THE UNIVERSITY OF CHICAGO

MOSTLY ORDER PRESERVING DICTIONARIES

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE DIVISION
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
CHUNWEI LIU

CHICAGO, ILLINOIS
JUNE 4TH, 2019
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. iv
LIST OF TABLES ................................................................. v
ACKNOWLEDGMENTS ............................................................. vi
ABSTRACT ................................................................. vii

1 INTRODUCTION ................................................................. 1

2 RELATED WORK ................................................................. 5
  2.1 Dictionary encoding ..................................................... 5
  2.2 Consensus Protocol ....................................................... 7

3 MOP OVERVIEW ................................................................. 9
  3.1 MOP Definitions .......................................................... 9
  3.2 Cascade-MOP .............................................................. 10
  3.3 Dictionary Generation Assumptions .................................. 11

4 METHODOLOGY ................................................................. 12
  4.1 MOP Generation .......................................................... 12
     4.1.1 Initialization ...................................................... 12
     4.1.2 MOP Encoding ................................................... 14
     4.1.3 Finalization ...................................................... 15
  4.2 MOP Queries ............................................................. 16
     4.2.1 Equality Operator ............................................... 16
     4.2.2 Range Operator .................................................. 16
     4.2.3 Sort Operator ................................................... 19

5 EXPERIMENTS AND EVALUATIONS ........................................ 21
  5.1 Experimental Setup ..................................................... 21
  5.2 Evaluation ............................................................... 22
     5.2.1 Range Filtering Evaluation .................................... 23
     5.2.2 SORT Evaluation ................................................ 27
     5.2.3 MOP Generation .................................................. 28
     5.2.4 Overall Compression Performance ......................... 41

6 CONCLUSION ................................................................. 43

REFERENCES ................................................................. 44
LIST OF FIGURES

1.1 Tension between three properties of OP dictionary .............................. 2

3.1 Key space for MOP dictionary .................................................. 10
3.2 Key space for C-MOP dictionary ............................................. 11

4.1 MOP generation example ........................................................ 13
4.2 Steps for range filtering with C-MOP ........................................ 18
4.3 Sort with MOP ................................................................. 19

5.1 Parquet Columnar Store Format .................................................. 22
5.2 Range filter overview on taxi dataset ...................................... 24
5.3 Percent of records encoded by ordered keys. ............................... 24
5.4 Type-2 filter on taxi dataset ................................................ 24
5.5 Range query on TPC-H dataset ............................................. 25
5.6 TPC-H type 2 range filter with selectivity 0.00001 .......................... 25
5.7 TPC-H Type-2 range filter with selectivity 0.27869 ....................... 25
5.8 C-MOP range filter on taxi dataset with increasing the number of cascading levels. 26
5.9 SORT Operator Runtime ....................................................... 28
5.10 Average generation time. ...................................................... 29
5.11 Evaluating the effect of lookahead on MOP generation performance (Runtime and Ordered Percentage). ........................................ 30
5.12 Evaluating the effect of pitch on MOP ordered percentages using uniform, zipf, and taxi workloads. ........................................ 31
5.13 Worker configurations ......................................................... 32
5.14 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 ............................................................. 33
5.15 Evaluating the effect of MOP layer ratio on MOP ordered percentages using uniform, zipf, and taxi workloads. ................................. 34
5.16 Evaluating the effect of dynamic MOP layer ratios when correcting for data distribution changes. .................................................. 35
5.17 Evaluating the space saving benefit of C-MOP with regard to padding ratio. ................................................................. 38
5.18 Evaluating the benefit of C-MOPs with regard to padding ratio when using uniform and histogram insertion techniques for the second layer. .................................................. 39
5.19 Evaluating the impact of sorted data and sampling strategy on a MOP’s ordered ratio. ................................................................. 40
5.20 Encoded Column Size and Dict Padding ..................................... 41
5.21 Scanning and decoding 180 million values. .................................. 42
LIST OF TABLES

5.1 Evaluating cardinality estimation techniques. .......................... 37
ACKNOWLEDGMENTS

Acknowledgements: This work was supported by the CERES Center for Unstopable Computer, NSF award CCF-1757970, and gifts from Google and Futurewei. The experiments used the Chameleon testbed supported by the NSF.
ABSTRACT

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.
CHAPTER 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 [3] (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 [15], which can be CPU-intensive [11].

While all columnar encodings offer compression benefits over unencoded data, query benefits can be limited given the value domain and data distribution [3]. 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 [31]. 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) [31, 19], as well as supporting fast query execution via efficient hardware instructions [19]. As a result, dictionary encoding is widely supported in many analytic platforms systems [32, 6, 31, 1, 25, 16], and offers significant benefits for string data types that are common in many domains, such as enterprise data [12] and open data initiatives [19]. 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 [26]. 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 [7]. 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 [7]. 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 [33]. 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 [19].

![Figure 1.1: Tension between three properties of OP dictionary](image)

Figure 1.1: Tension between three properties of OP dictionary

Therefore, in Figure 1.1, 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 [6]. 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 2 discusses related work. Section 3 overviews MOP and key definitions. Sections 4.1 and 4.2.2 cover MOP generation and query execution respectively. Section 5 evaluates MOP for both range filtering and generation. The paper concludes in Section 6.
CHAPTER 2
RELATED WORK

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.

2.1 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 [24, 37]. 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 [7].

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 [18]. These compression techniques, such as GZip and Lempel-Ziv [38], 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 [3].

