and 2.0. To determine the ratio we examine the estimated cardinality for the prior zone (scaled by estimated progress of the dataset), and the observed cardinality for the prior zone(s). If the observed cardinality is less than the estimated cardinality, we set the MOP zone ratio to 0.2 with the expectation that the spillover is due to the distribution being slightly deviated. If the observed cardinality is much higher than the estimated cardinality, we set the ratio closer to 2.0 as we assume our estimates were way off or that the distribution in the dataset has changed and we should account for larger capacity to avoid introducing an additional OP zone.

We explore biasing the placement of values into new OP zones using a histogram of the prior zone(s). Here, we create a histogram of values bucketed on the original sample values. We then look at all values currently inserted into the dictionary and count the number of values that either landed between the sample values or would have landed between the sample values if space permitted. Next, we partition the new OP zone into sections whose sizes are scaled by the histogram data. Values inserted into these OP zones will be distributed evenly

within their respective sections. As we later demonstrate, this approach works well under certain use cases, but we opt for the same MOP allocation strategy of uniformly placing values in the key space for simplicity.

### *3.6.3 Finalization*

Once all values are encoded, the process is finalized. For each OP zone and the disordered section we create a zone map of min and max values for query evaluation and record the starting position of this zone (i.e. the first key). Figures 3.2 and 3.3 respectively show the final organization for MOP and C-MOP. The generated dictionary is written to the output file.

## 3.7 MOP Queries

In this section, we describe how MOP/C-MOP supports query operator execution efficiently. MOP is optimized for equality predicates, range predicates, and sorting. These operators can work directly on encoded data with little or no decoding overhead. For other operators, data needs to be decoded and materialized, just as when using a normal dictionary. The goal of our query evaluation is to rewrite any query $q$ that filters or sorts on a MOP encoded attribute $x$, such that it minimizes decode operations and evaluates as much of the query directly on the encoded keys, as opposed to evaluating predicates against decoded values.

### *3.7.1 Equality Operator*

An equality operator ($x = v_a$, $x \neq v_a$) can be performed efficiently on MOP just as on a normal dictionary. When taking $x = v_a$, for example, we first check if there exists a key $k$ in MOP satisfying $MOP(k) = v_a$. A miss means no value in the column matches the predicate and the execution can return prematurely. Otherwise, we scan the encoded entries $k_i$, looking for $k_i = k$. This operation skips decoding operations on the encoded entries and

thus saves CPU overhead.

### *3.7.2 Range Operator*

Without loss of generality, we discuss in this section how to execute an inclusive range operator ($x \in (v_a, v_b)$) on a MOP encoded column; the approach can be generalized to a compare operator, exclusive ranges, and `like` predicates with constant prefixes, as well as conjunctive and disjunctive combinations of these operators.

MOP and C-MOP can both be viewed as a combination of multiple order-preserving dictionaries (OP) and an unordered dictionary (DIS), where MOP contains only one OP, and C-MOP can have multiple OP sections. We first discuss the execution of a range operator on OP and DIS separately, then show how to combine the results to obtain the final result. An inclusive range operator for an OP can be easily extracted from a query $q$ via query rewriting. To evaluate $x \in (v_a, v_b)$, we first find the corresponding key range $k_a = \min(\{k|OP(k) \geq v_a\})$ and $k_b = \max(\{k|OP(k) \leq v_b\})$, then perform a scan on the encoded column, looking for entries satisfying $k \in (k_a, k_b)$. This process involves no decoding operations.

When executing a range operator on DIS, we compare the zone map $(v_{min}, v_{max})$ of DIS against the query range and consider three cases.

Type 1 $(v_{min}, v_{max}) \cap (v_a, v_b) = \varnothing$. In this case, DIS does not contain any value within the query range, and can be safely skipped for the query.

Type 2 Here, the query range and zone map overlap. In this case, for the keys in DIS we perform a decode operation and compare the decoded result against the range. Let $k_s$ be the starting key in DIS. For each encoded entry $k$, we check $k \geq k_s$ to make sure the key belongs to DIS, and, if so, we decode $v = DIS(k)$ and check if $v \in (v_a, v_b)$. This operation only involves decoding keys belonging to DIS, which is relatively small compared to decoding the entire column.

Type 3 $(v_{min}, v_{max}) \subseteq (v_a, v_b)$. In this case, all keys in DIS are included in the query range. Let $k_s$ be the starting key in DIS; we can rewrite the range query to be $k \geq k_s$ and execute it on the encoded column. This involves no decoding operations.

As a MOP contains two sections, OP and DIS, with disjoint key ranges, executing a range operator on MOP is equivalent to first applying the operator to OP and DIS separately then performing a disjunction of the results. As an example, we consider a query $x \in (apple, cherry)$ on the MOP shown in 3.4. The operator on OP is rewritten as $k \in (0, 4)$. As "cherry" is within the zone map of DIS, we also need to perform decoding for keys belonging to DIS, which yields $k \geq 14 \wedge decode(k) \in (apple, cherry)$. The result, as the disjunction of the two parts, is then

$$k \in (0, 4) \vee (k \geq 14 \wedge decode(k) \in (apple, cherry))$$

Similarly, query $x \in (apple, mango)$ will be translated to $k \in (0, 11)$ on OP, and, as "mango" is larger than $v_{max}$ of DIS, it is translated on DIS as $k \geq 14$, and the result is

$$k \in (0, 11) \vee k \geq 14$$

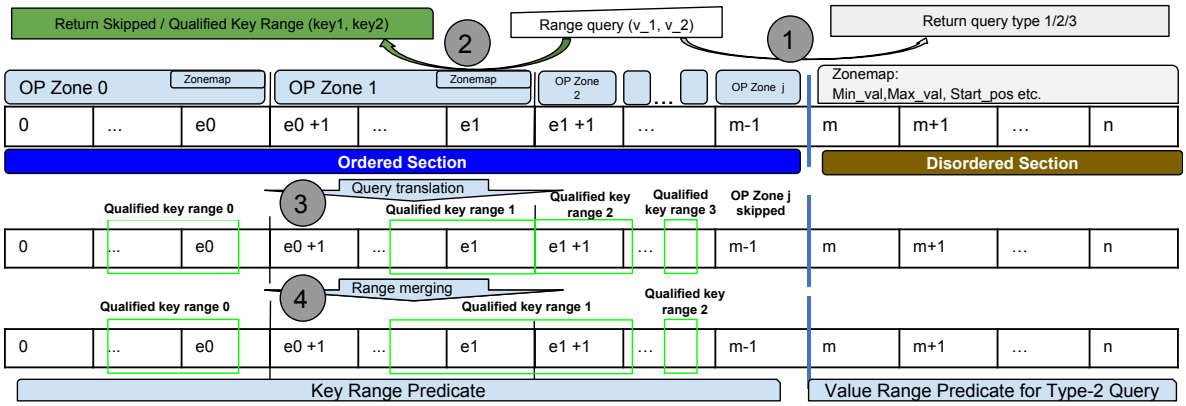



Figure 3.5: Steps for range filtering with C-MOP

C-MOP is comprised of multiple OP sections and a DIS section, with their key ranges

non-overlapping, and thus can be processed in a similar manner as described above in MOP. More precisely, we first execute the operator on each section independently and perform a disjunction of the result. To further optimize query evaluation, we utilize a zone map for each ordered section, or zone, to minimize unnecessary zone look ups. To further optimize the result, we also perform range merging. If the final result contains range clauses that are adjacent to each other, e.g., $k \in (k_1, k_2] \vee k \in (k_2, k_3)$, they will be merged together as a larger range $k \in (k_1, k_3)$. This step reduces the number of ranges to be evaluated and potentially improves efficiency. This is shown in 3.5. Evaluating a range operator for a C-MOP works as follows:

**Check the zone map for the disordered section to decide the query type:** As with a MOP, step (1) is to check the type of query to determine if the disordered section is needed.

**Check the zone map for each OP zone to see if skipping is possible:** For each OP zone, step (2) checks its zone map before getting the appropriate keys in that zone. If the query range is disjoint with zone map, this zone is skipped.

**Range query translation per OP zone:** If skipping a zone is not feasible, for step (3) the range query is translated into the qualified key range for this OP zone (i.e. $k_a = \min(\{k|OP(k) \geq v_a\})$ and $k_b = \max(\{k|OP(k) \leq v_b\})$). After query translation, each OP zone outputs either skipped or a qualified key range that is added a disjunctive predicate.

**Qualified range merging:** A large number of qualified ranges adds more integer operations for verification on each record. In order to eliminate unnecessary compare operations, we merge key ranges into a larger one if they are adjacent in step (4).

### *3.7.3 Sort Operator*

A sort operator for a MOP shares similar logic with range filtering, by leveraging the orderedness of the MOP dictionary. To avoid decoding the disordered section, we temporarily expand the key space to sort disordered keys in relation to the sorted section. The MOP sort operator pre-scans the dictionary and temporarily assigns dictionary entries in the dis-

ordered section with a float key that follows the orderedness in the ordered section and new float keys. We then build a mapping from the old integer code to the new float code for entries in the disordered section. We apply a sorting algorithm on encoded integer values and translated float values, without decoding any values.

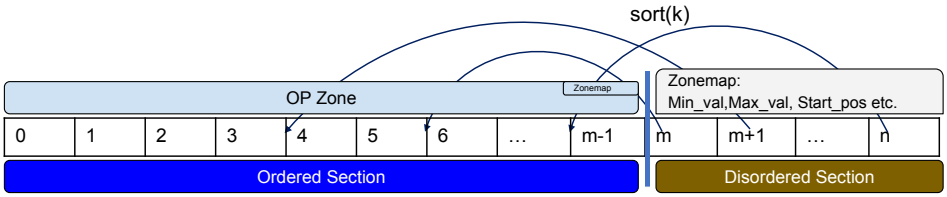



Figure 3.6: Sort with MOP

To perform sort operation on a MOP encoded column, we create a bijection $sort : \mathcal{N} \to \mathcal{R}$ which maps all keys in $MOP(OP, DIS)$ to a real number space, satisfying that

1. For $(k, v) \in MOP$, the dictionary $(sort(k), v)$ is fully order-preserving.

2. For $k \in OP, sort(k) = k$.

With property 1, we can apply $sort$ to the encoded entries $\{k_i\}$, and do a merge sort on the sequence $\{sort(k_i)\}$. In addition, by property 2, the dictionary $sort$ uses contains much less entries than MOP (it only contains keys in $DIS$), making the sorting operation more efficient than first decoding the entries and then perform sorting.

Now we explain how to define the bijection $sort$ for $MOP(OP, DIS)$. We construct a sorted list $L_{sort}$, and a dictionary $D_{key}$. For each entry $(k_o, v_o) \in OP$, we insert $v_o$ into $L_{sort}$. Then for each entry $(k_d, v_d)$ in $DIS$, we perform a search in $L_{sort}$ to find two adjacent entries $(v_1, k_1)$ and $(v_2, k_2)$ satisfying $v_1 < v_d < v_2$, insert $v_d$ into $L_{sort}$, and insert $(k_d, \frac{k_1+k_2}{2})$ into $D_{key}$. The $sort$ bijection is then defined as

$$sort(k) = \begin{cases} k & k \in OP \\ D_{key}(k) & k \notin OP \end{cases}$$

The goal of our experimental evaluation is twofold. The first section evaluates MOP's and C-MOP's ability to improve query performance for range filtering and sorting. We compare these against order preserving and dictionaries that require decoding (e.g. non-order preserving). We evaluate across varied ordered ratios and selectivity ratios with a synthetic and real-world dataset. The second section evaluates MOP's and C-MOP's ability to generate highly ordered dictionaries and to understand what are the critical factors in doing so.

## 3.8 Experimental Setup

All experiments were performed on servers with 2 Intel(R) Xeon(R) CPUs E5-2670 v3 @ 2.30GHz, 128GB memory, 250GB HDD, Gigabit Ethernet, and Ubuntu 14.04. Experiments use a real-world dataset of `taxi` rides from New York City [7], the lineitem table from `TPC-H`, and two synthetic datasets derived from an English word dictionary based on `uniform` and `zipf` distributions. Unless otherwise stated we use scale factor 30 for TPC-H and 15 million rides from the taxi dataset. The default MOP configuration has a lookahead of 0.1, pitch size of 1, and a worker batch size of 20. The default C-MOP layer ratio is 0.2.

For these experiments we use the Parquet file format as our query performance testbed. Parquet is an open-source column-oriented file format for distributed analytic frameworks, such as Spark and Impala [94, 52]. It provides efficient data compression and encoding schemes with the ability to handle complex nested data types (e.g. lists and maps) [62]. In the Parquet file format, values from each column are logically organized to be adjacent and physically stored in contiguous memory locations for improved compression and query I/O. Parquet supports distributed writers by storing metadata at the end of the file. As Parquet only has native support for local dictionary encoding, we add a global dictionary encoding feature into Parquet. We write a stand-alone query engine in Scala that filters and joins on attributes with equality and range predicates. Depending on the query and encoding predicate evaluation can occur directly on the dictionary key (i.e. query rewriting) or by

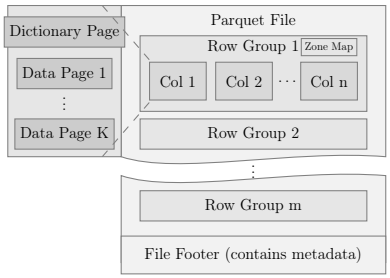


Figure 3.7: Parquet Columnar Store Format

decoding the encoded keys.

As shown in Figure 3.7, a Parquet file is made up of several row groups, which are indexed by block metadata saved in a file footer. A row group consists of several column chunks with metadata also in the file footer. In each column chunk, there are data pages and a dictionary page if dictionary encoding enabled. The dictionary is stored in a dictionary page per column chunk. Columns are aligned in row group level, which means all data for a given row is organized in the same row group. In the file footer metadata is organized about column chunk metadata, zone maps (e.g. min, max and number of nulls), encoding, and compression information. In order to evaluate query performance of both local dictionary and global dictionary with the same system, we implement global and OP dictionaries in Parquet.

## 3.9 Evaluation

We firstly show query performance on MOPs encoded datasets to prove their abilities of improving query performance for range filtering and sorting operator. Then we show MOPs generation experiment results to understand the critical factors of generating MOPs.

### *3.9.1 Range Filtering Evaluation*

In this section, we encode the given dataset under different MOP or C-MOP configurations in order to generate various ordered ratios. Order preserving and standard dictionary encoded datasets are generated for comparison. All encoded datasets are stored in Parquet's file

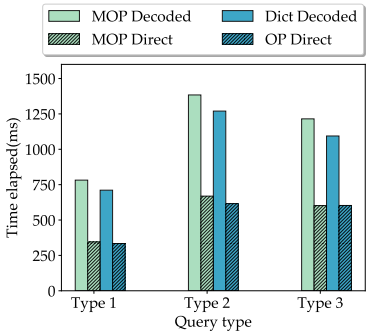



Figure 3.8: Range filter overview on taxi dataset

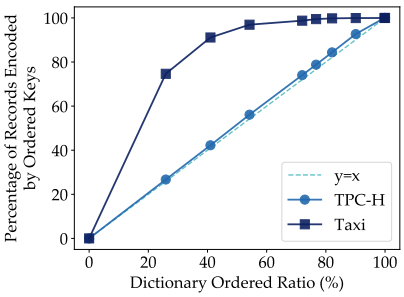



Figure 3.9: Percent of records encoded by ordered keys.

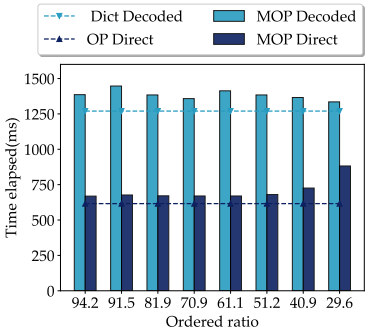



Figure 3.10: Type-2 filter on taxi dataset

format.

Using a stand-alone Java query execution framework, we evaluate the three types of range queries discussed in Section 3.7.2 by decoding dictionary keys (decoded) and directly evaluating queries on keys via query rewriting (direct). In the following experiments, we use *MOP Decoded* for cases where we use the MOP organization but fully decode every value for checking the predicate, and we use *MOP Direct* to indicate best-effort filtering on encoded keys when possible (i.e. types 1, 2, and 3). *Dict Decoded* represents filtering on a dense dictionary (e.g. no padding) when decoding every value for a predicate evaluation. *OP Direct* indicates a dense sorted dictionary with predicate evaluation without any decoding on an OP encoded dataset. For all filtering evaluation, we do not materialize the output values and instead evaluate as a filter for late materialization that creates a bitmap indicating the records that satisfy the predicates [86]. Unless otherwise stated, we run each query 15 times and report the average running time.

## Taxi Dataset Filtering

We evaluate query performance on the taxi ride dataset. We encode column *pickup_latitude* from the *trip_data* table with OP and MOP dictionary encoding respectively and use Parquet's default encoding for other attributes [7]. With MOP's default configuration, we can achieve a 94.2% ordering. We manually tune MOP parameters to get dictionaries with different ordered ratios.

Figure 3.8 shows the range query performance on the default MOP encoded dataset compared with the OP encoded dataset. MOP Direct performs similarly to OP Direct in terms of range filters for Type-1 and Type-3. The two counterparts have similar performance as neither decode the value during query processing. However, MOP Decoded is about 10% slower than Dict Decoded for all types of range queries as there are more entries in the MOP dictionary due to dictionary padding. For Type-2 range filters, MOP Direct is slightly slower than OP Direct but is still far more efficient than decoded filtering. We have a detailed analysis on Type-2 range queries with varying ordered proportions in following experiments.

Figure 3.9 shows the percentage of records encoded by ordered key versus MOP ordered ratio. Unlike the linear trend for TPC-H, which has a uniform distribution, the percentage of records encoded by ordered key grows rapidly on the Taxi dataset. This is typical for skewed distributions as it is more likely to capture the frequent records early. Therefore, Type-2 range queries can still perform efficiently with small MOP sorted ratios (i.e. 30%).

As shown in Figure 3.10, Type-2 direct filtering on MOP encoded datasets takes more time as the ordered ratio decreases, as more values need to be decoded. MOP Decoded performs better at the same time due to fewer entries being present in the MOP dictionary. Regardless, MOP Direct always outperforms the best decoded filter, even on datasets with low ordered ratios.

## TPC-H Dataset Filtering

Using a TPC-H dataset of scale 30, we encode the table *lineitem* into a Parquet file with different dictionary encodings for the *shipdate* column and Parquet's default encoding for other columns. We apply a range filter on the *shipdate* column with varying selectivity.

We achieve a 100% ordered ratio on the *shipdate* attribute with the MOP default space allocation strategy. Therefore, we manually adjusted our space allocation strategy to get MOP encoded files with different ordered ratios. In the following experiments, we fix slack to 5 and change the lookahead.

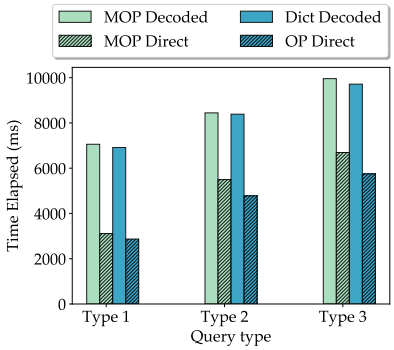



Figure 3.11: Range query on TPC-H dataset

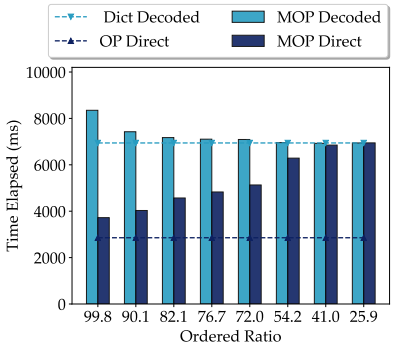



Figure 3.12: TPC-H type 2 range filter with selectivity 0.00001

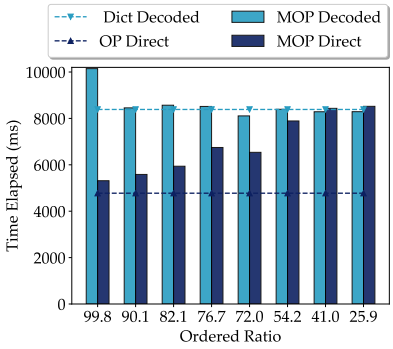



Figure 3.13: TPC-H Type-2 range filter with selectivity 0.27869

In Figure 3.11, we show the performance of three types of range queries on datasets with different dictionary encoding variations. We generate a MOP encoded file with an ordered ratio of 90.1% by manually setting the lookahead to 0.00001. These results show that direct query on MOP performs 1.5 to 2.2 times better than the regular decoded query version but 7% to 14% slower than a direct query on OP encoded datasets. The two direct queries perform quite similar to each other on Type-1 range queries, while they have a relatively greater performance difference on Type-2 and Type-3 queries. MOP direct Type-2 queries are slower as they decode records with keys from the disordered section before verifying the records. It is also slower than OP direct query on Type-3 queries as one more integer operation is needed to verify the records with disordered key. Even though disordered section decoding is not eliminated for Type-2 range queries, the MOP Direct query is still quite efficient because decoding is avoided on more than 90% of records. MOP direct query performance for Type-2 range queries varies as the proportion of records encoded by the ordered section changes, which is further dependent on the MOP ordered ratio. Again, Figure 3.9 shows the percentage of records encoded by the MOP ordered section increases uniformly as the MOP ordered ratio increases. This trend is typical for most uniformly distributed workloads as every value has relatively similar frequencies. Therefore, the MOP direct query performance on Type-2 range queries should be proportional to the MOP ordered ratio.

