2019 IEEE 35th International Conference on Data Engineering (ICDE)

Mostly Order Preserving Dictionaries

Chunwei Liu
University of Chicago
chunwei@cs.uchicago.edu

McKade Umbenhower
University of Wyoming
mumbenho@uwyo.edu

Hao Jiang
University of Chicago
hajiang@cs.uchicago.edu

Pranav Subramaniam
University of Chicago
psubramaniam@uchicago.edu

Jihong Ma
Alibaba Group
jihong.ma@alibaba-inc.com

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

***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.**

I. 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 [1] (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 [2], which can be CPU-intensive [3].

While all columnar encodings offer compression benefits over unencoded data, query benefits can be limited given the value domain and data distribution [1]. 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 [4]. 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) [4], [5], as well as supporting fast query execution via efficient hardware instructions [5]. As a result, dictionary encoding is widely supported in many analytic platforms systems [4], [6]–[10], and offers significant benefits for string data types that are common in many domains, such as enterprise data [11] and open data initiatives [5]. 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 [12]. 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 [14]. 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 [5].

Therefore, we propose a conjecture for dictionary encodings that 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

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00111

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 [7]. 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 II discusses related work. Section III overviews MOP and key definitions. Sections IV and V cover MOP generation and query execution respectively. Section VI evaluates MOP for both range filtering and generation.

II. Related Work

Dictionary encoding creates a bijective mapping between values of variable length to compact integer codes, and replaces the original data entries with corresponding codes. 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 [15], [16].

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 [17]. These compression techniques, such as GZip and Lempel-Ziv [18], 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 [1].

Research has explored optimizing dictionary encoding for database systems. Chen et al. [19] proposes a hierarchical dictionary encoding scheme for string attributes, supporting encoding at different granularities (attribute level, word level, prefix level, etc.). Paradies et al. [20] demonstrates an entropy-based approach that is adaptive to user query patterns. Column-oriented databases, such as MonetDB [21] and C-Store [6], 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. [12] 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. [22] 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. [19] design a compression-aware optimizer to estimate overhead brought by accessing compressed data. Ray et al. [23] and Abadi et al. [1] 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. [5] 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 [24] 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. [25] 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 [26] and different compression strategies, such as Huffman coding, Hu-Tucker coding [27], front coding [28], and RE-PAIR [29]. 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 [30], 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.

III. MOP Overview

Here we introduce the MOP organization, necessary data structures, and how query execution works with a MOP.

A. 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 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 IV we discuss how MOP allocates keyspace and assigns keys.

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.

B. 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. We call each order-preserving section an OP zone. Figure 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.