Research has explored optimizing dictionary encoding for database systems. Chen et al. [9] proposes a hierarchical dictionary encoding scheme for string attributes, supporting encoding at different granularities (attribute level, word level, prefix level, etc.). Paradies et al. [29] demonstrates an entropy-based approach that is adaptive to user query patterns. Column-oriented databases, such as MonetDB [39] and C-Store [32], 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. [26] 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. [40] 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. [9] design a compression-aware optimizer to estimate overhead brought by accessing compressed data. Ray et al. [30] and Abadi et al. [3] 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. [19] 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 [4] 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. [5] propose an order-preserving data structure for DBMS and Binnig et al. [7] 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 [13] and different compression strategies, such as Huffman coding, Hu-Tucker coding [17], front coding [34], and RE-PAIR [23]. 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 [20], 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.

2.2 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. [35] presents using distributed logs to maintain unified dictionary views on each node. Amazon’s Dynamo [10] 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 [14], Raft [27], and RAMCloud [28].

Protocols with Distinguished Leader

In this type of protocol, a unique node is designated([14, 28]) or elected([27]) 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 [27]. 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.
CHAPTER 3
MOP OVERVIEW

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

3.1 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 3.1 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* 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 4.1 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 is instead of a single ordered section, is cascading ordered sections before defaulting to a disordered section.

### 3.2 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 3.2 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 4.1.2 we discuss determining the number of cascading levels.

### 3.3 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.
CHAPTER 4
METHODOLOGY

4.1 MOP Generation

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.

4.1.1 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, 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 4.1. It is worth noting that this process could leverage more advanced sample-based distinct value estimators [8]; however, our experiments in Section 5.2.3 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 5.2.3.

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.
4.1.2 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 4.1 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 4.1 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.

4.1.3 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 3.1 and 3.2 respectively show the final organization for MOP and C-MOP. The generated dictionary is written to the output.
4.2 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.

4.2.1 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.

4.2.2 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 via query rewriting. To evaluate \( x \in (v_a, v_b) \), we first find the corresponding key range \( k_a = \min\{k | \text{OP}(k) \geq v_a\} \) and \( k_b = \max\{k | \text{OP}(k) \leq v_b\} \), 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 \((v_{min}, v_{max}) \cap (v_a, v_b) = \emptyset\). 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 4.1. 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 \( k \geq 14 \land \text{decode}(k) \in (apple, cherry) \). The result, as the disjunction of
the two parts, is then

\[ k \in (0, 4) \lor (k \geq 14 \land \text{decode}(k) \in (\text{apple, cherry})) \]

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

\[ k \in (0, 11) \lor k \geq 14 \]

Figure 4.2: 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., \( k \in (k_1, k_2) \lor k \in (k_2, k_3) \), 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 4.2. 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).

4.2.3 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 disordered 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 4.3: Sort with MOP](image)

To perform sort operation on a MOP encoded column, we create a bijection \( sort : \mathcal{N} \rightarrow \mathcal{R} \). 


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

$$sort(k) = \begin{cases} 
  k & k \in OP \\
  D_{key}(k) & k \notin OP 
\end{cases}$$
CHAPTER 5
EXPERIMENTS AND EVALUATIONS

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.

5.1 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 [2], 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 [36, 22]. It provides efficient data compression and encoding schemes with the ability to handle complex nested data types (e.g. lists and maps) [25]. 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 decoding the encoded keys.

As shown in Figure 5.1, 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.

5.2 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.
5.2.1 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 format.

Using a stand-alone Java query execution framework, we evaluate the three types of range queries discussed in Section 4.2.2 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 [32]. 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 [2]. 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 5.2 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 5.3 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 5.4, 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 5.5: Range query 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 5.3 shows the percentage of records encoded by the MOP ordered section increases uniformly.

In Figure 5.5, 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 5.3 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 from 99.8% to 25.9%. Figures 5.6 and 5.7 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 5.8 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.

5.2.2 SORT Evaluation

In this experiment, we implement a MOP sort operator based on a recursive quick sort algorithm as described in Section 4.2.3. Figure 5.9 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.
5.2.3 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 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 5.10, 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 5.10(a), the dictionaries are
generated on the *shipdate* of the *lineitem* table with TPC-H scale 30. In Figure 5.10(b), the dictionaries are generated on the pickup latitude of the full taxi dataset, which is roughly 30 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 performs 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 5.11(a) shows resulting ordered percentages and speeds of generating MOPs 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 5.11(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 [11]. This result shows that high lookahead percentages can come with a high cost.

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 5.12, 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. 5.2.4) and worse query performance [19].
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 5.13(a) shows how changing 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 5.13(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 5.14: 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 5.14, 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 5.15 show that increasing the layer count increases ordered percentage as each successive layer creates more space for encoded values to be inserted into. Figure 5.15 also shows that increasing the MOP layer 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

Figure 5.15: Evaluating the effect of MOP layer ratio on MOP ordered percentages using uniform, zipf, and taxi workloads.
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.

![Graph showing the effect of dynamic MOP layer ratios](image)

Figure 5.16: 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 exper-
iment 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 5.16 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 5.1 compares our simple cardinality estimator that divides the number of distinct values in the sample by the lookahead and a more advanced adaptive estimator [8] 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.

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

Table 5.1: 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 5.17. 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 5.17: Evaluating the space saving benefit of C-MOP with regard to padding ratio.](image)

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 5.18 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 5.18: 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 4.1.1, 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 [21] ≥ 0.8 or ≤−0.8). In Figure 5.19 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 5.19: 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).

### 5.2.4 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).

As is shown in Figure 5.20(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

![Figure 5.20: Encoded Column Size and Dict Padding](image)

((a)) TPC-H

Figure 5.20: Encoded Column Size and Dict Padding

((b)) Taxi
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%.

![Graph showing impact of padding ratios on scanning and decoding time](image)

**Figure 5.21:** Scanning and decoding 180 million values.

Figure 5.21 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.
CHAPTER 6

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.
REFERENCES