As we decrease the lookahead from 0.0001 to 0.000001, the MOP ordered ratio decreases

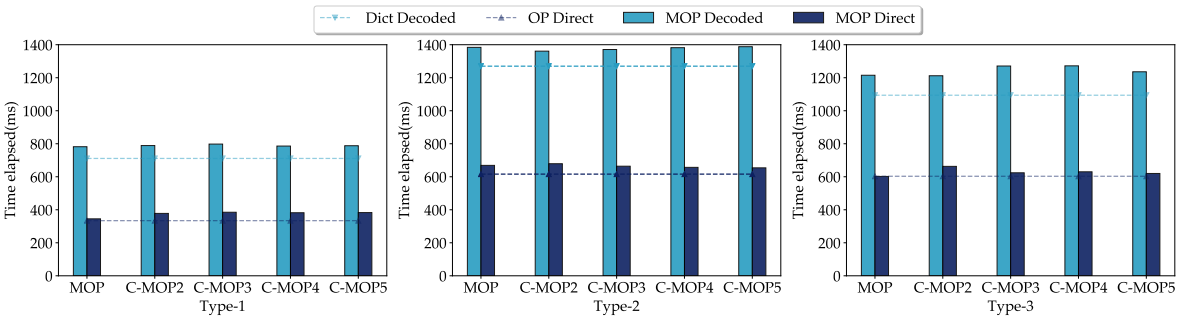



Figure 3.14: C-MOP range filter on taxi dataset with increasing the number of cascading levels.

from 99.8% to 25.9%. Figures 3.12 and 3.13 show the query performance for Type-2 queries on a dataset with different MOP ordered ratios. The dashed lines show the baseline query times for the regular decoded query and direct query on an OP encoded file, and the bars show the query time on a MOP encoded file with corresponding ordered ratios. Overall, the decoded query becomes more efficient as the ordered ratio decreases as less key space is used for the MOP dictionary, resulting in more efficient value decoding. However, it does not offset the overhead caused by an increased decoding workload in the MOP direct query. As more values need to be decoded for a direct query, the query time increases proportionally as the ordered ratio decreases. Please note that the datasets we are using are tuned to show the Type-2 query performance on different ordered ratios. However, we can achieve 100% ordered ratio using MOP default settings, where MOP's performance is almost identical to that of OP.

## C-MOP Filtering

Figure 3.14 shows range filter performance for a C-MOP with an increasing number of cascading levels for the Taxi dataset; TPC-H results are not included as it does not benefit from cascading due to its uniform distribution and low cardinality. Three sub-figures correspond to types of range filters respectively. In this experiment set, there are 1 qualified key range and 2 qualified key ranges on C-MOP datasets for queries of Type-1 and Type-3 respectively after range merging. For Type-2 queries, the number of qualified key ranges is

always equal to the number of OP Zones after range merging. With deeper cascading levels, there will be more OP Zones thus more qualified key ranges need to be checked, resulting in more integer operations for each record. However, the overhead from checking multiple qualified key ranges is alleviated by qualified range merging and offset by fewer disordered keys needing to be decoded. Therefore, the increasing cascading level shows minimal impact on filtering performance overall and in the next section we demonstrate space savings gained by cascading.

### *3.9.2 SORT Evaluation*

In this experiment, we implement a MOP sort operator based on a recursive quick sort algorithm as described in Section 3.7.3. Figure 3.15 shows the sort performance on the Taxi dataset. The bars indicate end-to-end sorting time for a MOP encoded dataset with different ordered ratios, and the dashed lines show the baseline query time for regular decoded sorting and direct sorting on an OP encoded file. Overall, MOP sorting is slightly worse than OP direct sorting, but outperforms a decoded sorting. In spite of some inevitable translation overhead for entries from the disordered section, sorting operations primarily on integers and some floats are far more efficient than that of strings. MOP sorting takes more time as the ordered ratio decreases as there is more encoded value translation needed from entries in the disordered section. However, MOP sorting is still superior to decoded, even with a relatively low ordered ratio.

### *3.9.3 MOP Generation*

In this section, we analyze the effects of changing each of the MOP generation parameters: lookahead, pitch, number of layers, and MOP layer ratio. We also look at the MOP's ability to dynamically adjust the number of keys allocated and the way values are distributed based on incoming data. We measure MOP generation performance by looking at the runtime of the generation and the ordered percentage of the resulting dictionary when using different

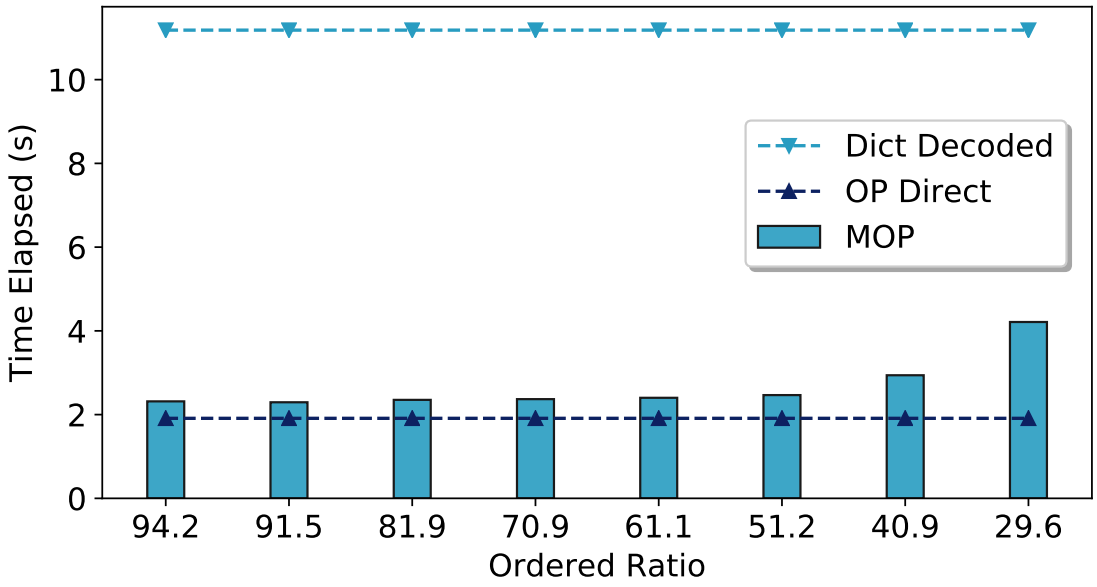
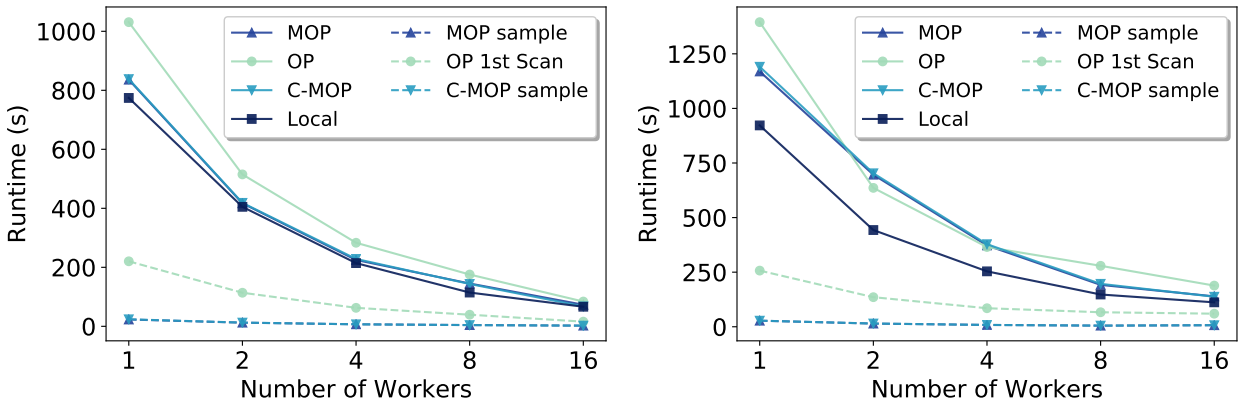



Figure 3.15: SORT Operator Runtime

datasets. Given the results of the prior section, for many workloads an ordered ratio of at least 70% gives the largest filtering performance improvement, which is trivial to achieve for most cases. As the ordered ratio nears 90%, filtering performance nears that of order preserving dictionaries. For many of these experiments, we highlight a trade-off between space and orderedness to allow the reader to understand how the impact of generation parameters. Based on these observations we believe a pitch size of 1, lookahead of 10%, worker batch size of 20, and at least 3 cascading layers with an auto layer-ratio set to 0.2 results in a good compromise of orderedness and performance. Therefore, unless otherwise stated, the experiments use this configuration. The default setup uses one server for the coordinator and one for all workers, with experiments including network latency in runtimes.

As shown in Figure 3.16, we evaluate MOP, C-MOP and OP generation performance with scaling the number of workers. We also show a local non-ordered dictionary as a performance baseline. For MOPs, we use a configuration with lookahead of 10%, worker batch size of 100. For C-MOP, we use a cascading level of 4. In Figure 3.16(a), the dictionaries are generated on the *shipdate* of the *lineitem* table with TPC-H scale *30*. In Figure 3.16(b), the dictionaries are generated on the pickup latitude of the full taxi dataset, which is roughly 30



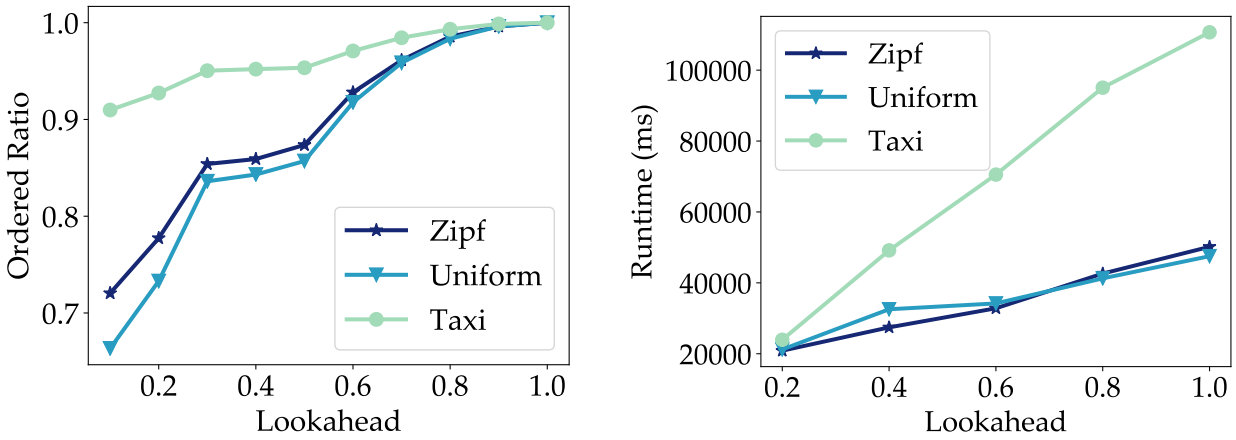

((a)) Encoding time on TPC-H ((b)) Encoding time on Taxi

Figure 3.16: Average generation time.

GB and 125,000 values spread over 173 million records. This experiment parses the input CSV file and writes out the entire Parquet file. The leader and workers are co-located on a single machine that has 16 HDDs striped via RAID 0. Note that for a single OP worker, one process reads the file, sorts the keys, and then writes the file. For multiple workers, each worker reads their segment and sends values to the leader, who after sends the updated dictionary for workers to encode with. MOP is faster than OP in most cases and preforms close to local dictionary encoding in TPC-H due to the relatively low cardinality. When the cardinality for the target column is relatively high, as with taxi, MOP has overhead for frequent coordination with the leader. MOP outperforms OP in most cases due to not learning the domain first (i.e., OP 1st scan), except for occasional cases with few workers where one worker blocks more than others when waiting for keys (i.e. 2 workers on Taxi). C-MOP performs quite close to MOP in terms of generation time as C-MOP only applies more key allocation for the spillover values, whose number is usually small.

## Lookahead

For these MOP generation experiments, we test MOP performance when changing lookahead (sampling rate of the first X% of the file). Here, all datasets use approximately 173 million records. Figure 3.17(a) shows resulting ordered percentages and speeds of generating MOPs




((a)) Ordered Percentage

((b)) Runtime (ms)

Figure 3.17: Evaluating the effect of lookahead on MOP generation performance (Runtime and Ordered Percentage).

with different data skews. Skewed datasets have more repeated values, so key allocation costs are reduced leading to lower runtimes. Skewed workloads are also more predictable, leading to higher ordered percentages.

The higher variation in the uniform distribution workloads resulted in less predictable data that also lead to lower ordered percentages. Generating a MOP using the taxi dataset also resulted in higher ordered percentages than either of the synthetic workloads. This is because MOP predicted that the cardinality of the data was larger than it actually was, resulting in extra slack space being allocated. This extra slack space allowed a higher percentage of batch values to fit in the ordered section at the cost of a more inflated key space.

In Figure 3.17(b), we also show the costs of different lookahead percentages. As expected there is a linear growth for the datasets due to sampling from the head of the file, and the Taxi dataset is slower due to parsing 14 attributes to encode a single attribute. The Zipf and Uniform workload each only has one attribute per record, reducing the CPU intensive task of parsing and validating [28]. This result shows that high lookahead percentages can come with a high cost.

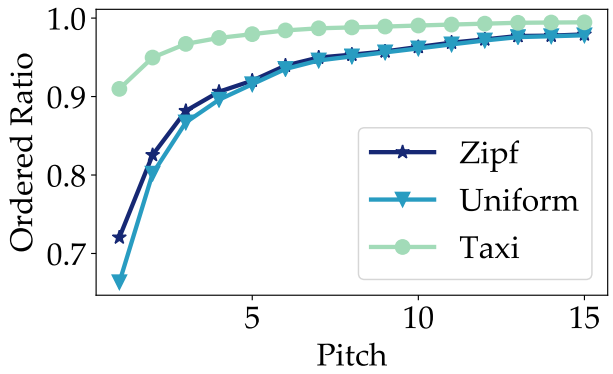



Figure 3.18: Evaluating the effect of pitch on MOP ordered percentages using uniform, zipf, and taxi workloads.

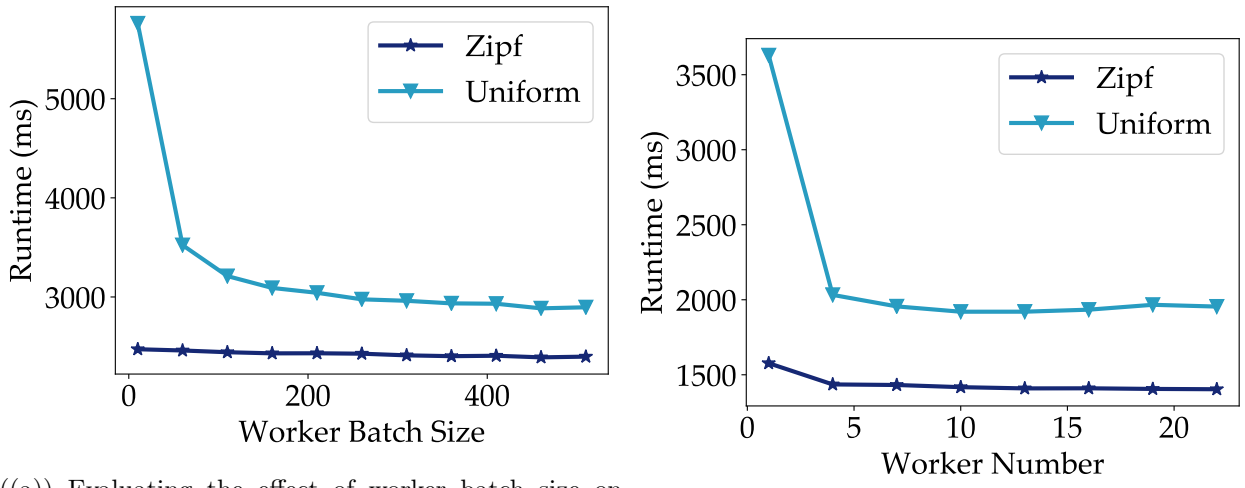
## Pitch

Here we discuss the impact of scaling the slack space between sampled values with the pitch parameter. The slack space, or the empty space left between sample values, is calculated by dividing pitch by lookahead, which for these experiments is set to 10%. In Figure 3.18, we show how adding more space in between sample values affects the MOP's ability to make ordered datasets. Overall, all workloads see similar benefits from an increase in pitch with respect to the ordered percentage.

As pitch scales the initial space between sample values, it changes the amount of room left for encoding values in the ordered section of the dictionary, which results in fewer values in the disordered section. By changing the pitch, we are able to adjust the ordered percentage of the resulting MOP, but to increase the ordered percentage we incur the costs of having a larger key space, which can result in a larger file (Sec. 3.9.4) and worse query performance [44].

## Worker Configuration

For these experiments we measure how worker batch size and the number of workers affects MOP generation. In these experiments, we do not materialize the encoded file, the leader process and worker processes are run on different machines to account for network latency, and the generation is run on files with $100,000$ values. Figure 3.19(a) shows how changing the




((a)) Evaluating the effect of worker batch size on MOP generation runtime using uniform and zipf workloads

((b)) Evaluating the effect of worker number on MOP generation runtime using uniform and zipf workloads

Figure 3.19: Worker configurations

the worker batch size affects the runtime of the MOP generation. By increasing the amount of batching being done by the workers, we increase the number of new values the worker will discover at once. As all values new to the worker need to be sent to the coordinator as a proposal, increasing the amount of items sent at once will decrease the total number of messages that need to be sent. This will in turn reduces message passing overhead and subsequently runtime. In Figure 3.19(b), we show how parallelism, through adding more workers, affects the speed at which a MOP can be generated. As expected, we see that the sharp decrease in the runtime early on indicates that the MOP generation is able to effectively leverage parallelism, but with adding many workers the benefit gains diminish as the coordinator is not able to allocate keys faster than the workers propose them. Partitioning the keyspace for multiple leaders could enable higher parallelism if needed.

## Coordinator Batch size

This experiment shows how changing the coordinator batch size, or the number of worker proposals being received at once before processing, affects the MOP generation. The coordinator batch size was tested for values between 1 and 22.

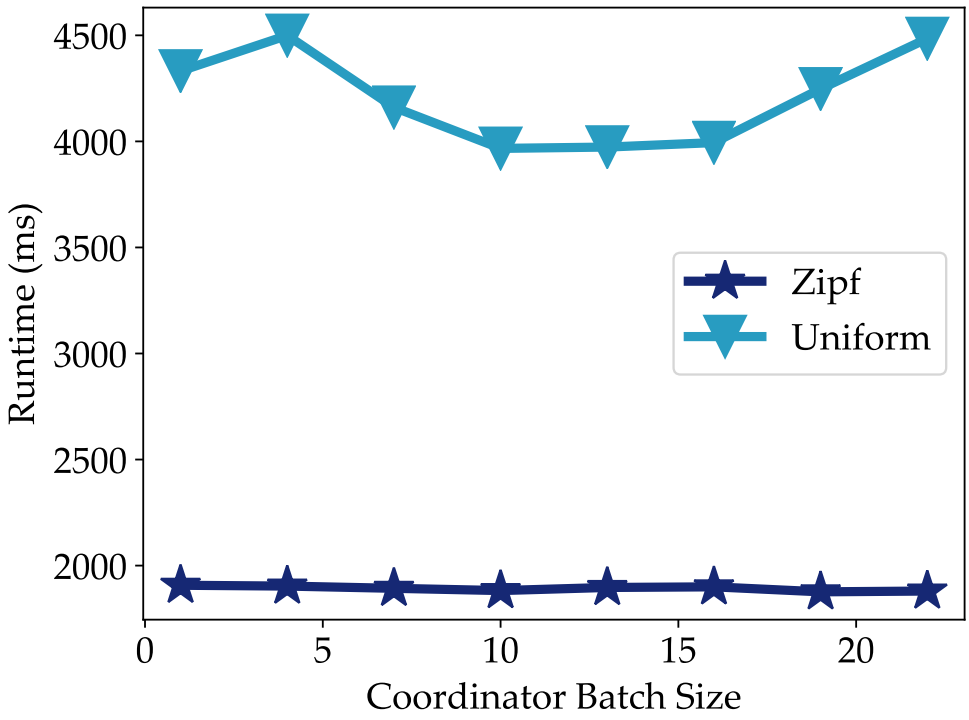



Figure 3.20: Evaluating the effect of coordinator batch size on MOP generation runtime using uniform and zipf workloads. 24 workers are being used with worker batch size set to 1

In Figure 3.20, we show that, for uniform workloads, by increasing the coordinator batch size, up to a point, the key allocation process runs faster as the coordinator can process groups of values more efficiently than a single value at a time. However, by making the coordinator do more work before sending keys back to the workers, we incur costs on the worker side because the workers must wait for a response before they continue processing values. This results in the eventual increase in runtime that happens with large coordinator batch sizes. Unlike the uniform workload, the zipf workload does not see much change in runtime as workers will not have to send as many messages to the coordinator. This means leader does not need to process many values, and the workers do not need to wait for many responses.