IV. 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.

A. 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 τ which generates a real number in $[-1, 1]$. An output of 1

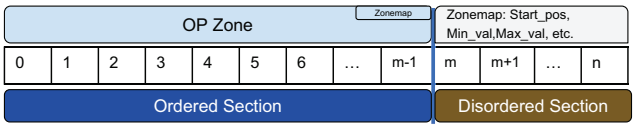


Fig. 1: Key space for MOP dictionary

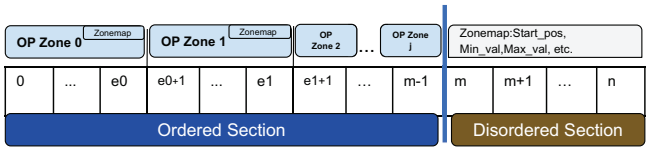


Fig. 2: Key space for C-MOP dictionary

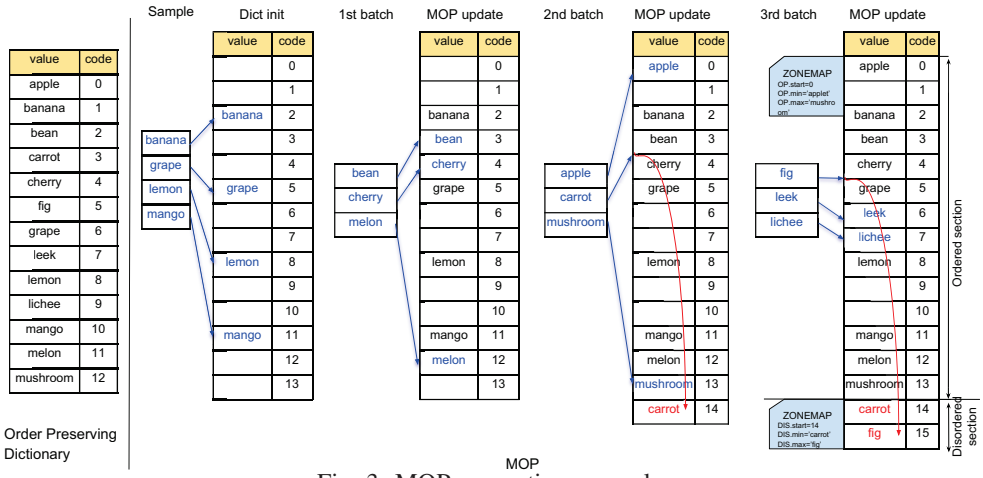


Fig. 3: MOP generation example

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. It is worth noting that this process could leverage more advanced sample-based distinct value estimators [31]; however, our experiments in Section VI-B5 show these approaches add limited value to MOP performance.

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.

B. 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 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.

1) 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 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.

2) 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.

C. 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 1 and 2 respectively show the final organization for MOP and C-MOP. The generated dictionary is written to the output file.

V. 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.

A. 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.

B. 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 Figure 3. 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

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

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.

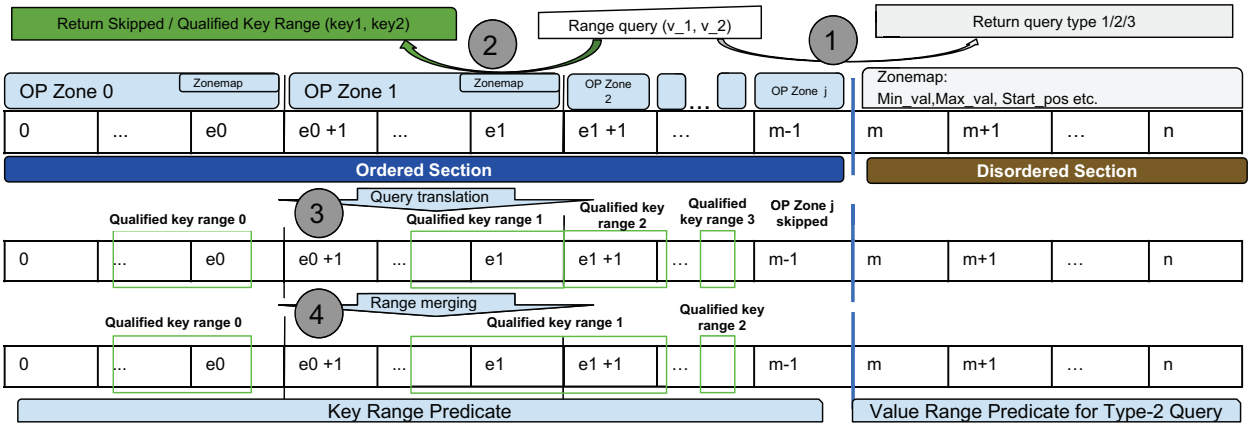


Fig. 4: Steps for range filtering with C-MOP

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 Figure 4. 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).

C. 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.

VI. Evaluation

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.

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 [32], 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.