## Impact of C-MOP Layers' Size and Spacing

These experiments show how additional C-MOP layers affect ordered percentages, where all values within any C-MOP layer are considered ordered and all values that fail to fit into any level to be included in the disordered section. The tests in Figure 3.21 show that increasing the layer count increases ordered percentage as each successive layer creates more space for encoded values to be inserted into. Figure 3.21 also shows that increasing the MOP layer

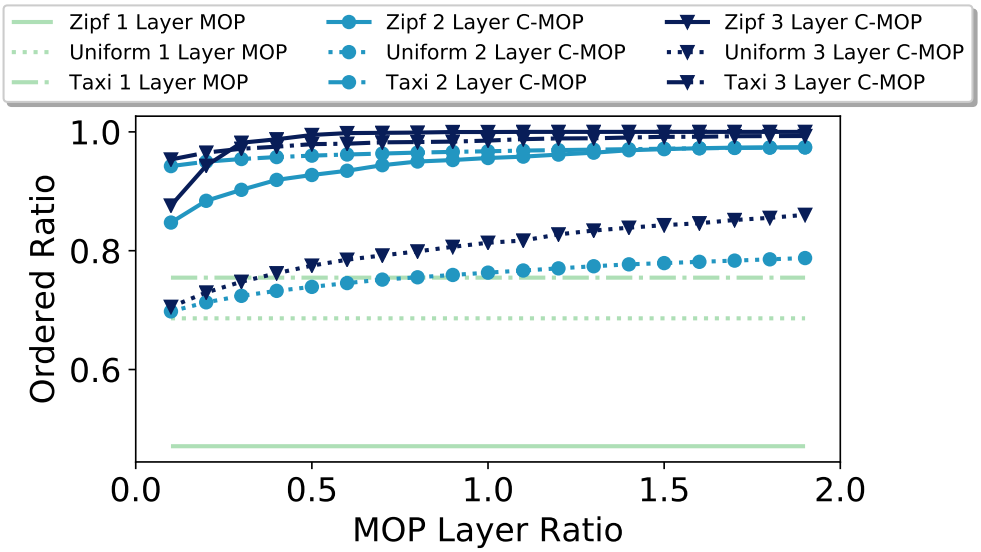



Figure 3.21: Evaluating the effect of MOP layer ratio on MOP ordered percentages using uniform, zipf, and taxi workloads.

ratio parameter increases the ordered percentage of the dictionary. The first layer of the C-MOP is calculated in the same fashion as in the normal, single-layer MOP. The successive layer sizes are set by multiplying the previous layer size by the MOP layer ratio. For example, if the MOP layer ratio is 0.20 and the first layer's key space size is 1000, the second and third layer key space sizes would be 200 and 40 respectively. In this experiment, we tested ratios between 0.10 and 2.00. Larger ratios result in larger key spaces of the new layers, so more space exists in the ordered part of the dictionary for new values. Therefore, by increasing this ratio, we can increase the orderedness of the MOP, but the size of the key space being allocated will grow. Given these results and the minimal query overhead, we believe at least

three layers should be used. In the following experiments we further analyze the impact of sizing the layers.

## Handling Distribution Changes

In this section, we will discuss the C-MOP's ability to correct for a worst-case scenario of large changes in incoming data distributions. By recalculating the estimated cardinality when new layers are being created, the C-MOP can dynamically grow the MOP layer ratio if the number of distinct incoming values was underestimated initially. This ratio will be set between 0.20 and 2.0 as previous experimental data showed that a too small ratio would not sufficiently increase ordered percentage and a too large ratio would over-inflate the key space.

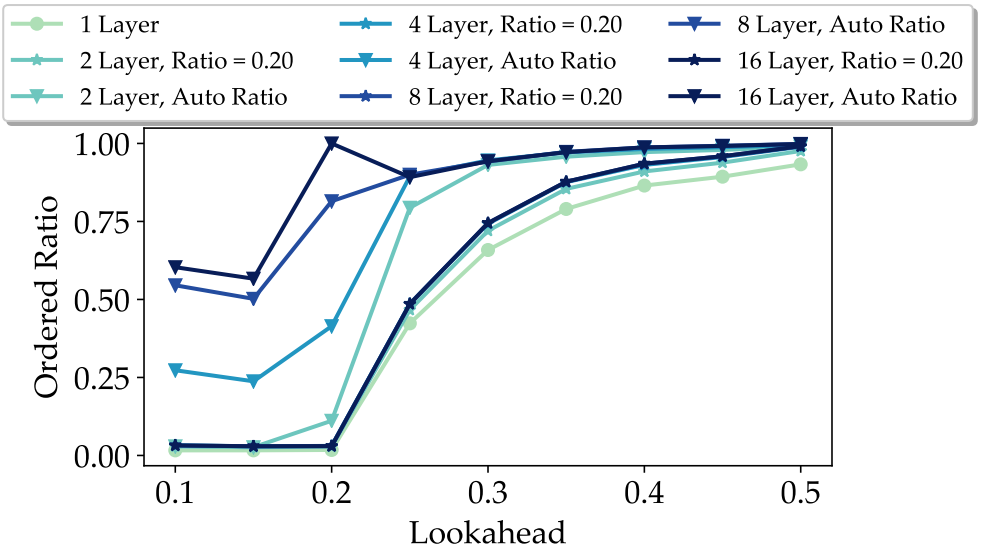



Figure 3.22: Evaluating the effect of dynamic MOP layer ratios when correcting for data distribution changes.

To demonstrate how C-MOP can correct for large distribution changes, we ran an experiment on a synthetic file where the first 20% of the file had a zipf distributed workload and the remaining 80% was uniform. When varying lookahead from 10% to 50%, Figure 3.22 shows how 1, 2, 4 8, and 16 layer C-MOPs using both a static MOP layer ratio percentage

of 0.20 and the dynamic MOP layer ratio handles distribution changes based on the amount of data collected on the new distribution.

For lookaheads less than or equal to 0.20, both single layer MOPs and static ratio C-MOPs performed poorly as they had no information of the distribution change. However, dynamic ratio C-MOPs expect to need more space after seeing the distribution change, and work to correct initial estimated cardinality mispredictions. Furthermore, as each successive layer will have more time to learn, adding additional layers will greatly improve ordered percentages. This shows that adaptively changing the MOP layer ratio allows for robustness in the presence of adverse datasets.

## Cardinality Estimation

Table 3.1 compares our simple cardinality estimator that divides the number of distinct values in the sample by the lookahead and a more advanced adaptive estimator [20] that separately estimates high frequency and low frequency cardinalities based on the sample. As the simple estimator generally overestimates the actual cardinality, it generates MOPs/C-MOPs with more padding and a higher ordered ratio than the adaptive estimator which generally has lower and more accurate cardinality predictions. For good MOP generation performance, we look to have a high ordered ratio and a low padding ratio to improve query performance and decrease file size. However, there is a tradeoff between the ordered and padding ratios. Increasing the number of keys allocated to the ordered section of the dictionary will increase the ordered ratio at the cost of extra padding and vice versa. As the cardinality estimation only affects the size of the key space, it does not improve general MOP performance. On a given dataset, for a fixed pitch, an estimator could have good ordered ratio performance, good padding ratio performance, or be somewhere in the middle. Scaling the key space through pitch allows the MOP to be easily tuned along the ordered ratio-padding ratio performance spectrum for any given cardinality estimate–so long as the estimate is not completely off base. As our simple estimator and adaptive estimator produce reasonable estimates, we chose to

use the naive estimator as it is less computationally expensive. However, as described in the next experiment, the ordered ratio-padding ratio trade-off results in no padding ratio benefit when using one estimator over another.

| | | Zipf | | Uniform | | Taxi | |
|---|---|---|---|---|---|---|---|
| Method | Pitch | Ordered Ratio | Padding Ratio | Ordered Ratio | Padding Ratio | Ordered Ratio | Padding Ratio |
| MOP Simple | 1 | 0.711 | 0.699 | 0.662 | 0.457 | 0.910 | 0.850 |
| | 2 | 0.818 | 0.840 | 0.802 | 0.688 | 0.950 | 0.924 |
| | 4 | 0.891 | 0.918 | 0.896 | 0.837 | 0.975 | 0.962 |
| | 8 | 0.945 | 0.956 | 0.950 | 0.959 | 0.988 | 0.981 |
| MOP Adaptive | 1 | 0.470 | 0.265 | 0.686 | 0.501 | 0.755 | 0.356 |
| | 2 | 0.582 | 0.481 | 0.813 | 0.711 | 0.796 | 0.540 |
| | 4 | 0.698 | 0.677 | 0.902 | 0.848 | 0.868 | 0.755 |
| | 8 | 0.806 | 0.823 | 0.953 | 0.923 | 0.919 | 0.868 |
| C-MOP Simple | 1 | 0.960 | 0.740 | 0.743 | 0.532 | 0.979 | 0.879 |
| | 2 | 1.000 | 0.868 | 0.890 | 0.741 | 1.000 | 0.939 |
| | 4 | 1.000 | 0.933 | 0.973 | 0.867 | 1.000 | 0.970 |
| | 8 | 1.000 | 0.966 | 1.000 | 0.933 | 1.000 | 0.984 |
| C-MOP Adaptive | 1 | 0.621 | 0.302 | 0.771 | 0.573 | 0.821 | 0.445 |
| | 2 | 0.779 | 0.529 | 0.903 | 0.761 | 0.865 | 0.615 |
| | 4 | 0.934 | 0.720 | 0.975 | 0.877 | 0.932 | 0.800 |
| | 8 | 1.000 | 0.853 | 1.000 | 0.938 | 0.985 | 0.893 |

Table 3.1: Evaluating cardinality estimation techniques.

## Key Space

Here we compare key space sizes between single-layered MOPs and multi-layered C-MOPs with both the simple and adaptive estimators. For each experiment, we adjusted the pitch until dictionaries were 70%, 80%, and 90% ordered. The average key space needed to achieve these ordered percentages are shown in Figure 3.23. All MOPs were generated using the same sets of zipf, uniform, and taxi data. C-MOP generation can better leverage key space in several, smaller order preserving sections than one large one as new layers can correct for distribution changes and help correct mispredictions made while sampling. By adding a second layer, we can generally see a reduction in the key space needed to produce a dictionary when compared to single-layered MOPs. The MOP generation is better able to leverage the allocated space in several, smaller order preserving sections than one large one. If the initial

sample is not fully representative of the distribution of data, there will be values that the MOP did not account for. These values will then overflow into the successive layers and influence the distributions of the new layers. The layering in C-MOPs can then work to correct mispredictions made in the initial sampling process.

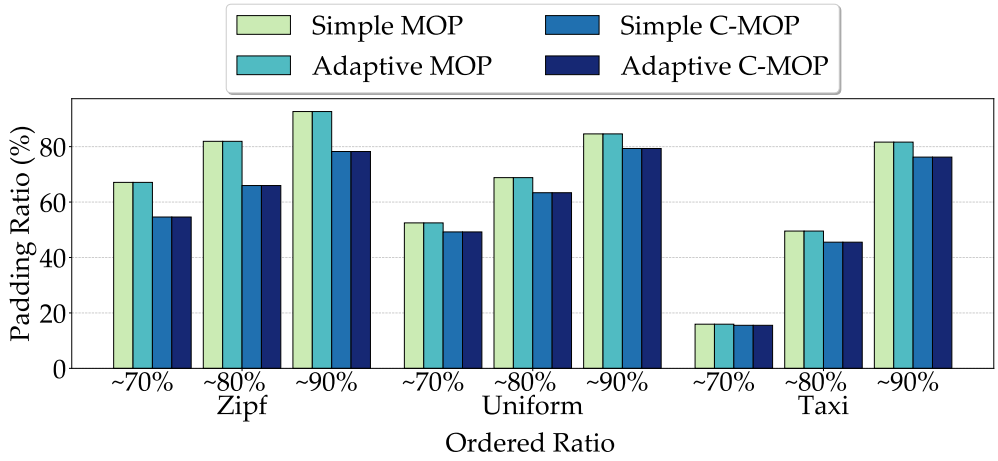



Figure 3.23: Evaluating the space saving benefit of C-MOP with regard to padding ratio.

The choice of estimator is also shown to have no effect on the padding ratio as reaching a target ordered ratio requires adjusting pitch until a specific slack value is reached. For example, to achieve an ordered ratio of 0.70, there needs to be enough slack space between sample values for 70% of the data to fit into the ordered section(s), regardless of the initial cardinality estimation.

MOP generation performance with regard to padding is dependent on the distribution of values, so we do not see any padding benefit when using different estimators. We therefore use a simple estimator as it is less computationally expensive.

Figure 3.24 also shows how different techniques for allocating keys to successive layers affects the key space used. When the distribution stays constant throughout the file, as in the uniform workload, a significant benefit can be had by using a histogram of the previous data to influence the distribution in the successive layers. The histogram insertion technique is able to make corrections based on all data seen before the first spillover into a new layer, so it is able to notice congested segments in the dictionary and take care of values that spill over

in these regions effectively. However, in cases where distributions do not stay constant, the uniform insertion technique has small benefits over the histogram method. As the uniform method does not bias key spaces based on any previous information, it is not penalized for predicting wrong, which may happen when changes in incoming values cause spillovers in unexpected ranges, like seen in the zipf and taxi workloads. Therefore, while a histogram based approach can benefit certain cases, we opt for simplicity and stay with a uniform allocation strategy.

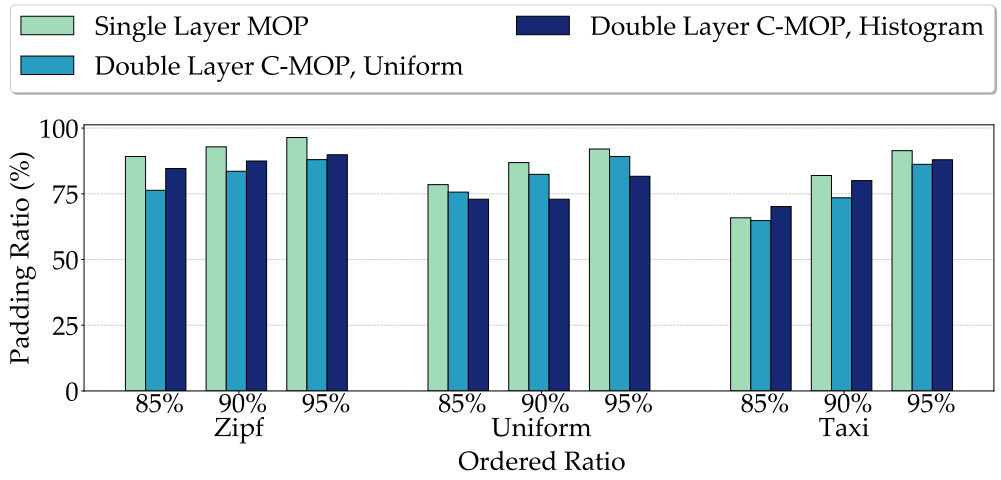



Figure 3.24: Evaluating the benefit of C-MOPs with regard to padding ratio when using uniform and histogram insertion techniques for the second layer.

## Column Sortedness

As described in Section 3.6.1, MOP draws the sample from the file head by default and falls back to a uniform sampling strategy if the file appears sorted (i.e. Kendall's Tau [49] $\geq 0.8$ or $\leq -0.8$). In Figure 3.25 we observe generating a MOP on sorted columns to justify this approach. Experiments are run on both a sorted and randomized version of three different columns, one with a zipf distribution, one with a uniform distribution, and one from the taxi dataset. The MOPs for the randomized columns were generated using a head sample, and the MOPs for the sorted columns were generated using both a head sample and a uniform sample. Each sampling strategy for each column was then used to generate both a 1-layer

MOP and an 8-layer C-MOP.

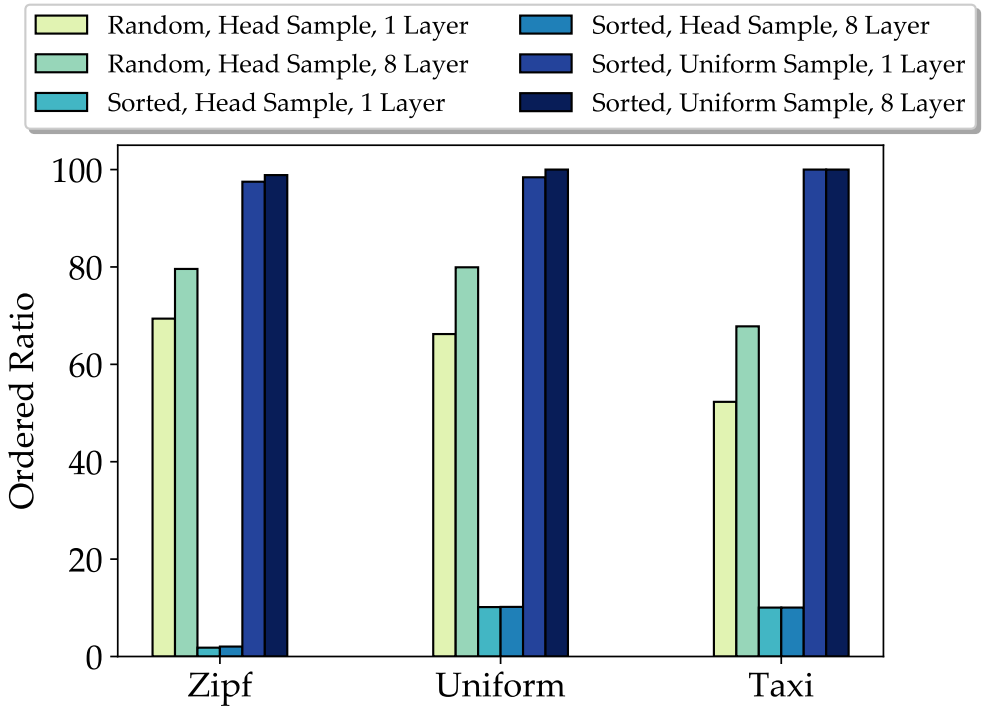


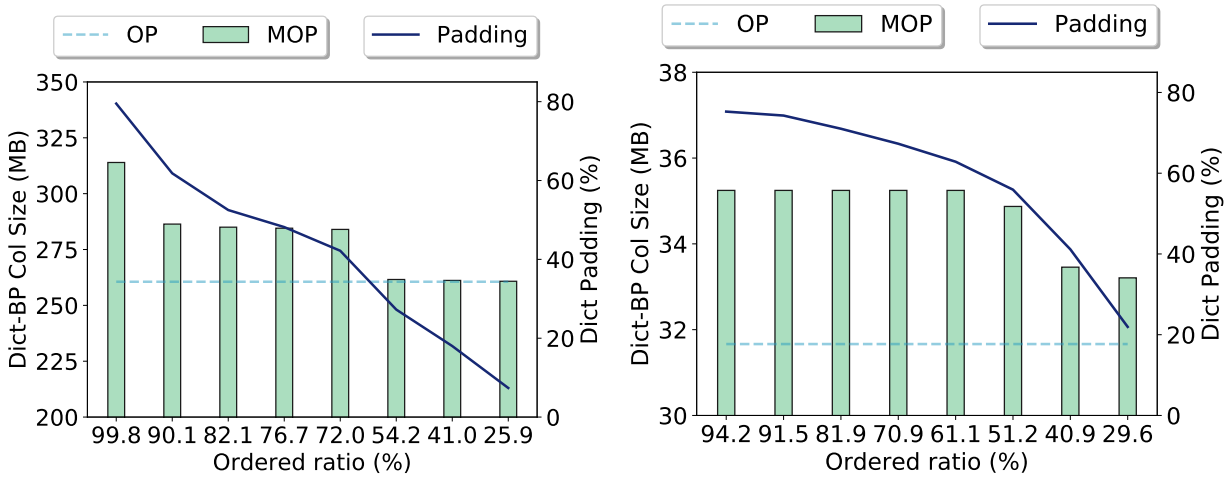

Figure 3.25: Evaluating the impact of sorted data and sampling strategy on a MOP's ordered ratio.

When using head sampling on sorted columns, MOPs have poor ordered percentages. All batch values being inserted after the sample will be greater than the largest already inserted value, so the only available room in the ordered section is in the remaining space after the last sample value.

When forced to resample uniformly, the sample values will reflect the overall column distribution, so ordered percentages increase. However, generation then incurs the costs of taking a uniform sample. To read and parse the file sequentially, the entire column may have to be read before the batching process. In certain cases, sampling may be cheap, such as a large file stored on a distributed block store, and uniform sampling is then preferred. For this work we target head-based samples, as we assume that random access is expensive and reading random records is expensive due to string escape characters (e.g. line breaks occurring outside of record delimiters).

### *3.9.4 Overall Compression Performance*

In this experiment set, we mainly focus on dictionary and bit-packing hybrid encoding compression performance. With dictionary encoding only, there is no storage overhead difference for MOP regardless of the key space used, as each record always takes exactly 4 bytes, which is not the case when bit-packing is introduced. Here, we use bit-packing encoding to further encode the targeted attributes in the previous experiment and report the column size. We use bit-packing locally in each partition of a Parquet file (called row groups) that truncates the keys to use the fewest bits to represent the largest key in the partition (i.e. 3 bits needed for keys 0–7).




((a)) TPC-H ((b)) Taxi

Figure 3.26: Encoded Column Size and Dict Padding

As is shown in Figure 3.26(a), the dashed line indicates the column size for the order preserving dictionary encoded dataset and the bars show the column size for MOP encoded datasets. The blue line corresponding to the right vertical coordinate represents the padding ratio in the MOP dictionary. For certain ordered ratios there is the same storage cost compared with OP. Even though the key space used for MOP increases, the number of bits to represent the max value does not change. For TPC-H one more bit is added on each record for the dataset with ordered ratio from 72.0% to 90.1%, and two more bits are needed if we want achieve MOP ordered ratio 99.8%. Three more bits are needed to get MOP ordered

ratio > 90% on targeted attribute of the Taxi dataset. According to our experiment, 9.9% and 11.3% extra storage (compared with OP encoded column size) are required respectively for targeted attribute in the TPC-H and Taxi dataset to achieve a MOP ordered ratio > 90%.

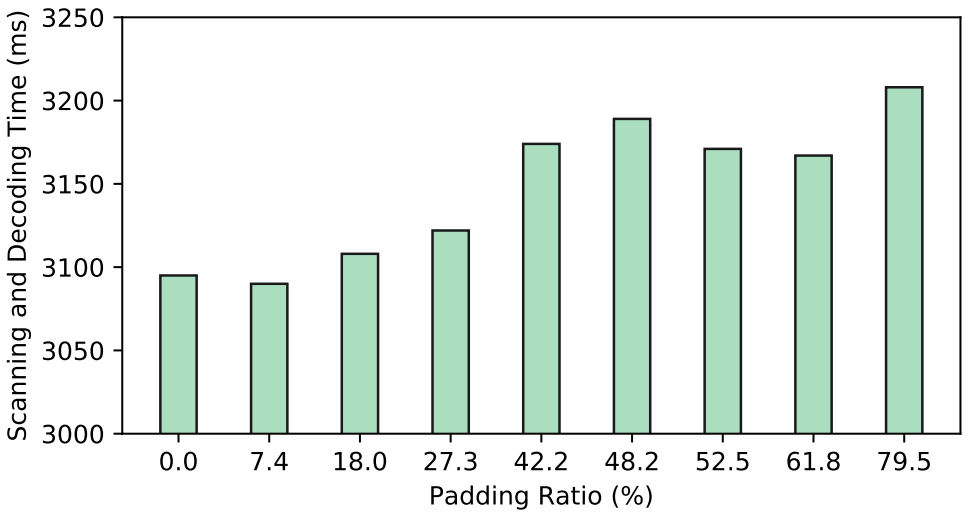



Figure 3.27: Scanning and decoding 180 million values.