A. 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 V 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

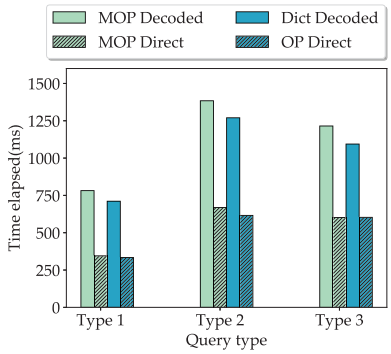


Fig. 5: Range filter overview on taxi dataset

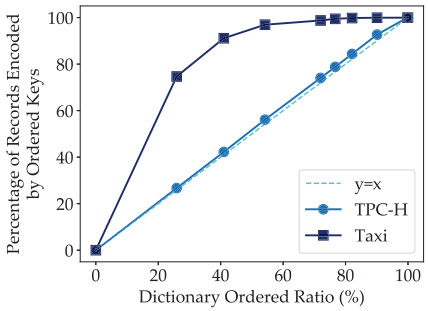


Fig. 6: Percent of records encoded by ordered keys.

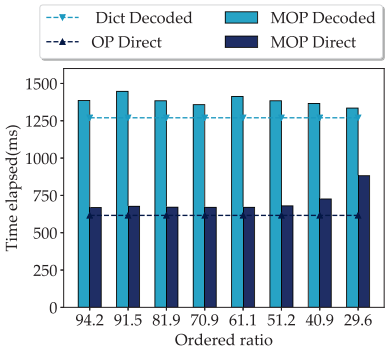


Fig. 7: Type-2 filter on taxi dataset

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 [6]. Unless otherwise stated, we run each query 15 times and report the average running time.

1) 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 [32]. 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 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 6 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 7, 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.

2) 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.

In Figure 8, 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 6 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

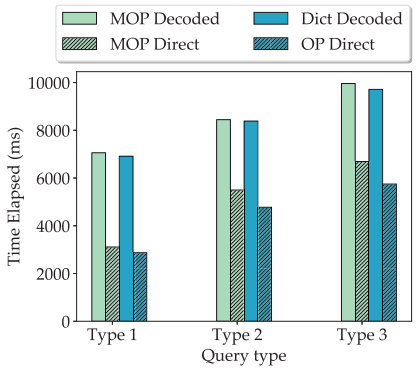


Fig. 8: Range query on TPC-H dataset

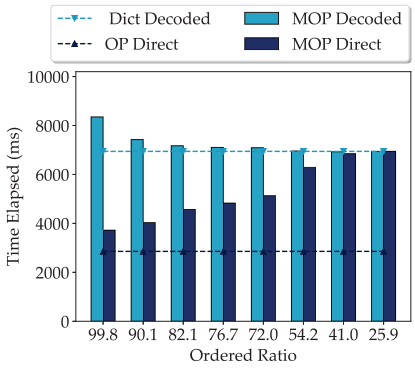


Fig. 9: TPC-H type 2 range filter with selectivity 0.00001

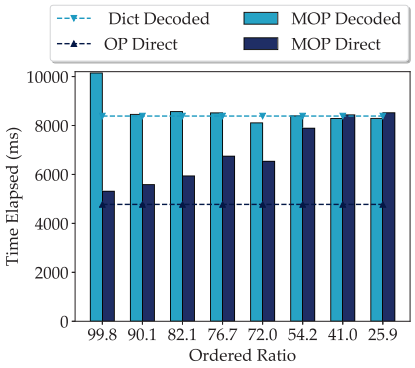


Fig. 10: TPC-H Type-2 range filter with selectivity 0.27869

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 9 and 10 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.

3) C-MOP Filtering: Figure 11 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.

4) SORT Evaluation: In this experiment, we implement a MOP sort operator based on a recursive quick sort algorithm as described in Section V-C. Figure 12 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.