Figure 3.27 shows the impact of padding ratios on scanning and decoding. Here, we take TPC-H lineitem shipdate and generate various MOPs with varied padding ratios (via ordered ratios). We then do a full scan of the column and decode every encoded key sequentially. These results show that a larger dictionary negatively impacts decoding performance.

## 3.10 Conclusion

In this paper we introduce mostly order preserving dictionaries (MOP) for supporting efficient range queries on encoded datasets. We present a technique for generating a MOP with a limited sample of the input dataset while minimizing the size of the dictionary. In addition, we introduce a variation that uses cascading MOPs (C-MOP) that has multiple levels of ordered keys. We present query rewriting rules to minimize decoding of keys to minimize predicate evaluation latency. We implement MOP and C-MOP in the open-source columnar framework, Parquet, and evaluate query and generation performance. Our results

demonstrate that MOPs are able to accelerate range filtering and sorting, and achieve high order ratios with small samples.

# CHAPTER 4

# DECOMPOSED BOUNDED FLOATS FOR FAST COMPRESSION AND QUERIES

## 4.1 Introduction

Modern applications and systems are generating massive amounts of low precision floating-point data. This includes server monitoring, smart cities, smart farming, autonomous vehicles, and IoT devices. For example, consider a clinical thermometer that records values between 80.0 to 120.9, a GPS device between -180.0000 to 180.0000, or an index fund between 0.0001 to 9,999.9999. The International Data Corporation predicts that the global amount of data will reach $175ZB$ by 2025 [80], and sensors and automated processes will be a significant driver of this growth. To address this growth, data systems require new methods to efficiently store and query this low-precision floating-point data, especially as data growth is outpacing the growth of storage and computation [72].

Several popular formats exist for storing numeric data with varied precision. A *fixed-point* representation allows for a fixed number of bits to be allocated for the data to the right of the radix (i.e., decimal point) but is not commonly used due to the more popular floating-point representation. *Floats* (or floating-point) allows for a variable amount of precision by allocating bits before and after the radix point (hence the floating radix point), within a fixed total number of bytes (32 or 64). For floats, the IEEE float standard [40] is widely supported both by modern processors and programming languages, and is ubiquitous in today's applications.

However, two main reasons result in floats not being ideal for many modern applications: (i) an overly high-precision and broader range that wastes storage and query efficiency; and (ii) not being amenable for effective compression with efficient in-situ filtering operations. For the former reasons, many databases offer custom precision format, typically referred to as a *numeric* data type. For example, consider Figure 4.1 that shows how precision varies for the

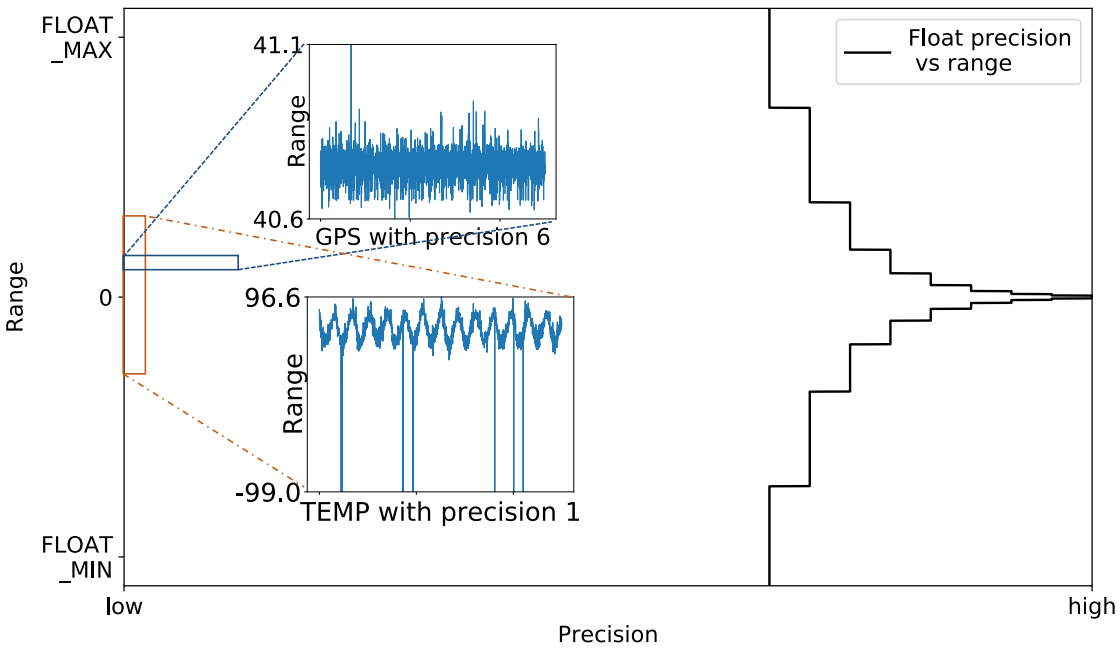



Figure 4.1: Many datasets only span a fixed range of values with limited precision, which is a small subset of broad float number range and precision.

IEEE float standard and sample application requirements on GPS and temperature datasets. The numeric approach allows for "just enough" precision but is not well optimized for efficient storage and filtering. Several methods have been explored for the latter reason, but due to the standard format, compression opportunities are limited for compression effectiveness, throughput, and in-situ query execution. This paper proposes a new storage format that extends the ideas of a numeric data type that supports custom precision but is optimized for fast and effective encoding while allowing to work directly over the encoded data.

To motivate our design, we first describe short-comings with popular compression techniques for bounded-precision floating data. Specifically, the recently proposed Gorilla method [75] is a delta-like compression approach that calculates the XOR for adjacent values and only saves the difference. Gorilla achieves compression benefits by replacing the leading and trailing zeros with counts. Gorilla is a state-of-the-art compression approach for floats, but it does not work well on low-precision datasets as low-precision does not impact float's representation similarity. In addition, Gorilla's encoding and decoding steps are slow because of

its complex variable coding mechanism. Alternative float compression approaches leverage integer compression techniques by scaling the float point value into integer [14]. Despite their simplicity, these approaches rely on multiplication and division operations that are usually expensive and, importantly, cause overflowing problems when the input value and the quantization (i.e., scaling) factor are too large.

General-purpose byte-oriented compression approaches, such as Gzip [25], Snappy [90] and Zlib [32], can also be applied to compress floats. The input float values are serialized into binary representation before applying byte-oriented compression. These compression approaches are usually slow because of multiple scans looking for commonly repeated byte sub-sequences. Furthermore, full decompression is needed before any query evaluation. Dictionary encoding [82, 61, 11] is applicable for float data, but it is not ideal since the cardinality of input float data is usually high, resulting in an expensive dictionary operation overhead. Note that we are only considering lossless formats or formats with bounded loss (e.g., configurable precision). Lossy methods (e.g., spectral decompositions [27] and representation learning [69]) may compress the data more aggressively but often at the cost of losing accuracy.

Decades of database systems demonstrate the benefit of a format that allows for a value domain that includes a configurable amount of precision. We take this approach, and to address the above concerns, integrate ideas from columnar systems and data compression for our proposed method, **B**o**U**nded **F**ast **F**loats compression (Buff). Buff provides fast ingestion from float-based inputs and a compressible decomposed columnar format for fast queries. Buff ingestion avoids expensive conditional logic and floating-point mathematics. Our storage format relies on a fixed-size representation for fast data skipping and random access, and incorporates encoding techniques, such as bit-packing, delta-encoding, and sparse formats to provide good compression ratios[1] and fast adaptive query operators. When defining an attribute that uses Buff, the user defines the precision and optionally the minimum

1. Compression ratio is defined as $\frac{\text{compressed\_size}}{\text{uncompressed\_size}}$.

and maximum values (e.g., 90.000-119.999). Without the defined min and max values, our approach infers these through the observed range, but at a decreased ingestion performance. Our experiments show the superiority of Buff over current state-of-the-art approaches. Compared to the state-of-the-art, Buff achieves up to 35× speedup for low selective filtering, up to 50× speedup for aggregations and high selective filtering, and 1.5× speedup for ingestion with a single thread, while offering comparable compression sizes to the state-of-the-art.

We start with a review of the relevant background (Section 4.2), which covers numeric representations (Section 4.2.1) and a wide range of compression methods (Section 4.2.2). In Section 4.3, we present Buff compression and query execution with four contributions:

- We introduce a *"just-enough"* bit representation for many real-world datasets based on the observation of their limited precision and range (Section 4.3.1).
- We apply an aligned decomposed *byte-oriented columnar storage* layout for floats to enable fast encoding with progressive, in-situ query execution on compressed data (Section 4.3.2).
- We propose *sparse encoding* to handle outliers during compression and query execution (Section 4.3.3).
- We devise an *adaptive filtering* to automatically choose between SIMD and scalar filtering (Section 4.3.4).

We evaluate Buff with its competitors in terms of compression ratio and query performance in Section 4.4, and conclude in Section 4.5.

## 4.2 Background and Related Work

In this section, we first introduce several numeric data representations, including the most popular float format. Then, we review several float compression approaches with a set of

| Value | Fixed<2,20> (Sign bit; Integer bits; Fractional bits) | 32-bits Float (Exponent bits; Mantissa bits) | Gorilla (Control bits: 11 if current meaningful bits does not fit bit range of previous meaningful bits; Control bits: 0 if xor=0) |
|---|---|---|---|
| | | | ***Step 1*** *XOR with previous value* (Leading zeros; Meaningful bits; Trailing zeros) | ***Step 2*** *write control bits, length and difference bits* |
| ***0.66*** | 00.10101000111101011100 | 00111111001010001111010111000011 | 00000000**10011100100011110010001**0 | 00111111000111101011100001010010 |
| ***1.41*** | 01.01101000111101011100 | 00111111101101000111101011100001 | 00000000**10011100100011110010001**0 | [b11,d8,**d22**] 10011100100011110010001 |
| ***1.41*** | 01.01101000111101011100 | 00111111101101000111101011100001 | 00000000000000000000000000000000 | # leading zeros (decimal) **# meaningful bits (decimal)** [b0] |
| ***1.50*** | 01.10000000000000000000 | 00111111110000000000000000000000 | 00000000**11101000111101011100001** | [b11.d9,**d22**] 11101000111101011100001 |
| ***2.72*** | 10.10111000010100011111 | 01000000001011100001010001111011 | 0**11111111101110000101000111101**1 | [b11,d1,**d31**]11111111101110000101000111011 |
| ***3.14*** | 11.00100011110101110001 | 01000000010010001110101110000011 | 00000000**11001101110000110111**000 | [b10]00000000110011011100001101110000 |
| ***size(bits)*** | *132* | Integer bits; Fractional bits | *192* | Control bits: 10 if use previous meaningful bits range | *176* |

| Sprintz | | | | | Mostly | | Dictionary | | | | Gzip |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Step 1*** *Quantize×100* | ***Step 2*** *prediction* | *Predicted value* | ***Step 3*** *delta* | ***Step 4*** *bit-packing* | ***Step 1*** *quantize* | ***Step 2*** *mostly16* | ***Step 1*** *dictionary encoding* | | *encoded* | ***Step 2*** *bit-packing* | ***Step 1*** *LZ77;* ***Step 2*** *Huffman coding* |
| 66 | **Lookup table** | | 66 | 1000010 | 66 | 0x0042 | 0.66 | 0 | 0 | 000 | *3f28f5c3* |
| 141 | | 66 | 75 | 1001011 | 141 | 0x008D | 1.41 | 1 | 1 | 001 | *3fb47ae1* |
| 141 | [a,b,c]->d | 141 | 0 | 0000000 | 141 | 0x008D | 1.50 | 2 | 1 | 001 | *[8,8]* |
| 150 | ... | 141 | 9 | 0001001 | 150 | 0x0096 | 2.72 | 3 | 2 | 010 | *3fc00000* |
| 272 | *Vk = Predictor( Vk-1, Vk-2, ...)* | 150 | 121 | 1111001 | 272 | 0x0110 | 3.14 | 4 | 3 | 011 | *402e147b* |
| 314 | | 272 | 42 | 0101010 | 314 | 0x013A | | | 4 | 100 | *404[40,5]* |
| ***size(bits)*** | | | | *42* | | *96* | | | | *210* | *328** |

**For Gzip, we skip showing step 2 Huffman coding for space, and report the final encoded size.*

Figure 4.2: Compression examples for different approaches.

examples. Finally, we include popular query-friendly data structures that have influenced our compression format.

### *4.2.1 Numeric Data Representation*

Numeric data representation is the internal representation of numeric values in any hardware or software of digital systems, such as programmable computers and calculators [84]. Many data representations are developed according to different system and application requirements. The most popular implementations include fixed-point (including numeric data type) and float-point.

**Fixed-point** uses a tuple $< sign, integer, fractional >$ to represent a real number $R$:

$$R = (-1)^{sign} * integer.fractional$$

Fixed-point partitions its bits into two fixed parts: signed/unsigned integer section and fractional section. For a given bits budget $N$, the fixed-point allocate a single bit for sign, $I$ bits for the integer part, and $F$ bits for the fractional part (where $N = 1+I+F$). Fixed-point always has a specific number of bits for the integer part and fractional part. Figure 4.2 shows examples of fixed-point; as we can see from the "Fixed" column, the radix point's location

is always fixed, no matter how large or small the corresponding real number is. Fixed-point arithmetic is just scaled integer arithmetic, so most hardware implementations treat fixed-point numbers as integer numbers with logically decimal point partition between integer and fractional part. Fixed-point usually has a limited range ($-2^I \sim 2^I$) and precision ($2^{(-F)}$), thus it can encounter overflow or underflow issues. A more advanced dynamic fixed-point representations allow moving the point to achieve the trade-off between range and precision. However, this is more complicated as it involves extra control bits indicating the location of the point. Fixed-point is rarely used outside of old or low-cost embedded microprocessors where a floating-point unit (FPU) is unavailable. Fixed-point is currently supported by few computer languages as floating-point representations are usually simpler to use and accurate enough.

**Floating-point** uses a tuple $< sign, exponent, mantissa >$ to represent a real number $R$:

$$R = (-1)^{sign} * mantissa * \beta^{exponent}$$

Instead of using a fixed number of bits for the integer part and fractional part, Float point reserves a certain number of bits for the exponent part and mantissa, respectively. The base $\beta$ is an implicit constant given by the number representation system, while in our floating-point representation, $\beta$ equals to 2. The number of significant digits does not depend on the position of the decimal point. For a given bit budget $N$, the floating-point allocates a single bit for sign, $E$ bits for the exponent part, and $M$ bits for the mantissa part (where $N = 1 + E + M$). The IEEE standard defines the format for single and double floating-point numbers using 32 and 64 bits, respectively. The exponent is encoded using an offset-binary representation with the zero offset by exponent bias:

$$R = (-1)^{sign} * 1.mantissa * 2^{exponent}$$

The IEEE standard specifies a 32-bits float as having: Sign bit: 1 bit, Exponent width: 8

bits (offset by 127), and Significand precision: 24 bits (23 explicitly stored). Figure 4.2 shows several examples of 32-bits float format; as we can see from the '32-bits float' column, the underlined bits correspond to the fractional bits, and the radix point is floating base on its represented value.

**Numeric** data type allocates enough bits for a given real number depending on the precision and scale of the input. Numeric data type can store numbers with a very large number of digits and perform calculations exactly. To declare a column of type numeric, we use the syntax: NUMERIC(precision, scale). The scale is the count of decimal digits in the fractional part, to the right of the decimal point. The precision is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point (e.g., 65.4321 has a precision of 6 and a scale of 4).

The Numeric data type is used in most database systems, e.g., MySQL, PostgreSQL, Redshift, and DB2, when exactness is required. However, the flexibility and high precision come at the cost of more control bits associated with each value. The storage cost of the Numeric type varies depending on the system implementation. PostgreSQL uses two bytes for each group of four decimal digits, plus $3 \sim 8$ bytes overhead for each value. In MySQL, Numeric values are represented in a binary format that packs nine decimal digits into four bytes. This makes the Numeric type inefficient for space. Besides, the query on Numeric data type is either by materializing to float point for approximate calculation or using its own exact arithmetic which is very expensive compared to the integer types or the floating-point types described earlier.

### *4.2.2 Compression for Decimal Numbers*

Due to the popularity of float-point data, several compression techniques exist, including methods for float-point data, general-purpose methods, and migrated approaches from integer compression. We limit our discussion to lossless compression techniques.

**Gorilla** [75] is an in-memory time-series database developed by Facebook. It introduces

two encodings to improve delta encoding: delta-of-delta (delta encoding on delta encoding, e.g., [101,102,103, 104] are encoded as [101,1,0,0]) for timestamps, which is usually an increasing integer sequence, and XOR-based encoding for value domain, which is a float type. In the XOR-based float encoding, successive float values are XORed together, and only the different bits (delta) are saved. The delta is then stored using control bits to indicate how many leading and trailing zeroes are in the XOR value. Similar compression techniques are also used in numerical simulation [59, 42, 58] and scientific computing [78, 19]. Figure 4.2 shows an example of Gorilla encoding with the first step calculating XOR, and the second step writing the control bits, number of leading zeros, number of meaningful bits, and actual meaningful bits. We use 32-bits to save space even though Gorilla was originally designed for 64-bits format. Gorilla is the state-of-the-art approach for float compression, which is widely used in many time series database systems, such as InfluxDB [4] and TimescaleDB [6].

However, Gorilla compression uses variable-length encoding, which means records are not aligned with their encoded bits. The encoded bits have to be decoded sequentially to reach the target records. Each value depends on its previous record; thus, for a target value, all previous values must be decompressed. Those features impede effective record skipping and random data access.

**Sprintz** [14] was initially designed for integer time series compression. Sprintz employs a forecast model to predict each value based on previous records via a lookup table, and then encodes the delta between the predicted value and the actual value. Those delta values are usually closer to zero than the actual value, making it smaller when encoded with bit-packed encoding. It is also possible to apply Sprintz to floats by first quantizing the float into an integer. As is shown in Figure 4.2, we first multiply input values by 100 and get a series of integers. Then we predict the next value based on the prediction model. In the third step, we calculate the delta between the prediction and actual value, and compress the delta with bit-packing encoding in the last step. Sprintz uses fixed-length for encoded values, which

is easy to locate the target value. However, to decode the target value, we need to fully decode prior values, since previous values are still needed to predict the current value and then decode the value with compressed delta. Furthermore, the multiplication of decimal numbers is costly and potentially raises integer overflow issues during quantization.

**Mostly encoding** encodes the data column where its data type is more extensive than most of its stored values required. Mostly encoding is used for integer and numeric data type compression in the Amazon Redshift data warehouse [38]. For numeric values, Mostly encoding quantifies floats into an integer before applying compression. Most of the column values are encoded to smaller data representation with mostly encoding, while the remaining values that cannot be compressed are stored in their original form. Mostly encoding can be less effective than no compression when a high portion of the column's values cannot be compressed. Figure 4.2 shows an example of Mostly encoding. It first quantifies the float values and then applies MOSTLY16 (2-byte) for the given data as it needs 9 bits for each quantized value (we use hex to save space). This representation is less effective than bit-packing encoding that stores input value using as few bits as possible.

**General-purpose byte-oriented compression** encodes the input data stream at the byte level. Popular techniques, such as Gzip [25], Snappy [90] and Zlib [32] derive from the LZ77 family [97] that looks for repetitive sub-sequence within a sliding window on the input byte stream and encodes the recurring sub-sequence as a reference to its previous occurrence (first step in Figure 4.2). For a better compression ratio, Gzip applies Huffman encoding on the reference data stream (second step in Figure 4.2). Snappy only applies LZ77 but skips Huffman encoding for higher throughput. Byte-oriented compression treats the input values as a byte stream and encodes them sequentially. The data block needs to be fully decompressed before any original value can be accessed.

### *4.2.3 Query-friendly Storage Layout*

Compression should not only keep the data size small but also support fast query execution. Prior work speeds up query performance by introducing a hierarchical data representation layout. Some works [56, 88, 77, 64] are orthogonal to data compression but are applicable to Buff to speed up the query performance.

DAQ [77] uses a bit-sliced index representation and computes the most significant bits of the result first then less significant ones. With bit-sliced index representation, it performs efficient approximation and provides efficient algorithms for evaluating predicates and aggregations for unsigned integers type. Column Sketch [39] introduces a new class of indexing scheme by applying lossy compression on a value-by-value basis, mapping original data to a representation of smaller fixed width codes. Queries are evaluated affirmatively or negatively for the vast majority of values using the compressed data, and check the original data for the remaining values only if needed. PIDS [46] identifies common patterns in string attributes from relational databases and uses the discovered pattern to split each attribute into sub-attributes and further compress it. The sub-attributes layout enables a predicate push-down to each attribute. And the intermediate result from the prior column could be projected to the following attributes efficiently with the aid of fast data skipping and progressive filtering.

BitWeaving [56] provides fast scans in a main-memory database by exploiting the parallelism available at the bit level in modern processors. It organizes the input codes horizontally or vertically into a processor word, allowing early pruning techniques to avoid accesses on unnecessary data at the bit level and speed up the scan performance. ByteSlice [30] is another main memory storage layout that supports both highly efficient scans and lookups. Similar to BitWeaving, ByteSlice assumes encoded binary as input. But instead of stripping the input code in bit units, ByteSlice decomposes the input code at the byte level and pads the trailing bits into a byte to achieve better read/write speed and be compatible with SIMD instructions. MLWeaving [88] is an in-memory data storage layout that allows efficient materialization of quantized data at any level of precision. Its memory layout is based on

BitWeaving, where the first bit in a batch of input values is saved as a word, followed by the second bits of the same batch, and so on. Data at any level of precision can be retrieved by following a different access pattern over the same data structure. Using lower precision input data is of special interest, given their overall higher efficiency.

These techniques speed up query performance by progressively filtering out disqualified records and narrowing down the records that need to be parsed. They mainly process a query in their defined unit (either sub-attributes or bits) on their columnar data layout. This fine-grained columnar data layout is the foundation for Buff.

## 4.3 Buff Overview

Many applications generate data that varies within a specific range with limited precision. Based on our discussion of state-of-the-art, none can leverage this feature to support efficient compression and query execution. To alleviate those deficiencies, we propose a novel compression method for floats. The compression works at two levels: eliminating the less significant bits based on a given precision (Section 4.3.1), and splitting float into the integer part and fractional part, then compressing those bits separately with two splitting strategies (Section 4.3.2). We then introduce sparse encoding to handle outliers to avoid compression performance deterioration (Section 4.3.3). In the end, we describe query execution on our compressed data format (Section 4.3.4). With these techniques, we achieve both good compression ratio and outstanding query performance.

### *4.3.1 Bounded Float*

Computer systems can only operate on numeric values with finite precision and range. Using floating-point values as real numbers does not clearly identify the precision with which each value must be represented. Too small precision yields inaccurate results, and too big wastes computational and storage resources [26]. We target applications that need limited precision

and data that falls within a finite range. Therefore, our proposed compression takes full advantage of the range and precision features of a given dataset.

## Bounded Precision

Float point data-type uses most bits for *mantissa*, which is not necessarily needed for many sensors or applications since many sensors usually provide limited decimal precision (e.g., the clinical thermometer has one place of precision 0.1 °F). However, the decimal precision is not aligned with those *mantissa* bits in a float due to the floating radix point in the format. Float format makes the most of its *mantissa* bits to get the best approximation for the given number. With 0.1 for example, all 52 *mantissa* bits are used to get its approximation (0.1000000000000000555) in IEEE double format. This mechanism is intended to support high precision for float format, but it impedes efficient float compression in most real-world cases. Those *mantissa* trailing bits provide far more extra precision than required, which leads to a huge bits change with a tiny number fluctuation. For example, Gorilla achieves good compression by first applying XOR to the current number with the previous one, then removing the leading and trailing zeros. The problem here is that *mantissa* bits are too sensitive to the number changes, making it difficult to leverage the trailing zero benefits. Suppose we know the precision of the incoming numbers. In that case, we can provide just

Table 4.1: Bounded float still keeps decimal precision

| Real number | | Sign | Exp(8 bits) | Mantissa(23 bits) | Actual value |
|---|---|---|---|---|---|
| 3.14 | binary | 0 | 10000000 | 10010001111010111000011 | 3.1400001 |
| | bounded-2 | 0 | 10000000 | 10010001100000000000000 | 3.13671875 |
| | | | | | ≈ 3.14 |

enough precision support by eliminating the less significant *mantissa* bits. At the same time, we are still able to provide a guaranteed precision float for downstream analytic. Table 4.1 shows float representation of a given value 3.14. As is shown in the binary row, the float format saves the closest approximation as 3.1400001. If we only care about two decimal positions after the decimal point, as is shown with *bounded-2* row, we can remove some

trailing bits, while its value is still possible to approximate to the original value with two places of precision. The removal of insignificant bits saves a lot of bits when representing a number with a given precision. This is one motivation for Buff. To efficiently determine the number of fractional bits need for different required precision, we run brutal-force verification on all possible numbers under each precision requirement. We get the maximal factional bits needed for its corresponding precision, as shown in Table 4.2, which becomes the lookup table for bounded precision. We only provide up to 10 digit places precision here as it is good enough for most datasets. We can extend to higher precision, but it is less meaningful as commonly used float only has 52 bits for mantissa.

Table 4.2: Number of bits needed for targeted precision

| **Precision** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Bits needed** | 5 | 8 | 11 | 15 | 18 | 21 | 25 | 28 | 31 | 35 |

## Bounded Range

Another observation for most application scenarios is bounded range, The measured value is delimited by a lower and an upper measuring bound limit that defines the measuring span. The bounded range could result from either the measuring range of the instrument (e.g., clinical thermometer measuring range from $35°C$ to $42°C$), the physical definition of measurand (e.g., CPU usage is defined between 0% to 100%), or based on other domain knowledge. The bounded range is another feature that is helpful when compressing the data since we only focus on the given range, and we can encode the number in a more concise but accurate manner. Buff can work with no provided ranges, but as our experiments show, it hinders ingestion performance as the compression and allocation needs to find the minimal and maximal observed value when encoding a set of values. With bounded precision and range, we can get rid of the less useful and redundant bits initially designed for higher precision and broader range. With those two techniques, we can get a condensed representation of the input data. In addition to the compression benefits, Buff also adopts float splitting

to organize our compressed data better.

### *4.3.2 Float Splitting and Compression*

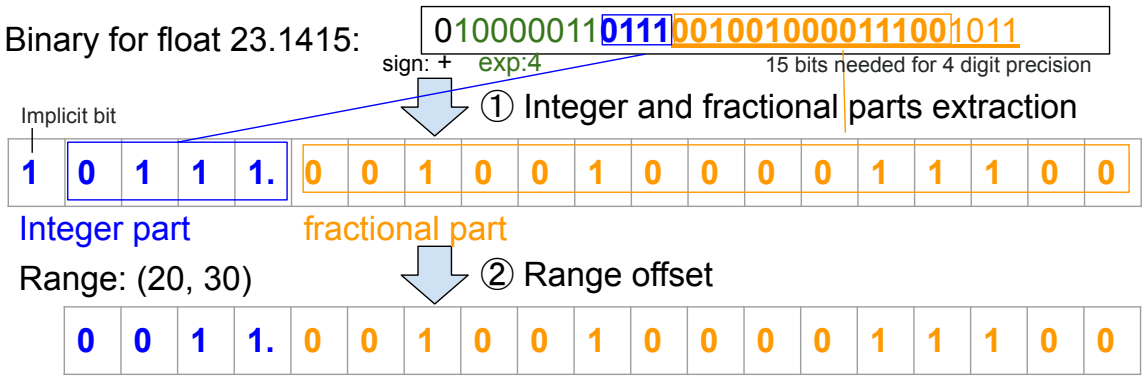



Figure 4.3: Extracting range bounded integer part and precision bounded fractional part for float number

The basic idea of float spitting is decomposing the input number into integer and fractional parts. This division separates the given number by radix point in a logical way and organizes each component into a different physical location. We can then apply efficient compression to those two parts, respectively. Bit splitting on fixed-point numbers is intuitively easy to do as its fixed position for the decimal point. However, the splitting is not easily applicable to float data directly, since float numbers are not aligned by the radix point. We extract both parts of a given float number with bit operations. First, we extract the exponent with a bit-mask to locate the decimal point. With the exponent value, we then bit-shift the float number and extract all left-side bits (including one implicit bit) as the integer part, and right-side bits as the fractional part. Figure 4.3 shows how this process works for a float value 23.1415 with scale 4 represented in 32-bit IEEE format. The exponent (marked with green font) is decoded as 4, which means the integer part has 5 bits (with a single leading 1 given implicitly by IEEE standard). We then ① extract 4 leading bits from the mantissa(marked with red) and add a leading 1 to get the integer part. Based on Table 4.2, we extract 15 following bits as fractional part for required precision **4**. ② With the range information either given by user or detected by our encoder, we offset the integer

part by the minimal value to further condense the range space. This splitting is fast and efficient with only bit-wise operations and no *if* branches. It can be further improved if the user-provided data range has the same exponent value (e.g. all values fall within the same power of 2), in which case those float numbers have a fixed decimal point location, and float splitting could be conducted easier with direct bit-masking.

The float splitting divides the given float numbers into two components and then compresses each component separately. We discuss its potential benefits on progressive filtering and data skipping later in Section 4.3.4. Here our analysis focuses on its compression efficiency in terms of compression ratio and reads/writes throughput. The encoded bit length of the integer part is determined by the range of input numbers, while the encoded bit length for the fractional part is determined by the actual precision required. We can compress both parts effectively with delta and bit-packing encoding for the integer part and bounded precision for the fractional part. The fixed-length encoding simplifies the encoding and decoding process and makes it possible for data skipping to be applied on the encoded bits directly, which improves performance.

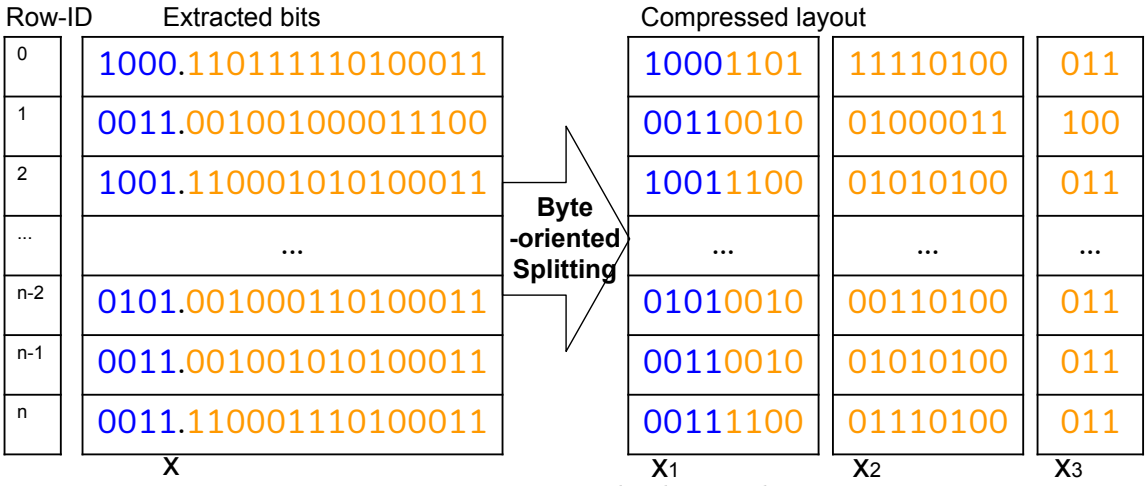


Figure 4.4: Byte-oriented columnar layout

However, float splitting introduces some overhead for the compression process. Byte-oriented reads/writes are more efficient than bit-oriented operations because of low-level data type boundary-crossing issues where the CPU needs to unpack and offset the bits to

align them on byte boundaries. Assume we have $m$ bits to read/write, this could be finished by $\lfloor \frac{m}{8} \rfloor$ times byte read/write, and at most one more bit read/write for remaining $m - \lfloor \frac{m}{8} \rfloor$ bits. Whereas float splitting divides $m$ bits into $x$ bits and $y$ bits (where $x+y=m$), and this leads to $\lfloor \frac{x}{8} \rfloor + \lfloor \frac{y}{8} \rfloor$ times byte read/write, and at most twice bit read/write for remaining $x - \lfloor \frac{x}{8} \rfloor$ bits and $y - \lfloor \frac{y}{8} \rfloor$ bits. Additionally, the integer part usually takes less than one byte for many datasets because of the narrow value range for many measurands. To avoid slowness of bits read/write, we can always pad int bits with several following fractional bits to extend it into a byte. This keeps the fine-grained float splitting, but good read/write performance as well. If the goal is to save $m$ bits anyway, we should split the bits in a more efficient way. This is the motivation of our advanced splitting version: byte-oriented float splitting.

Byte-oriented float splitting compression works similarly to the float splitting discussed previously. We first extract integer bits and just enough fractional bits that meet the precision requirement, then offset the intermediate value by minimal value before writing those bits in the byte unit separately. Each byte unit is treated as a *sub-column* and stored together. We store the remaining trailing bits of each record as the last sub-column, as is shown in Figure 4.4. We save the range and precision information as metadata along with compressed data to guarantee the decompression capability. With byte-oriented float splitting, compression and decompression are improved because of the fast byte reads/writes. We can also support custom variable precision materialization and achieve more efficient query execution because of a byte-oriented splitting layout.

### *4.3.3 Handling Outliers*

In addition to variable precision materialization support brought by byte-oriented splitting, we can also handle outliers efficiently.

Outliers in the input data sequence are the data points that differ significantly from others. Outliers enlarge the value range significantly and further inflates the code space,

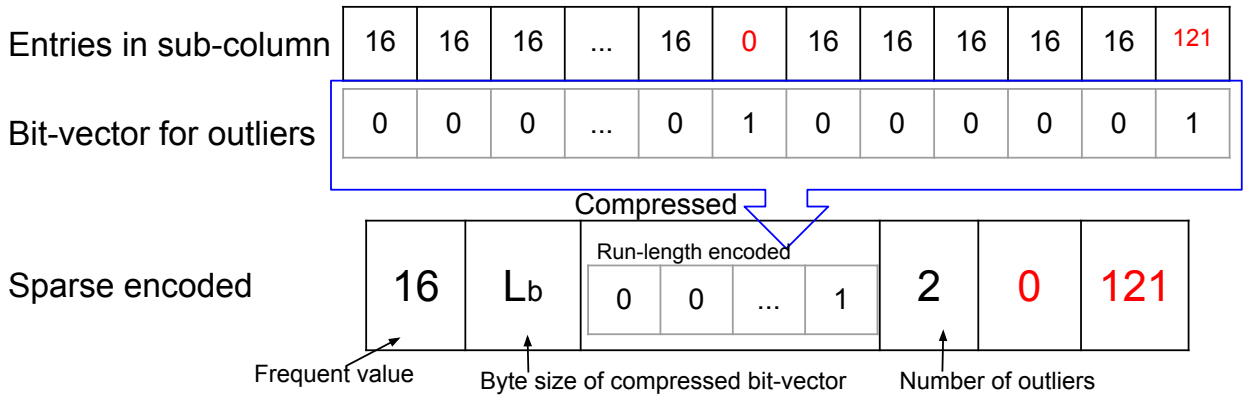



Figure 4.5: Sparse encoding condenses sub-column.

which requires more bits to encode the input data sequence. For example, 99% of the input value can be represented by 1 byte, while 1% needs 4 bytes. Encoding all numbers in the same length inflates the code space to 4 bytes because of the 1% outliers. The code space inflating, if not handled well, will deteriorate the compression effectiveness and query efficiency. Outliers can be too large or too small that are far away from the common range. While in either case, the outliers usually only inflate the higher bits either with padding zeros or padding ones. When there are both large and small outliers, the padding bits can be any bit sequence between all zeros and all ones. This requires us to detect outliers and distinguish the most frequent value in the higher encoded byte. We first head sample the dataset and then apply Boyer–Moore majority voting algorithm [16] in each compressed byte unit to find the majority item (occurring more than 50%). If the majority item has a frequency greater than 90%, we get most frequent value and re-organize the corresponding byte column to achieve better compression by *sparse encoding* as shown in Figure 4.5: we keep the frequent value at the beginning and use a compressed bit-vector to indicate the outlier records, followed by outlier values in the current byte column. The outlier handling enables better compression performance and faster query execution with the aid of bit vectors. Higher byte columns can be skipped for most queries that target on some normal values only.

Sparse encoding requires a number of bits to be allocated per frequent value, to indicate

if a value uses a frequent value or is an outlier. Compared with pure byte representation, sparse encoding further reduces the compressed data size if only there is a frequent item with a frequency greater than 1/8, assuming 1 bit overhead per record in the bit-vector. However, the outlier scheme comes with costs that special logic is required to handle outliers during compression and query execution. So we apply factor analysis on the frequency of frequent item through micro-benchmark with controlled outlier ratio. According to our experiments, sparse encoding achieves a significant query boost compared with pure byte representation when the frequency of frequent item is greater than 90%, which is the current default threshold in Buff.

### *4.3.4 Query Execution*

In addition to the compression benefits, byte-oriented float splitting improves the query execution on the target float columns in the following aspects: byte-oriented data arrangement enables fast reading/writing and value parsing. Fixed-length coding helps effective data skipping to save CPU resources. The sub-column layout enables predicate push-down and progressive query execution. For efficient query execution directly on the encoded sub-columns, we apply query rewriting to decompose original query operators into a combination of independent query operators on sub-columns. Those translated query operators could be executed independently and combined to obtain the final results. If we evaluate each query operator on the sub-column one at a time, it is possible to skip some records that are determined to be qualified or disqualified already by previous sub-column query operators. Therefore, only a small amount of records need to be parsed and further checked.

#### Progressive Filtering

With byte-oriented float splitting compression, we can progressively evaluate the translated filter on each sub-column sequentially, and obtain the final results when all filters are finished. For a given predicate $x\ OP\ C$ where $x$ is the attribute stored using Buff, $OP$ is the operator,

and $C$ is the operand of this predicate. Instead of decoding each record and matching the predicate, we rewrite the filtering predicate on the target column into a combination of filtering predicates on its compressed sub-columns. Assume there are $k$ sub-columns $x_1$, $x_2$, $x_3$, ..., $x_k$, we can get translated operands on those sub-columns as $C_1$, $C_2$, $C_3$, ..., $C_k$ and then apply filtering on each sub-column to obtain the final results [46].

To save computation in the filtering, we can avoid evaluating some records based on our current observations. During the progressive filtering process, all the intermediate results on each sub-column are shipped by a bit-vector, similar to late materialization [85]. Two bit-vectors $RESULTS$ and $TO\_CHECK$ are maintained to indicate the qualified rows and suspecting rows, respectively. The following filter only needs to further parse and check the suspecting rows in $TO\_CHECK$ bit-vector, then add qualified row numbers into $RESULTS$ and suspensive row into $TO\_CHECK$ for the filter of the following sub-column. When the filtering level goes deeper, there will be fewer records that need to be checked. We can even early prune the filter execution in some special cases if there is no item in the $TO\_CHECK$ bit-vector. Even for the cases where early pruning is not happening, progressive filtering can avoid a lot of value parsing and check, especially valuable for the trailing bits sub-column where value parsing is more expensive. Progressive filtering is eligible for most filtering query executions. We describe its detailed implementation for $EQUAL$ and $GREATER$ predicate here. Other predicates such as $GREATER\_EQUAL$ or $LESS\_EQUAL$ can be achieved by logically combine existing predicates, $NOT\_EQUAL$ can be achieved by negating the $EQUAL$ results.

**Equality filtering** predicate $x = C$ can be decomposed as $x_1 = C_1 \wedge x_2 = C_2 \wedge x_3 = C_3 \wedge ... \wedge x_k = C_k$ by query rewriting, which intuitively indicates the equality predicate could hold if and only if equality holds on all sub-columns. This can be formulated as:

$$x = C \iff \wedge_{i=1}^{k}(x_i = C_i)$$

The predicate push-down enables efficient progressive filtering on the sub-columns where we can execute the most selective predicate to maximize the filtering benefits - to filter out disqualified records as much as possible in the early phase. For filtering on the first sub-column, qualified rows are added into $TO\_CHECK$ bit-vector for filtering on the following sub-columns. Data skipping is used in the following evaluation, and only rows in $TO\_CHECK$ are parsed and further evaluated. Disqualified rows are removed from $TO\_CHECK$ bit-vector. In the last sub-column, all qualified rows are added into $RESULTS$ bit-vector as a final result for the filtering predicate $x = C$.

With the previous example in Section 4.3.2, if given an equality filtering predicate $x = 23.1415$, we can extract the bits of bounded range and precision 0011.001001000011100 with the aid of range and precision in the metadata. Then we can rewrite the predicate as $x_1 = 00110010 \wedge x_2 = 01000011 \wedge x_3 = 100$. On the encoded file shown in Figure 4.4, progressive filtering starts with evaluating all values in the first sub-column $x_1$, and row 1 and $n-1$ are qualified with the first predicate, so they are added to $TO\_CHECK$. On the second sub-column $x_2$, only row 1 and $n-1$ need to be checked, and row $n-1$ is disqualified, thus removed from $TO\_CHECK$. On the third column $x_3$, only row 1 is evaluated and qualified, thus row 1 is added into $RESULTS$ bit-vector returned as a final result.

**Range filtering** predicate $x > C$ can be decomposed as a combination of $x_i = C_i$ and $x_i > C_i$ after query rewriting. For example, $x_1 > C_1$ is sufficient to induce $x > C$. While $x_1 = C_1$ indicates further evaluation is needed for $x_2$. Then we proceed to sub-column 2 predicate evaluation. Similarly, if $x_2 > C_2$, we have $x > C$. Otherwise if $x_2 = C_2$, we proceed to sub-column 3. The evaluation is repeated until all sub-columns are processed. It can be formulated as:

$$x > C \iff \vee_{i=1}^{k}[\wedge_{j=1}^{i-1}(x_j = C_j) \wedge (x_i > C_i)]$$

For the first step, all records are parsed and evaluated. Only rows in $TO\_CHECK$ bit-

vector from the previous step are decoded and evaluated in the following evaluation step. There are two categories of records generated for each step: qualified rows (where $x_i > C_i$)) are added to $RESULTS$ bit-vector and unsettled rows (where $x_i = C_i$)) are added to a new $TO\_CHECK$ bit-vector. On the last sub-column, all qualified rows are added to $RESULTS$ bit-vector, which is returned is a final result for the filtering predicate $x > C$. During any filtering step, an early stop is activated when the $TO\_CHECK$ bit-vector is empty and all following bits are skipped. $NULL$ is returned for equality filtering, and the intermediate $RESULTS$ bit-vector is returned for range filtering in this case. We introduced cold start filtering cases where all records are potentially qualified. However, our filtering operators can be applied to the cases where intermediate query results are provided in the form of bit-vector, by using given bit-vector as $TO\_CHECK$ for start sub-column.

### Progressive Aggregation

Byte-oriented float splitting compression also facilitates some aggregation queries with the aid of techniques mentioned above [77]. Query execution could be improved either by data skipping for Min/Max queries or by transforming double operations into integer ones for Sum/Average queries.

**Min/Max** information is stored as metadata in the compressed byte array for each compressed segment. Therefore, efficient metadata lookup is sufficient to answer those queries. However, metadata lookup is not applicable for custom query scope predefined by a filter. Our compression approach also supports fast custom scope query execution, as encoded values are independently and neatly arranged in each sub-column level. Min/Max aggregation queries perform similarly with the equality query discussed previously. Starting with the first sub-column, the query executor finds the greatest/smallest value and puts all corresponding rows into $TO\_CHECK$ bit-vector. The query executor then proceeds to the next sub-column, evaluates all records corresponds to rows belonging to the previous $TO\_CHECK$ bit-vector, and generates a new $TO\_CHECK$ bit-vector if applicable. The min/max value

is assembled in the end and returned as final results after all sub-columns are evaluated. A $RESULTS$ bit-vector indicating the rows with min/max value is also returned along with query results. Min/Max queries are efficient with Buff because only very limited bits are evaluated during the query execution with the aid of progressive filtering and data skipping. In most cases, only a single row is qualified and added into $TO\_CHECK$ bit-vector for the first sub-column. Thus, the following sub-columns evaluation is merely skipping to the target row and assembling the specific single record, which is more efficient than decoding all bits and comparing the decoded values.

**Sum/Average** aggregation queries are very efficient with our compression approach with no full decoding needed. The byte-oriented splitting layout enables fast access to each byte/bits component corresponding to different precision. Rather than decoding every record into float value before applying the summation operation, value is accumulated on each sub-column in the form of small integer until the final summation of the intermediate sum results from each sub-column adjusted by its corresponding exponent and base value. The final summation is returned as *Sum* result or further processed to get the *Average* result. For example, on the encoded file shown in Figure 4.4, we first sum records in each sub-column, and get $sum(x_1)$, $sum(x_2)$ and $sum(x_3)$ respectively. Then we scale the summation with its corresponding basis. In this example, basis for column $x_1$ is $2^{-4}$, $x_2$ is $2^{-12}$ and $x_3$ is $2^{-15}$. Then the final sum result is $sum(x) = sum(x_1)*2^{-4}+sum(x_2)*2^{-12}+sum(x_3)*2^{-15}+B\times N$ where $B$ is the base value (min value) of this encoding block and $N$ is the number of entries. We can further divide by $N$ to get the average. The value is not fully decompressed to the original double value during the query execution, and all the execution is achieved with integer arithmetic, which improves the query performance.

## Variable Precision Materialization and Aggregation

Reduced precision has emerged as a promising approach to improve power and performance trade-offs [22]. Customized precision originates from the fact that many applications can tol-

erate some loss in quality during computation, as in media processing and machine learning. The byte-oriented encoding layout also enables variable precision materialization in addition to full precision required by downstream analytic. For example, consider a system-agnostic query like *printf("%.2f", latitude) where AVG(temp)>90°F*. We can skip reading the trailing bytes/bits for *temp* column if we get *AVG(temp)>90°* with several leading bytes, and only several leading bytes of *latitude* are needed for two digits precision printing. For any reduced precision value read, Buff aligns the requested precision with the least byte boundary covering the required precision. For example, for a given compressed dataset with 4 bits for the integer and 18 bits for the fractional part (supports 5 digit precision at most), Buff materializes 2 bytes ($\lceil(4+11)/8\rceil$) to support 3-digit precision query.

Buff also provides deterministic approximate aggregation query processing on data with reduced precision as suggested by DAQ [77]. Recall previous example in Section 4.3.2, on the encoded file shown in Figure 4.4, we can get deterministic approximate aggregation result by only reading the partial data. Similar to progressive aggregation execution in Section 4.3.4, we can get an sum approximation by reading the first sub-column as $sum(x) = sum(x_1) * 2^{-4} + B \times N$ with a deterministic lower bound $= sum(x_1) * 2^{-4} + B \times N$ where all trailing bits are zeros, and a upper bound $= sum(x_1) * 2^{-4} + (B + 2^{-4}) \times N$ where trailing bits are ones. Similarly for *avg* query, we get a deterministic bound by reading first sub-column as $[sum(x_1)*2^{-4}/N+B,\ sum(x_1)*2^{-4}/N+(B+2^{-4})]$. As we read and process more data, we can provide a more accurate result.

## Data Skipping

Data skipping is another technique used to enhance query performance. After scanning a sub-column, we know some records in the remaining columns do not satisfy the predicate, and we can skip them. The byte-oriented encoding layout enables efficient data skipping in the following aspects: byte-oriented layout guarantees efficient data reading and skipping without handling any cross-boundary issue. The fixed coding length for each component

enables efficient skipping with simple bits length calculations. The trailing bits in the last component are mostly skipped based on the previous evaluation. In some cases, we can early prune the query execution and totally skip checking the following bytes and trailing bits if no records need to be further checked. Besides, data skipping enables fast records access for custom row-level scope queries, such as queries on a given time interval for time series data. Data can still be skipped efficiently without parsing the irrelevant entries.

### Adaptive filtering

Different from prior works that use SIMD to accelerate query execution [45, 91, 76, 30], Buff uses a workload-adaptive execution strategy. When SIMD is available, Buff will use SIMD as much as possible. However, SIMD is not always the best when compared with the scalar progressive filtering. SIMD is usually good on low selective filtering, where a large percentage of records are evaluated. While progressive filtering is extremely efficient on high selective filtering tasks. For each sub-column, Buff will choose either the filtering strategy based on density ratio $r = \#candidates/\#records$. By default, SIMD execution is used for the first sub-column unless a sparse bit-vector is provided as input. For the following sub-columns, Buff uses SIMD execution if the density ratio is high and progressive filtering otherwise.

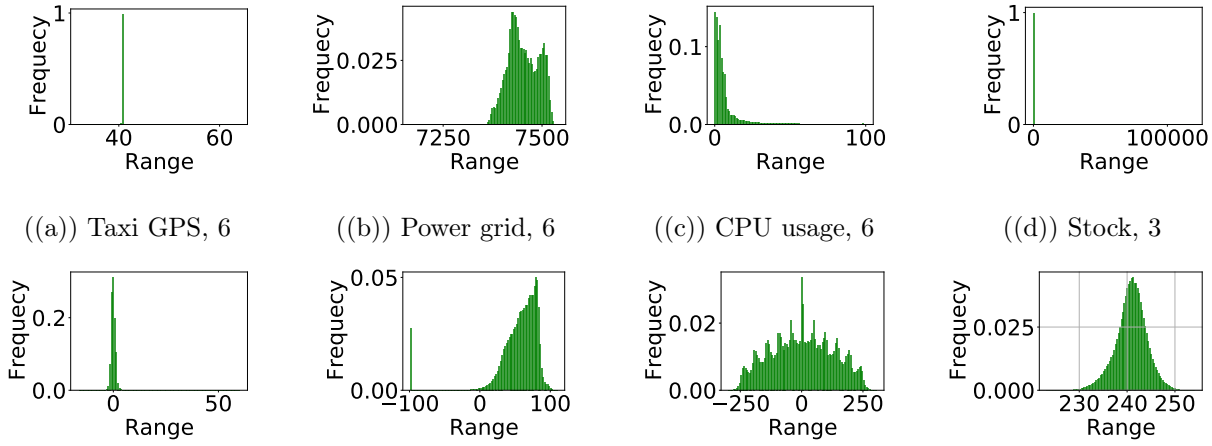
We perform a sensitivity analysis to find the crossover point. Progressive filtering takes more time as the density ratio increases and crosses over the constant SIMD cost at a density ratio 0.06, which is the current SIMD threshold used in Buff's adaptive filtering. The threshold number 0.06 means progressive filtering covers many query predicates since the average item frequency for sub-column is $1/2^8 \approx 0.0039$, assuming a uniform distribution. And for other highly skewed cases, sparse encoding helps in query execution. Please be noted that, the SIMD threshold changes on different hardware.

## 4.4 Experiments

Compression approaches should keep the compressed data size smaller and support fast query execution, including fast filtering and aggregation queries. This section applies the compression techniques mentioned above on various datasets covering common applications with bounded float and reports both compression and query performance. In addition, we also include end-to-end complex query evaluation on Time Series Benchmark Suite (TSBS) [5].

All experiments were performed on servers with 2 Intel(R) Xeon(R) CPUs E5-2670 v3 @ 2.30GHz, 128GB memory, 250GB HDD, and Ubuntu 18.04. All our implementations and experiments are done in Rust 1.49.0 and we use AVX2_m256i for SIMD. To use ByteSlice for floats we re-implement ByteSlice in Rust, also implement Gorilla and Sprintz in Rust. For Gzip and Snappy, we use rust flate2 and parity-snappy libraries respectively. We apply Gzip level-9 in our experiments. All our implemented baseline achieve their claimed throughput (e.g., Snappy: *250MB/s* Gzip: *20-50MB/s*, Sprintz: *200MB/s*).

### *4.4.1 Datasets*




((a)) Taxi GPS, 6 ((b)) Power grid, 6 ((c)) CPU usage, 6 ((d)) Stock, 3

((e)) UCR, 5 ((f)) Temperature,1 ((g)) Current, 5 ((h)) House, 2

Figure 4.6: Datasets have bounded range and precision

Our dataset covers various applications: stock data [2], taxi data from NYC (GPS) [7],

DevOps monitoring (CPU) [81], city temperature (TEMP) [51], humidity [17, 18], power grid (PMU) [83], magnet motor (Current) [8], household electric power consumption (House) [3] and machine learning [68]. As most datasets are shared using a *txt* or *csv* format, we can easily extract the precision by counting the fractional digits. As we target a compression technique that would work with application code or a database, we evaluate an ingestion process that sends data in the IEEE 64-bit float format which is used by many applications and compression evaluations [89, 59, 22, 19, 23, 75]. Figure 4.6 shows the data range, distribution and scale (precision) for representative measurements of those datasets. The x-axis shows data range including all outliers if any. For GPS we use the latitude attribute. The data distribution varies on different measurements. Those datasets cover data with different precision with/without outliers. Many datasets have a very skewed distribution, where most records are clustered around a tiny area. As an example, most data points in NYC's GPS dataset fall into the city area with coordinate between *(40.702541, -74.007347)* and *(40.793961, -73.883324)*. The high decimal position with scale *6* guarantees sub-meter level precision of the data points. Similarly, all these measurements have limited range and bounded precision. In addition to those read world datasets, we also generate datasets from TSBS. Here, we use the IoT dataset with a scale 1000, which includes 9.1 billion data points about tracking information for trucks.

### *4.4.2 Optimized Baselines*

During our evaluation, we found that the idea of bounded precision can improve Gorilla. Therefore, we provide an optimized version of Gorilla for comparison. In addition, we include fixed-point with bounded precision and ByteSlice variations for float as baselines.

#### Optimizing Gorilla

When we apply Gorilla on our datasets, we get a poor compression performance for most datasets. For datasets, such as CPU and Temperature, the compressed size is close or even

larger than the original size. After detailed analysis, we notice that many data points in these datasets fluctuate over value 0 or $\pm 2$. There are two reasons why Gorilla is sub-optimal on datasets covering those values. First, float numbers fluctuating around 0 flips the sign bit (the leading bit). Also, when floats fluctuate around an absolute value 2, they flip the leading bit of exponent bits as exponent value 1 is encoded as 10000000 in a 32-bits float. Second, high precision floating numbers make the mantissa bits sensitive to the value change, especially for those less significant bits. For these two reasons, Gorilla fails to perform efficient trailing zero compression and performs poorly on these datasets.

To improve the compression performance of vanilla Gorilla, we have two different directions to work on:

- Leverage the leading zeros by offsetting the input numbers to avoid the "pitfall" value points.
- Use bounded precision float for the input numbers to eliminate the less significant bits in the mantissa part.

We verified those two solutions on our datasets. We found the former solution improves compression ratio performance slightly but at the cost of slowing down the compression throughput significantly, since we have to offset the value before we can apply gorilla encoding. However, the latter solution significantly improves compression performance. In the following evaluation section, we include the latter Gorilla version ( GorillaBD). Nevertheless, GorillaBD only improves the compression performance and can do nothing to speed up the query execution because of the complex variable encoding/decoding mechanism inherited from Gorilla. This motivates us to devise a new compression for floats.

### Bounded Fixed-point

As discussed in Section 4.2.1, fixed-point is not precision aware, since it uses every bit to approximate the value no matter what precision is needed. Additionally, fixed-point is not

optimized for custom ranges, since it represents a range $-2^I \sim 2^I$. In order to include fixed-point into our evaluation, we implement bounded fixed-point (as *FIXED*) as a baseline with bounded precision applied to achieve just-enough precision and delta encoding used to offset the custom range. In FIXED, we scan the file and encodes the datasets with bounded fixed point representation.

### ByteSlice variations for float

ByteSlice is designed for an input code of a bit-packed integer, which is not directly applicable to float numbers. In order to apply ByteSlice on floats, a conversion from the integer code is needed to meet the ByteSlice input requirement. We can either borrow from Buff to extract the aligned bit representation of float (**BUFF-SLICE**) or scaling float into integer (**SCALED-SLICE**), and then bit-pack those code into ByteSlice. We compare against both ByteSlice variations.

## *4.4.3 Performance Overview*

The radar chart of Figure 4.7 shows the overall performance of all float compression approaches above. We normalized the performance results into a range between 1 and 10 on each dimension uniformly, and then take the average of all workable datasets (Sprintz and SCALED-SLICE fail on some dataset which are not included here). ByteSlice variations do not provide a specialized aggregation solution, so we materialize before max and sum operation. For all metrics here, higher is better. As we can see, the state-of-the-art Gorilla approach does not work well on those datasets. Sprintz and FIXED perform well in terms of compression ratio and materialization performance on those datasets while performing poorly on other performance dimensions. ByteSlice variations are good on materialization performance but sacrifice compression ratio because of bit padding. Buff performs best in terms of query and compression throughput. In addition, it handles outliers data quite well and achieves a better compression ratio on outlier-included datasets (e.g., GPS, Stock

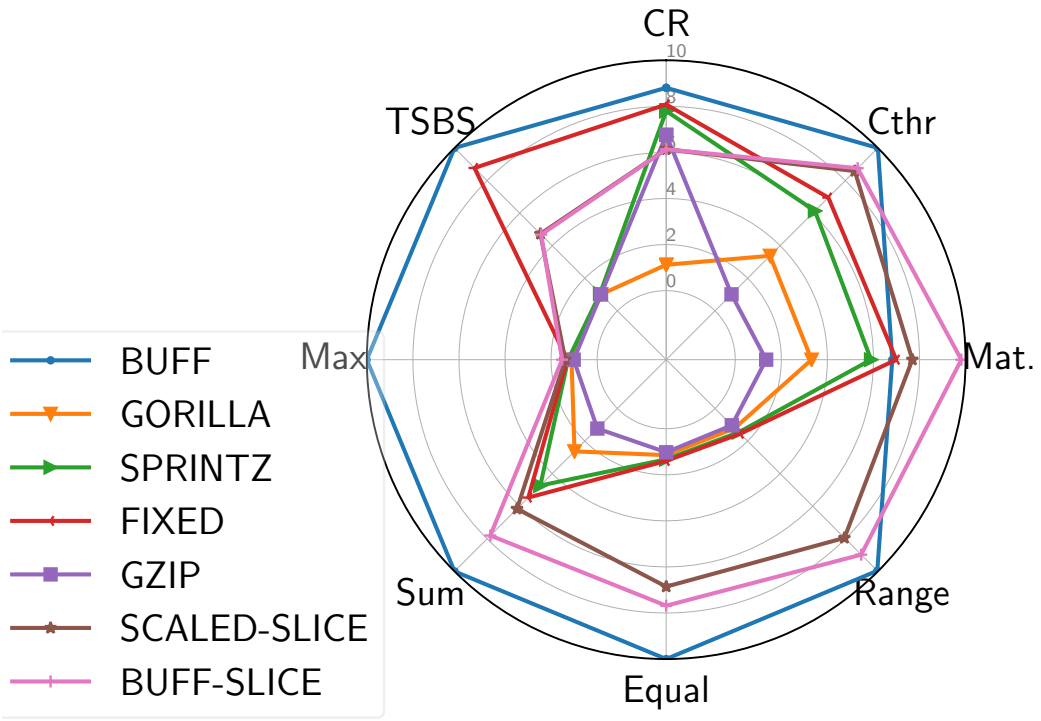



Figure 4.7: Compression performance overview (greater is better, CR: compression ratio, Cthr: compression throughput). Buff outperforms in query and compression throughput.

dataset). For some query types, such as min/max and high selective equality filtering, Buff is up to 50× faster than other compression baselines. General-purpose approaches (e.g., Gzip and Snappy) perform poorly in these dimensions.

### *4.4.4 Compression Performance*

We evaluate both the compression ratio and compression throughput on our datasets and benchmarks. A good compression must be fast and effective.

#### Compression Ratio

Figure 4.8 shows the compression ratio for each compression approaches. Gorilla compression performs poorly on most of our datasets as there are very limited repeated or similar adjacent values. GorillaBD works better than Gorilla since many trailing bits are formatted

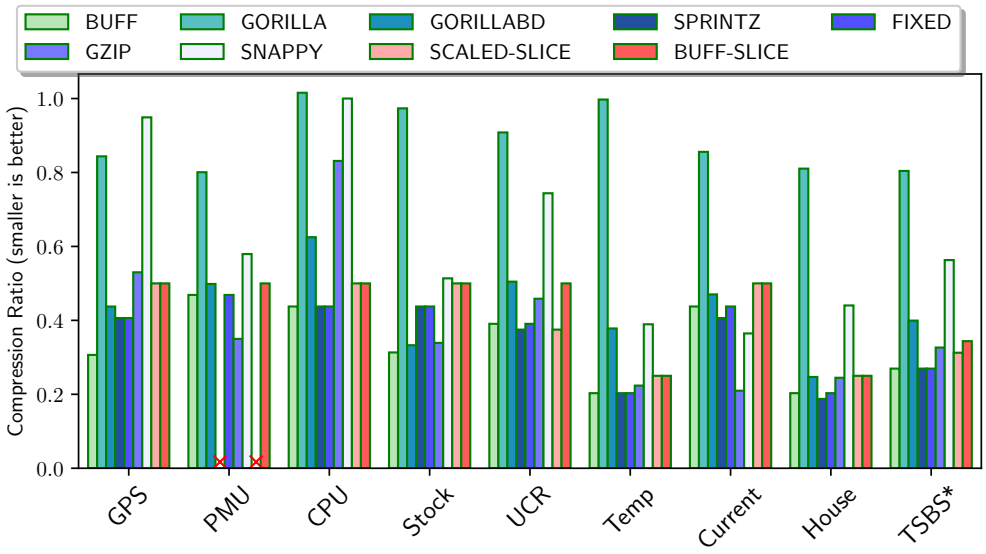



Figure 4.8: Compression ratio shows the Buff are comparable to the best and always better than vanilla Gorilla (Sprintz and Scaled-slice fails on PMU).

by bounded precision float, resulting in a better compression ratio. Buff performs similarly with Sprintz and FIXED since they leverage the bounded precision and range to map the numbers into almost the same finite encoding space. Value range and value precision determine the number of bits needed for the integer and the fractional parts. The slight compression ratio difference between Sprintz and Buff comes from the dividing of bits, which wastes some code space when the bits are not fully used. However, this brings a huge query benefits, which we will show in Section 4.4.5. In addition, as mentioned in Section 4.3.3, Buff can detect outliers and enables sparse encoding to compress the leading byte column further, resulting in a better compression ratio on GPS and Stock dataset compared with Sprintz and FIXED. ByteSlice variations pads the trailing bits for better query performance, so it is less effective than Buff, FIXED, and Sprintz on most datasets. The Byte-oriented compression approaches compress data by searching common sequences at the byte level, so Byte-oriented compression performs efficiently on temperature datasets because of the low cardinality brought by low data precision but performs poorly on datasets with less repeated values.

### Compression throughput

In addition to the compression ratio, we also evaluate compression throughput. We count the time, including loading data from memory, applying compression, and writing back to memory. Figure 4.9 shows compression throughput for all float compression approaches. The leftmost bar corresponds to Buff with/without user-provided range stats. With ranges given by a user, our method can skip the range checking step in the float decomposing step and achieve higher compression throughput. We refer to it as "Fast-Buff" as is indicated by the star marker bar with the light green bar as a base. Overall, Buff outperforms all its competitors on most datasets. Gorilla relies on $XOR$ between two adjacent values, then dynamically decides the number of leading and trailing zeros, and writes the middle residue bits. Gorilla's complicated encoding mechanism and variable encoded length limit its encoding throughput.

Buff is faster than FIXED and Sprintz because of its fast byte-oriented writing. According to profiling, Buff byte-oriented writing is $6\% \sim 12\%$ faster than writing bits. Sprintz for float also relies on arithmetic multiplications to quantify input float values into integer values, which causes overflow issues such that Sprintz and SCALED-SLICE fail on the PMU dataset. Overall byte-oriented compression is slower than Buff and Sprintz. Snappy achieves the highest compression throughput on the CPU dataset, but Snappy performs poorly on the CPU dataset in the previous compression ratio experiment, where no compression is achieved at all. According to previous experiments, Gzip compression provides a higher compression ratio but uses more CPU resources than Snappy, thus lowering compression throughput. Compared with ByteSlice variations, Buff writes fewer bits but pays more overhead on writing the last trailing bits. Therefore, we can see very similar compression throughput on most datasets for both approaches, but Buff is faster on GPS and Stock datasets where more space is saved by sparse encoding.

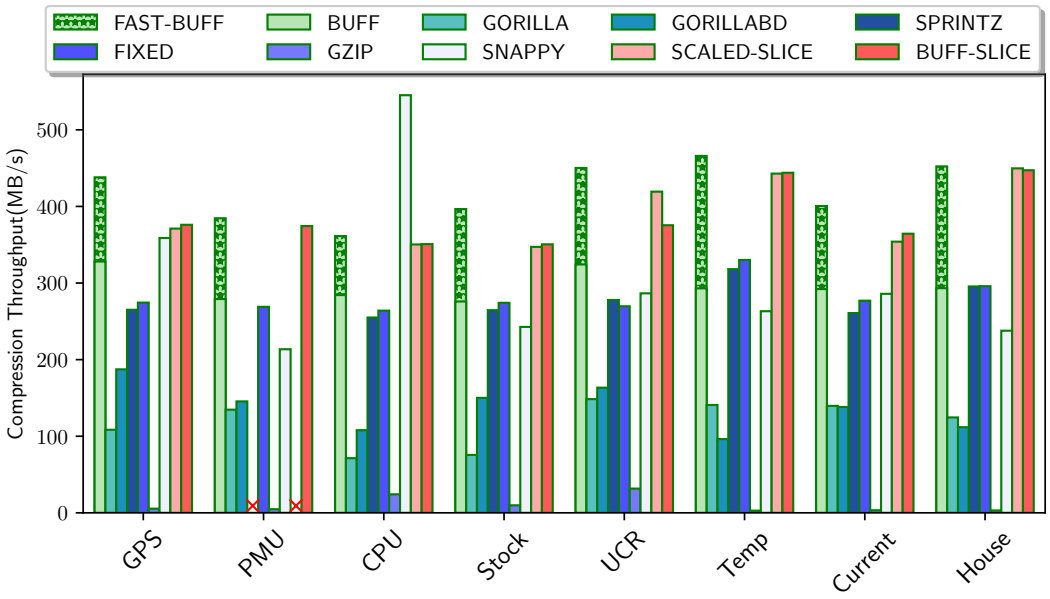



Figure 4.9: Compression throughput shows Buff performs best in most cases with user given range stats.

### *4.4.5 Query Performance*

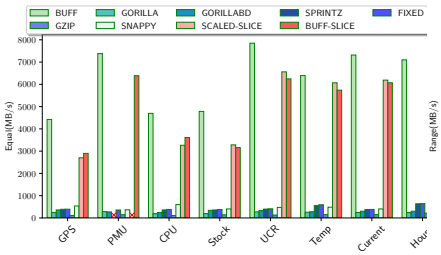


Figure 4.10: Low selective equality filter

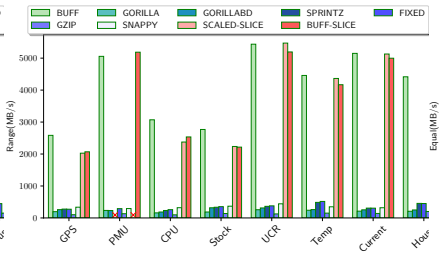


Figure 4.11: Low selective range filter

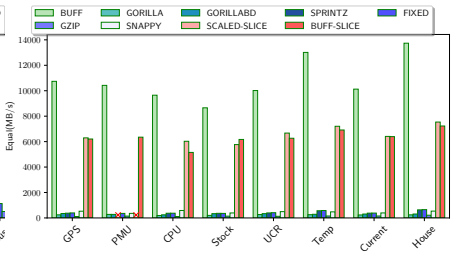


Figure 4.12: High selective equal filter

We also evaluate decompression and query performance. We measure execution time, including reading the data block from memory, parsing data, and evaluating the query. For all queries evaluated, we leverage each compression approach's characteristics to speed up its query as much as possible. Techniques, such as query rewriting, early pruning, and progressive filtering, are used for each approach whenever feasible, as discussed in each experiment section. We use the most frequent value (lower selectivity) in the corresponding dataset as a predicate operand to conduct a fair comparison for all equality filtering and range filtering. Progressive filtering is less efficient with low selectivity predicate as fewer

values are skipped during query execution. We also run high selectivity queries to show Buff's potential. SCALED-SLICE and Sprintz bars for PMU dataset are missing as they fail on PMU dataset because of an overflow issue.

### Filtering

We include filtering performance evaluation with query rewriting, early pruning, progressive filtering, data skipping and SIMD execution enabled when possible. For Buff, instead of materializing all records before applying the filtering predicate, we rewrite the query into query predicates combination on all sub-columns, then apply the translated query predicate on each sub-column sequentially. We also use adaptive filtering and data skipping with the aid of intermediate bit-vector results. For Sprintz and FIXED, we apply query rewriting techniques to avoid decompression overhead. For ByteSlice variations, we adopt their SIMD solution for filtering queries.

Our first experiment set shows low selective equality filtering performance. In Figure 4.10, Buff achieves outstanding performance compared to all the other competitors. This speedup is mainly from adaptive filtering and data skipping. As we execute the predicate on the first sub-column, only a tiny part of the input records are qualified and need to be further checked in the subsequent sub-column predicates. For most datasets, less than 5% of entries are qualified and need to be further checked after filtering on the first sub-column with Buff. However, for GPS and Stock datasets, more than 90% entries are still qualified because of code space inflated by outliers. As we hit the frequent value in this case, we can skip decoding the first sub-column by directly using a bit-vector from sparse encoding, which speeds up the query. Sprintz and FIXED filtering use query rewriting to avoid record decompression as well. Compressed bits are extracted and compared with the translated to the target directly by using integer arithmetic. ByteSlice variations are always the second to the best approaches due to the SIMD speedup and early stopping. Buff has a better data and code locality, as it works on the same predicate for the current sub-column. While ByteSlice needs to load

all sub-columns and jump to different sub-column with corresponding predicate evaluation in each iteration. Additionally, always using SIMD can slow down ByteSlice's throughput as a single match , resulting in SIMD loading and evaluating for all adjacent records within the same SIMD word length. In Buff, data skipping can efficiently jump to the target record and avoid unnecessary data loading. Except for these aforementioned approaches, decompression is required before predicate evaluation on float numbers. General-purpose byte-oriented compression approaches are sequential algorithms and need to decompress the last bits before the original data can be accessed. Gorilla encodes value with different bits and variable bits representation, which impedes query translation and data skipping. These bring a high latency for filtering the compressed data with those methods.

Figure 4.11 shows the low selective *greater than* range filtering experiments with the same operand. As we can see, the range filtering performs similarly to equality filtering but with smaller throughput overall. The selectivity of the range query is much lower than previous equality filtering. There are more qualified records in this case, which triggers the *IF* branch that edits the bitmap more frequently. Thus filtering throughput is deteriorated compared with previous equality filtering experiments. Similarly, ByteSlice variations is slower than equality filtering because of more comparison and less early stop. However, Buff still performs best. Similar to equality filtering, all other approaches require full decompression. As we can see from the figure, their filtering performance is proportional to their materialization performance, which we will discuss later in Section 4.4.5. Figure 4.12 shows high selective equality queries that filter out majority of records. Buff is more efficient on the high selective query as more bits are skipped by progressive filtering. Overall, Buff achieves 20× for average query speedup compared with the second-best float compression approach. Buff has similar trends on high selective range queries, and it achieves 16× on average query speedup compared to the second-best float compression method (we omit the figure).

## Aggregation

An aggregation query is another frequent query for numeric data analysis. We evaluate the query performance of *min/max* and *sum* for all compression approaches.

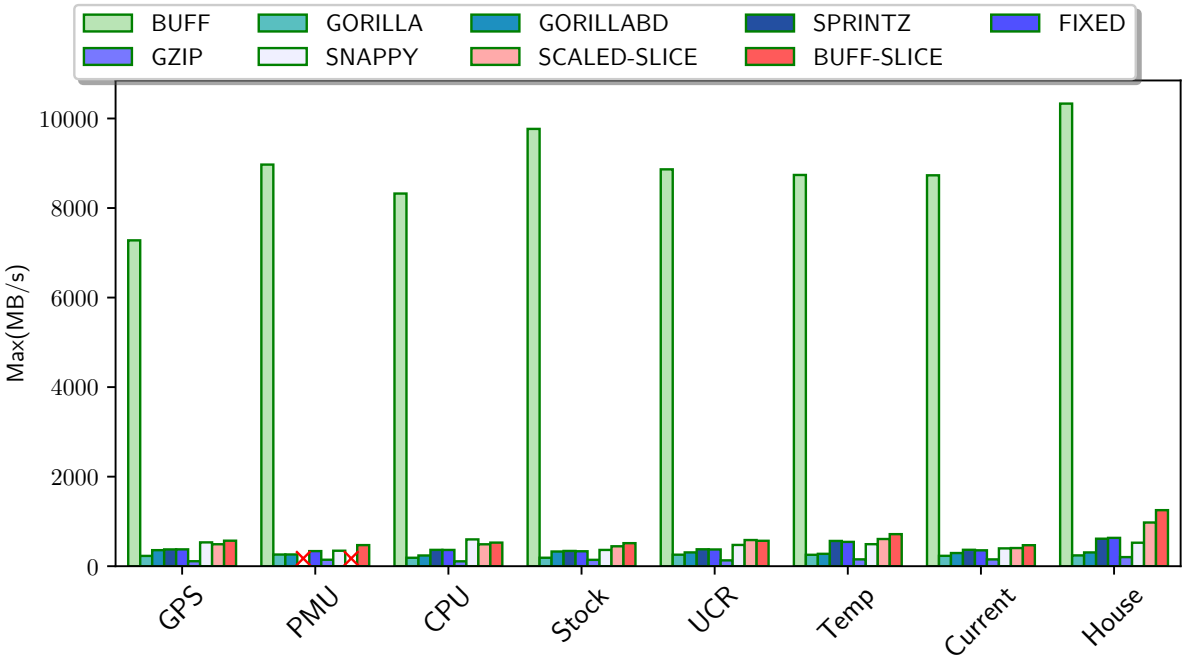



Figure 4.13: Max throughput

Figure 4.13 shows *min/max* query throughput. The query time includes bit parsing and query execution. Even though we can quickly extract min/max value from data statistics in the metadata block for Buff, we force aggregation query execution on compressed data for a fair comparison, and this is necessary for aggregation with a filter. In addition to this setting, we enable all the techniques used in previous experiments. For *min/max* aggregation, we can use progressive filtering and data skipping for Buff. For FIXED and Sprintz, we find the *min/max* integer code directly then convert this value into float numbers. For other approaches, float numbers are decompressed for evaluation. The result shows that Buff achieves 22× on average query speedup than the second-best float compression approaches Sprintz or FIXED. The *min/max* value is usually with a low frequency such that only a tiny portion of entries are added to the bit-vector and need to be checked in the following sub-columns. This makes Buff extremely efficient for *min/max* aggregation queries since a

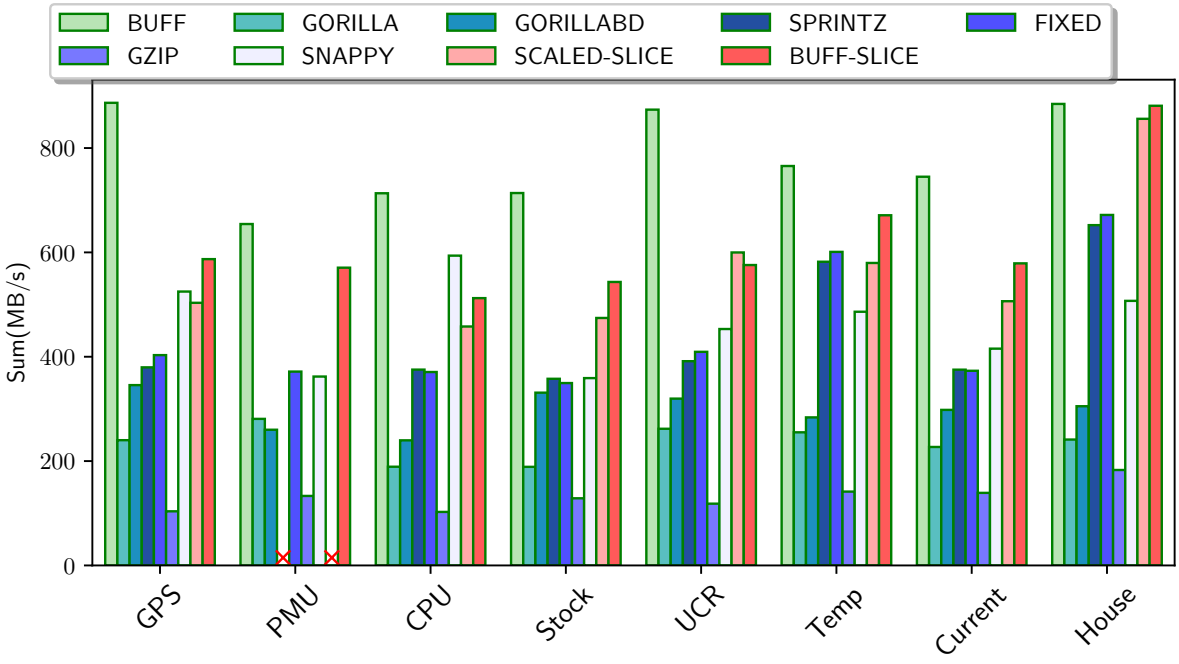



Figure 4.14: Sum throughput

huge amount of entries (bits) are skipped during the evaluation step.

The sum query performs similarly to materialization as all compressed data has to be decompressed, and all techniques mentioned previously are ineligible in this scenario. Figure 4.14 shows the *sum* query throughput on the same dataset. Buff outperforms all its competitors. For Sprintz, FIXED and Buff, we use lazy materialization to avoid the overhead of full decompression for each entry. The sum intermediate result is calculated with an integer representation for Sprintz and FIXED, then converted back to float numbers at the very end. This is similar to sum operator with Buff, where the sum is calculated on each sub-column with its integer representation respectively and concatenated into a final float-point result in the end. The performance difference between those approaches comes from the more efficient reading and parsing of Buff and no data dependency between adjacent numbers. All other approaches perform poorly since full decompression is inevitable.

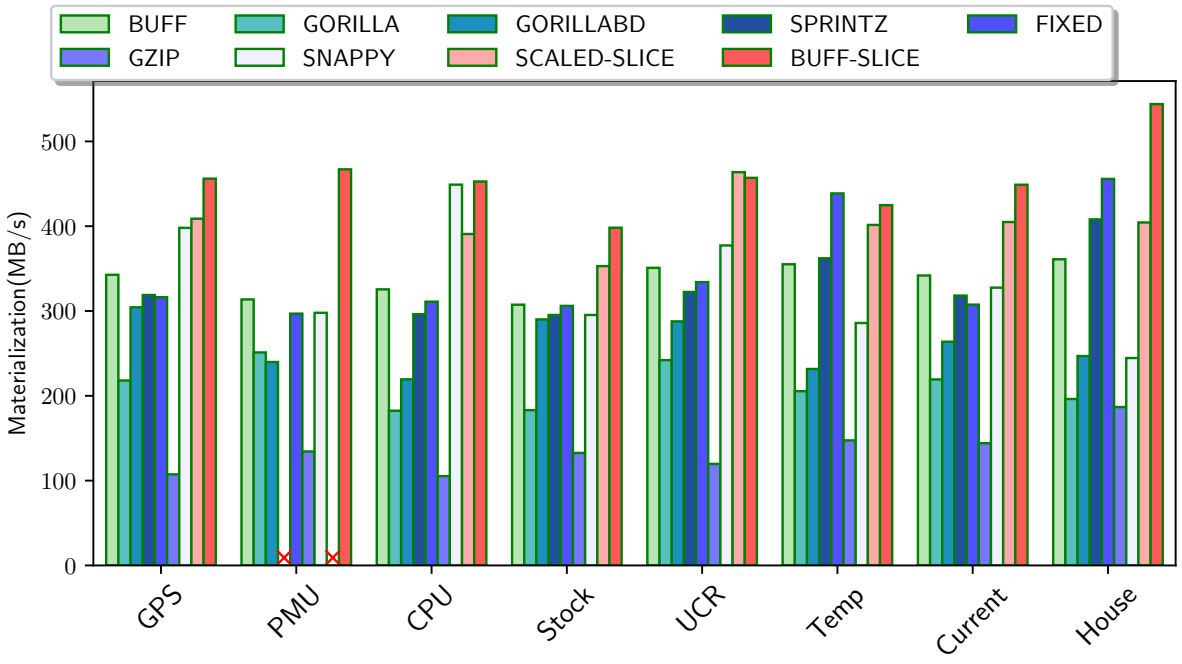



Figure 4.15: Materialization throughput

## Full Materialization

Materialization performance varies a lot for each compression approach. Overall, materialization overhead is comprised of bit parsing time and assemble time. Fixed-length encoding helps for bit parsing. As is shown in Figure 4.15, Sprintz and FIXED perform better than Buff as they read single fixed-length data chunk during materialization, which is faster than Gorilla that reads variable-length bits for each record, and Buff where multiple data chunks need to be fetched as requested, then parsed and concatenated into a float number. However, Buff is still comparable with Sprintz and FIXED thanks to the efficient reading and parsing of Buff. ByteSlice variations are faster than Buff because of its fast byte reading and decoding benefits. Since Buff bit extraction is more efficient than the scaled version on decompression, BUFF-SLICE is always faster than SCALED-SLICE. Gzip is the slowest among all candidates since it has to decode Huffman encoding before restoring the recurring sub-sequences. Snappy achieves a better performance than Gzip because it skips Huffman encoding for higher throughput.

## Materialization and Aggregation with Requested Precision

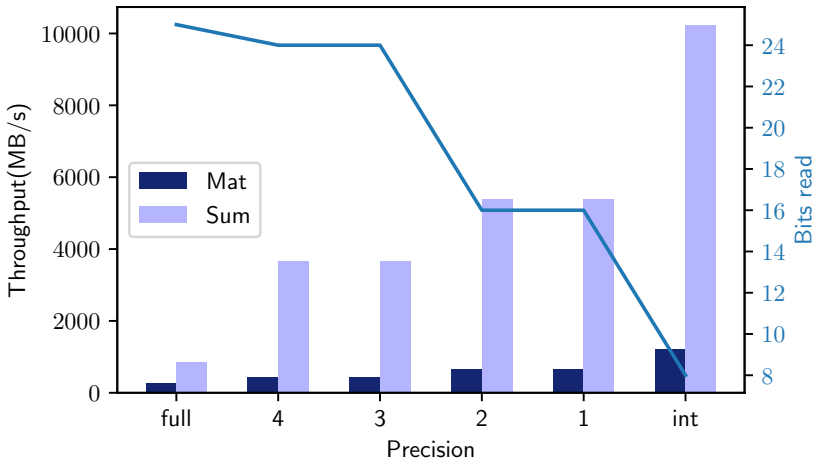



Figure 4.16: Materialization and aggregation with targeted precision boosts throughput by only reading the leading bytes.

We also show the versatility of Buff on variable precision materialization and aggregation support, which are especially useful when users only care about limited precision in formatted output and fast estimation of aggregation queries. Figure 4.16 shows the materialization and aggregation with requested precision on the UCR dataset. The red line-based right y-axis shows the bits needed to support the requested precision. The query performance linearly increases as bytes read decreases. A single expensive bit read can significantly slow down the query performance, as we can see from the first set of results on full data with 3 bytes and 1 bits read.

## 1-Nearest Neighbor

One nearest neighbor (1NN) is a popular machine learning tasks where an entry $X$ is assigned to the nearest neighbor's class. In addition to SQL tasks, We also include 1NN experiments to explore the impacts of Buff on 1NN accuracy.

We run 1NN task on the UCR dataset collection with different Buff precision for the input float numbers, and report the 1NN accuracy result for all 128 datasets. For better represen-

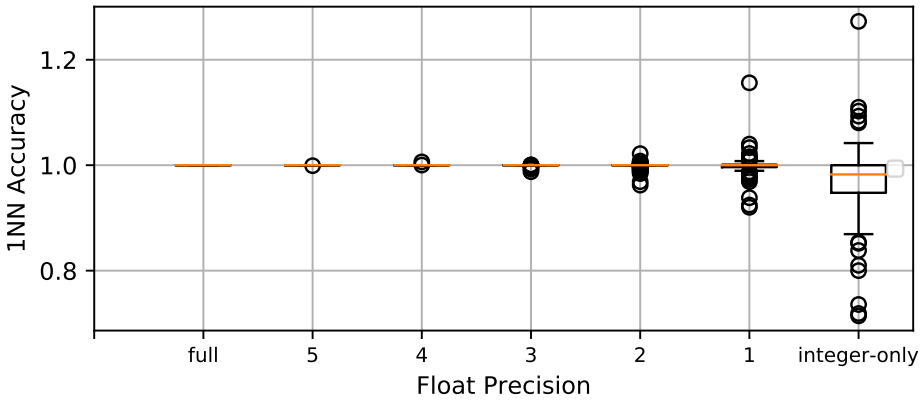



Figure 4.17: Relative 1NN accuracy on 128 UCR datasets shows reduced precision has a limited impact on accuracy.

tation, the accuracy number for each dataset is normalized by its corresponding accuracy of full precision input. As is shown in Figure 4.17, relative accuracy decreases gracefully as the Buff precision goes down. Nevertheless, we still achieve outstanding performance even with reduced precision greater than one. This experiment shows Buff has a good potential of saving storage and network transmission cost for some machine learning tasks without loss of precision.

### *4.4.6 Benchmark Evaluation*

In this set of experiments, we show complex query performance with TSBS. On TSBS, we generate “last-loc", “low-fuel" and “high-load" queries, which respectively include projection on longitude and latitude attributes (project), filtering on a fuel_state attribute containing single sub-column with Buff encoding (range-single), and filtering on current load attribute (range). In addition to the float-related operations, those end-to-end queries also include any filter, join, sorting on timestamp, string, or integer attributes. The runtime for these columns is independent of float compression approaches, so we take do not show them and only focus analysis on query runtime of float attributes. For Gorilla, the float-associated cost makes up 54% to 76% of the total runtime. However, Buff can diminish those costs to 1 or 2 orders of magnitude smaller on those queries. Figure 4.18 highlights the query runtime

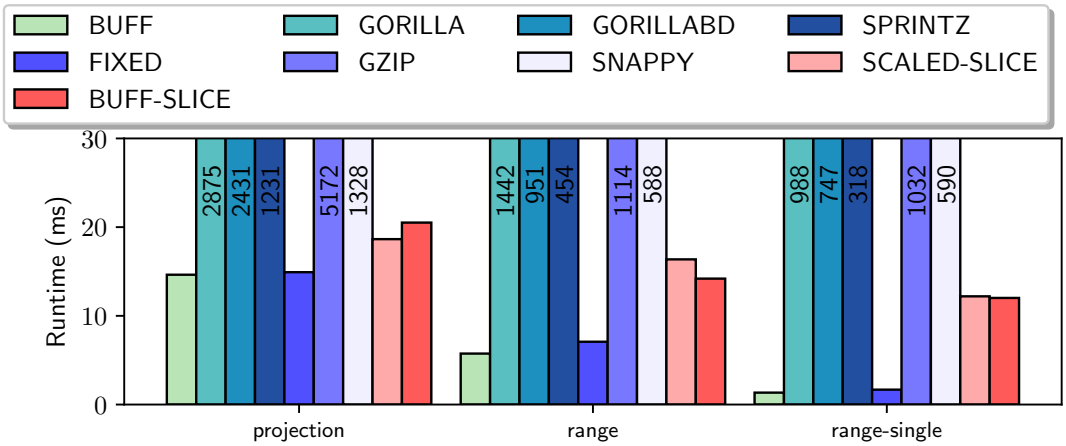



Figure 4.18: TSBS query runtime for float attributes.

on float attributes, Buff always performs the best over all other baseline and is faster than ByteSlice variations on TSBS range filter queries since adaptive filtering chooses progressive filtering execution for the highly selective filter.

## 4.5 Conclusions

This paper proposes Buff, a novel decomposed float compression for low precision floating-point data generated everywhere and every day. Buff uses "just-enough" precision for a given dataset, and leverages the data distribution feature (range, frequent value) to simplify its encoding mechanism and enhance the compression performance. The compression supports efficient in-situ adaptive query on encoded data directly. In addition, Buff provides fast aggregation and materialization for different precision levels. Buff's precision bounded technique is also applicable to the state-of-the-art Gorilla method and improves Gorilla's compression performance significantly. Our evaluation shows Buff achieves fast compression and query execution, while keeps comparable compression ratio to the best approaches. Future work includes studying the performance of Buff for common downstream complex analytical tasks, such as classification [73], clustering [70, 71], and anomaly detection [15].

# CHAPTER 5

# ADAEDGE: A DYNAMIC COMPRESSION SELECTION FRAMEWORK FOR RESOURCE CONSTRAINED DEVICES

IoT systems generate a massive amount of heterogeneous data from various sensors every day. The IoT data from a signal sensor can change over time in terms of data statistics. Therefore, it is impossible to provide a one-size-fits-all compression solution for all IoT data. An adaptive compression selection strategy is required to select the optimal compression for the incoming IoT data based on the data statistics, query workloads and limited resources available on the host. The edge node is a good component for compression tasks in IoT systems because of its closeness to the data source. Compressing the IoT data in an early stage of the generation saves the network and storage resources of the whole IoT system. However, the edge node has limited resources and more constraints compared with the cloud server. This includes the limited storage, bandwidth, and power budget.

In our proposed project, we vision AdaEdge as an adaptive hardware conscious compression selection framework for resource-constrained devices. AdaEdge can select the best compression according to system resource limitations, incoming data statistics, and query workload.

## 5.1 AdaEdge Overview

As we can see from Figure 5.1, There are several limitations and constraints in AdaEdge. The first one is the signal generation rate from the sensor, which ranges from zero to millions of data points per second. In AdaEdge, we assume no back control to the sensors regarding the data generation rate, so the ingestion rate is a hard constraint for our framework. All selected compression must be able to handle the given ingestion rate from the sensors.

The network bandwidth is another hard constraint for AdaEdge. The network bandwidth varies depends on the network connection technology. For a cellular network, the bandwidth

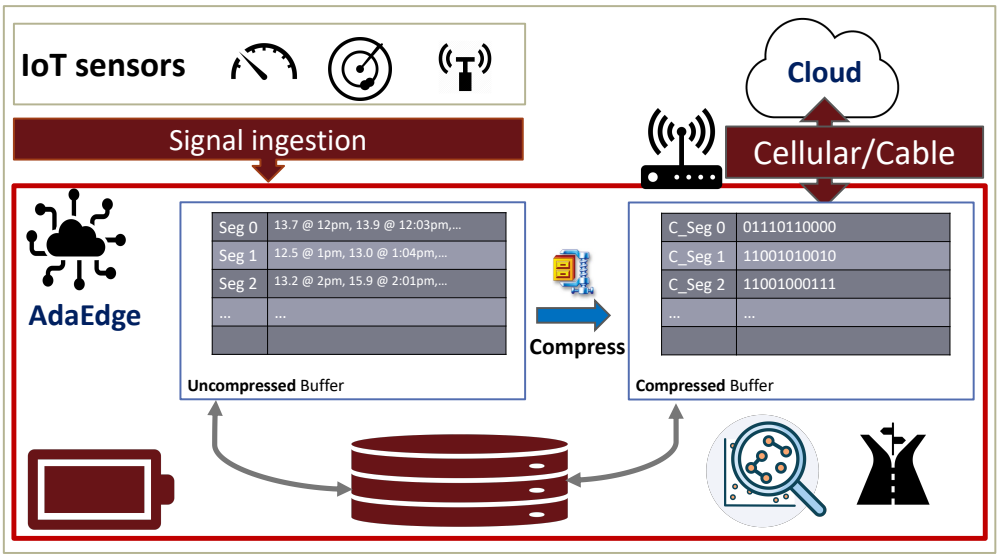



Figure 5.1: AdaEdge overview

changes from 0.01 Mbps to 200 Mbps in practice. In addition, the network bandwidth changes significantly in cases of the network congestion or the edge device movement.

Edge node has a very limited storage space compared with an advanced cloud server. The storage device on edge can be SSD or HDD according to the system design. Storage device are carefully accessed on edge device. For the sake of device life time and power consumption, some systems are designed to avoid using those device for common operations but only for special use during offline cases.

The last limited resource on the edge is power. Many edge devices are designed for demanding environments and deployed in remote area without sustainable power supply. When the battery becomes the only power supply for a given edge node, any task running on the edge should consider the power consumption.

Based on those constraints, we classfy the IoT use case into three modes. In terms of network connection, we introduce online mode and offline mode for AdaEdge. For the sake of enery efficiency, we introduce power-saving mode.

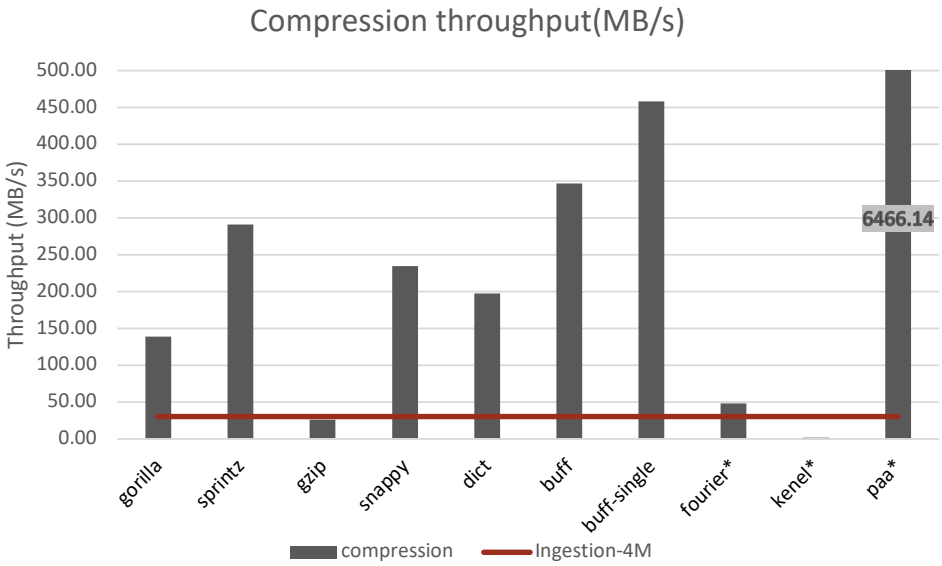



Figure 5.2: Compression should be able to handle the given signal generation rate. Example shows a signal with 4 million data points generated per second.

### *5.1.1 Online mode*

The online mode views the edge node as a data hub, and the goal is to transfer the signal data as much as possible within the given constraints, including signal rate and network bandwidth. The selected compression should be able to handle the given signal generation rate, and the compressed file size should be within the network transmission capacity. We get candidate compression approaches (lossless and lossy) with signal and network constraints.

Figure 5.2 show an example with 4 million data points generated per second. The line shows the ingestion rate (the size of data generated per second) and the bars are the ingestion rate (the size of data compressed per second at full speed) for different compression approaches. We can see from the figure that most compression methods are qualified to compress the signal example except for Gzip and Kernel methods which are usually slow on compressing the data. As we can see from Figure 5.3, The first bar is the original 4M data points. Without compression before data transmission, it is impossible to transfer the ingested data to the cloud with the given network setting. But if Assuming we use 4G here, then many lossless compression such as Sptintz, BUFF, dictionary encoding, and lossy com-

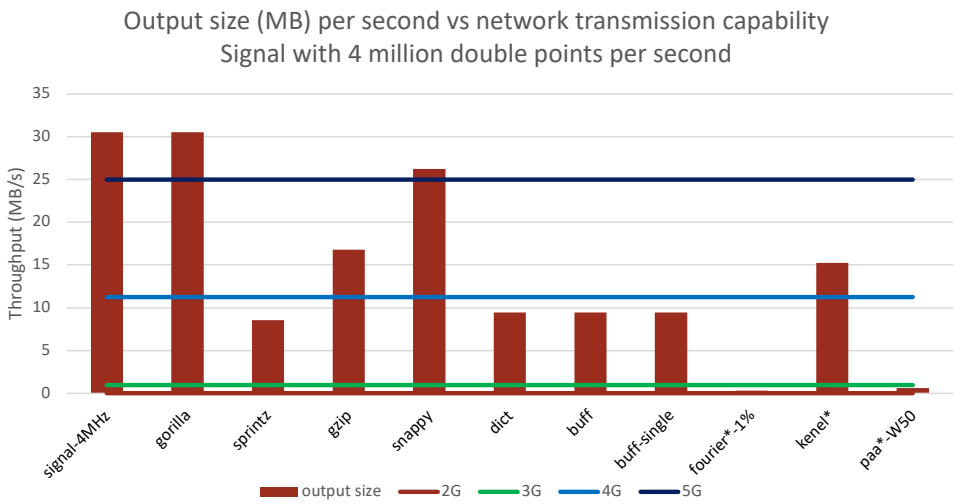



Figure 5.3: Compressed file size should be within the network transmission capacity. Example shows the compressed size of 4 million data points compared with network transmission capacity per second.

pression can send out the compressed data under the 4G network. However, if with the 3G network, there is no qualified lossless compression. So depending on the query workload, we need to choose corresponding lossy compression to minimize the query error rate.

For the online mode, we keep all data ingested and compressed in memory until we send out those segments through a network protocol. We omit saving data to the local storage to save storage space for future offline cases, save power and prolong the lifespan of the storage device. We choose the best lossless compression by default. In special case where there is no qualified lossless compression, we chose lossy approaches optimized for target workload.

### *5.1.2 Offline mode*

For use scenarios with poor network or no connection, the edge node serves as a local computation and storage node. The goal switches to keep ingested data/information as much as possible for data offloading if a future connection is expected.

Offline mode performs differently from the online mode that always optimizes for the compressed size. Instead, load shedding is needed when local storage runs out of space. For all ingestion data, we use the local disk as temporary storage. When there is not enough

space, we do more aggressive compression (even lossy compression) on less valuable or less informative segments instead of removing some segments.

The informativeness can be measured by the query usage of the given segment. A query counter for each segment is a straightforward solution for measuring the informativeness of segments. However, the informativeness should also reflect the contribution of each segment to the query. For example, segment A with 1% qualified entry should be less informative for segment B with 99% qualified entry. The cluster centroid should be more informative than a normal vector for the distance measure. The informativeness definition is still in progress.

### *5.1.3 Power-saving mode*

In the power-saving mode, we optimize for the most power-efficient compression in addition to meeting other constraints. We choose the most power-efficient compression approach from the qualified candidates from other constraints. We can also use a good scheduling policy for the compression task to optimize power consumption further [50, 63]. We can batch the segments before starting the compression process until we finish all compression tasks, then go back to idle for energy saving [48]. Another strategy is instead of applying compression by default, we lazily compress the informative segments and upload data. The segments with no query access or low information will be dropped directly.

**Current Progress:** We currently have the AdaEdge framework implemented with a dual buffer pool for uncompressed and compressed buffer. AdaEdge has basic signal ingestion, data storage, query functionality for the incoming signals. AdaEdge supports 9 lossless compression and 3 lossy compression approaches with optimized query operators. AdaEdge is able to estimate the compression performance based on the data statistics.

## 5.2 Research Plan

We outline our research plan and milestones for the AdaEdge project as below.

**Nov.2021 - Jan.2022**: Implement AdaEdge framework with dual buffer (uncompressed buffer and compressed buffer) and configurable client (signal generation), Compression and decompression components for 9 lossless compression and 3 lossy compression approaches, Compression performance modeling and estimation (compression ratio, compression throughput, query throughput, query accuracy for lossy methods)

**Jan.2022 - Feb.2022**: Finish optimization problem formulation and try to use linear programming to solve the compression selection problem; Build power consumption model and run validation experiments.

**Feb.2022 - Mar.2022**: Finish compression selection experiments for online version and add more lossy compression approaches (BTrDB for aggregation query).

**Mar.2022 - Apr.2022**: Finish compression selection experiments for offline version and compression-enabled eviction policy.

**Apr.2022 - May.2022**: Finish compression selection experiments for power saving mode and thesis draft.

# References


[1] Apache CarbonData. https://carbondata.apache.org/.

[2] Huge stock market dataset. `https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs`.

[3] Individual household electric power consumption data set. https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption.

[4] Influxdb - open source time series, metrics, and analytics database. `https://www.influxdata.com/`.

[5] Time series benchmark suite (tsbs). https://github.com/timescale/tsbs.

[6] Time-series data simplified. `https://www.timescale.com`.

[7] Tlc trip record data. www.nyc.gov/html/tlc/html/about/triprecorddata.shtml.

[8] Torque characteristics of a permanent magnet motor. `https://www.kaggle.com/graxlmaxl/identifying-the-physics-behind-an-electric-motor`.

[9] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[10] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving string compression. In *ICDE*, pages 655–663, Feb 1996.

[11] Gennady Antoshenkov. Dictionary-based order-preserving string compression. *VLDB Journal*, 6(1):26–39, 1997.

[12] Apache Foundation. Apache Parquet. https://parquet.apache.org/.

[13] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *ACM SIGMOD*, pages 283–296. ACM, 2009.

[14] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[15] Paul Boniol, John Paparrizos, Themis Palpanas, and Michael J Franklin. Sand: streaming subsequence anomaly detection, 2021.

[16] Robert S Boyer and J Strother Moore. Mjrty—a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.

[17] Javier Burgués, Juan Manuel Jiménez-Soto, and Santiago Marco. Estimation of the limit of detection in semiconductor gas sensors through linearized calibration models. *Analytica chimica acta*, 1013:13–25, 2018.

[18] Javier Burgués and Santiago Marco. Multivariate estimation of the limit of detection by orthogonal partial least squares in temperature-modulated mox sensors. *Analytica chimica acta*, 1019:49–64, 2018.

[19] Martin Burtscher and Paruj Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2008.

[20] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[21] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *SIGMOD Record*, 30(2):271–282, 2001.

[22] Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: A survey. *ACM Computing Surveys (CSUR)*, 53(2):1–35, 2020.

[23] Florent De Dinechin, Jérémie Detrey, Octavian Cret, and Radu Tudoran. When fpgas are better at floating-point than microprocessors. In *FPGA*, volume 8, page 260, 2008.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[25] Peter Deutsch et al. Gzip file format specification version 4.3. Technical report, RFC 1952, May, 1996.

[26] Hank Dietz, Bill Dieter, Randy Fisher, and Kungyen Chang. Floating-point computation with just enough accuracy. In *International Conference on Computational Science*, pages 226–233. Springer, 2006.

[27] Adam Dziedzic, John Paparrizos, Sanjay Krishnan, Aaron Elmore, and Michael Franklin. Band-limited training and inference for convolutional neural networks. In *International Conference on Machine Learning*, pages 1745–1754. PMLR, 2019.

[28] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. UDP: A Programmable Accelerator for Extract-transform-load Workloads and More. In *IEEE/ACM Micro*, pages 55–68, 2017.

[29] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[30] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.

[31] Edward Fredkin. Trie memory. *Commun. ACM*, 3.9:490–499, 1960.

[32] Jean-loup Gailly and Mark Adler. Zlib compression library. 2004.

[33] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE transactions on Computers*, (1):48–59, 1982.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[35] GNU Foundation. GZip. https://www.gnu.org/software/gzip/.

[36] Google. Snappy. https://github.com/google/snappy.

[37] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Symposium on Applied Computing*, pages 22–27, Apr 1991.

[38] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *ACM SIGMOD*, pages 1917–1923. ACM, 2015.

[39] Brian Hentschel, Michael S Kester, and Stratos Idreos. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 857–872, 2018.

[40] David Hough. Applications of the proposed ieee 754 standard for floating-point arithetic. *Computer*, (3):70–74, 1981.

[41] Te C Hu and Alan C Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[42] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877, 2005.

[43] Balakrishna R. Iyer and David Wilhite. Data Compression Support in Databases. In *VLDB*, pages 695–704, 1994.

[44] Hao Jiang and Aaron J Elmore. Boosting data filtering on columnar encoding with simd. In *DAMON*, page 6. ACM, 2018.

[45] Hao Jiang and Aaron J Elmore. Boosting data filtering on columnar encoding with simd. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, pages 1–10, 2018.

[46] Hao Jiang, Chunwei Liu, Qi Jin, John Paparrizos, and Aaron J Elmore. Pids: attribute decomposition for improved compression and query performance in columnar storage. *Proceedings of the VLDB Endowment*, 13(6):925–938, 2020.

[47] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Practical string dictionary compression using string dictionary encoding. In *Big Data Innovations and Applications*, pages 1–8. IEEE, 2017.

[48] Alexey Karyakin and Kenneth Salem. An analysis of memory power consumption in database systems. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, pages 1–9, 2017.

[49] M. G. Kendall. A New Measure Of Rank Correlation. *Biometrika*, 30(1-2):81, 1938.

[50] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Adaptive energy-control for in-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 351–364, 2018.

[51] Kelly Kissock. University of dayton kettering labs window energy use analysis. 2007.

[52] Marcel Kornacker, Victor Bittorf, Taras Bobrovytsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, and Milne Michael Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[53] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[54] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[55] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theor.*, 22(1):75–81, September 1976.

[56] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[57] Libelium. 50 Sensor Applications for a Smarter World. `http://www.libelium.com/50_sensor_applications/`, 2019. [Online; accessed 19-January-2022].

[58] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[59] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[60] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment*, 14(11):2586–2598, 2021.

[61] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J Elmore. Mostly order preserving dictionaries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1214–1225. IEEE, 2019.

[62] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[63] Adrian Michalke, Philipp M Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. An energy-efficient stream join for the internet of things. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*, pages 1–6, 2021.

[64] Ingo Müller, Andrea Arteaga, Torsten Hoefler, and Gustavo Alonso. Reproducible floating-point aggregation in rdbmss. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1049–1060. IEEE, 2018.

[65] Ingo Müller, Cornelius Ratsch, Franz Faerber, et al. Adaptive string dictionary compression in in-memory column-store database systems.

[66] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association.

[67] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[68] John Paparrizos. 2018 ucr time-series archive: Backward compatibility, missing values, and varying lengths, January 2019. `https://github.com/johnpaparrizos/UCRArchiveFixes`.

[69] John Paparrizos and Michael J Franklin. Grail: efficient time-series representation learning. *Proceedings of the VLDB Endowment*, 12(11):1762–1777, 2019.

[70] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870, 2015.

[71] John Paparrizos and Luis Gravano. Fast and accurate time-series clustering. *ACM Transactions on Database Systems (TODS)*, 42(2):1–49, 2017.

[72] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. Vergedb: A database for iot analytics on edge devices. In *CIDR*, 2021.

[73] John Paparrizos, Chunwei Liu, Aaron J Elmore, and Michael J Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1887–1905, 2020.

[74] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, and Jens Krueger. How to juggle columns: an entropy-based approach for table compression. In *IDEAS*, pages 205–215. ACM, 2010.

[75] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[76] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.

[77] Navneet Potti and Jignesh M Patel. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment*, 8(9):898–909, 2015.

[78] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142. IEEE, 2006.

[79] Gautam Ray, Jayant R. Haritsa, and S Seshadri. Database Compression: A Performance Enhancement Tool. 09 2004.

[80] David Reinsel-John Gantz-John Rydning. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 2018.

[81] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423*, 2020.

[82] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *ACM SIGMOD*, pages 731–742. ACM, 2012.

[83] Emma Stewart, Anna Liao, and Ciaran Roberts. Open $\mu$pmu: A real world reference distribution micro-phasor measurement unit data set for research and application development. 2016.

[84] Jon Stokes. *Inside the machine: an illustrated introduction to microprocessors and computer architecture*. No Starch Press, 2007.

[85] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[86] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A Column-oriented DBMS. In *Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

[87] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *ACM SIGMOD*, pages 204–215. ACM, 2002.

[88] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Accelerating generalized linear models with mlweaving: A one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment*, 12(7):807–821, 2019.

[89] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21(1):18749–19424, 2011.

[90] Wikipedia contributors. Snappy (compression) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Snappy_(compression)&oldid=977673115`, 2020. [Online; accessed 14-September-2020].

[91] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[92] I. H. Witten, A. Moffat, and T. C. Bell. Managing gigabytes: Compressing and indexing documents and images. *IEEE Transactions on Information Theory*, 41(6):2101–, Nov 1995.

[93] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[94] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[95] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, (3):337–343, September 1977.

[96] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, (5):530–536, September 1978.

[97] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[98] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, pages 17–22, 2005.

[99] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59, 2006.