B. 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 coor-

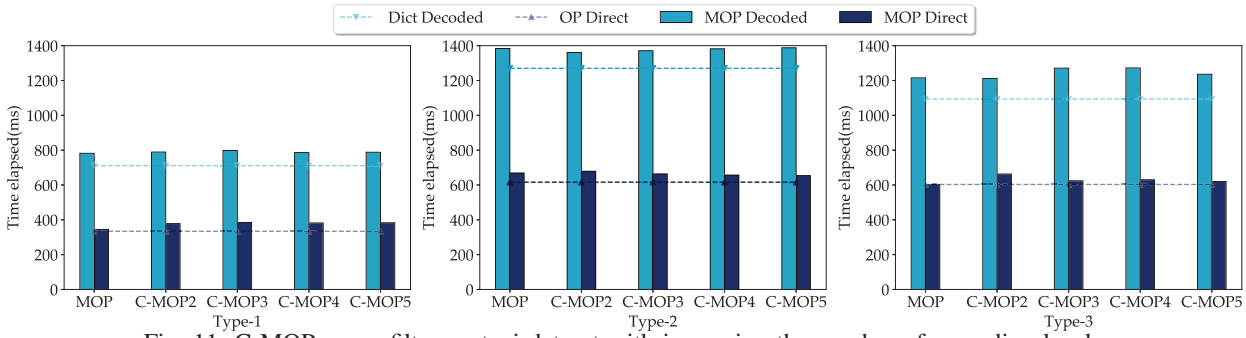


Fig. 11: C-MOP range filter on taxi dataset with increasing the number of cascading levels.

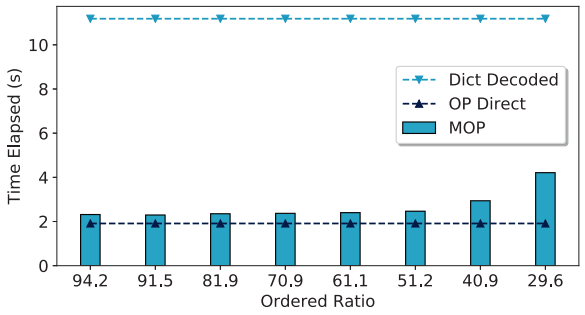
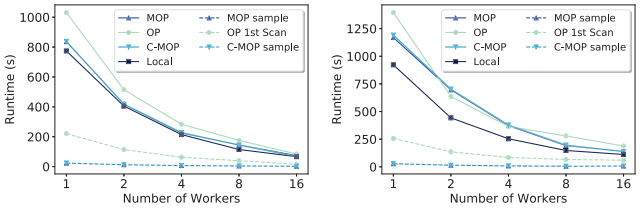


Fig. 12: SORT Operator Runtime

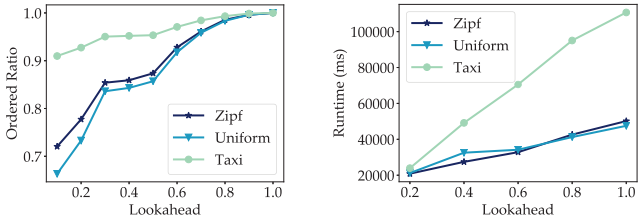
dinator and one for all workers, with experiments including network latency in runtimes.



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

Fig. 13: Average generation time.

As shown in Figure 13, 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 13a, the dictionaries are generated on the *shipdate* of the *lineitem* table with TPC-H scale *30*. In Figure 13b, 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 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.



(a) Ordered Percentage (b) Runtime (ms)

Fig. 14: Evaluating the effect of lookahead on MOP generation performance (Runtime and Ordered Percentage).

1) 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 14a 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.

In Figure 14b, 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

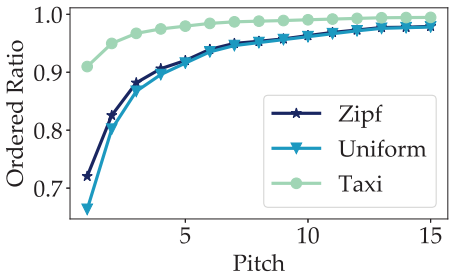


Fig. 15: Evaluating the effect of pitch on MOP ordered percentages using uniform, zipf, and taxi workloads.

of parsing and validating [3]. This result shows that high lookahead percentages can come with a high cost.

2) Pitch: Here we discuss the impact of scaling the key space with the pitch parameter. Figure 15 shows that larger pitch values increase the amount of room left for encoded values in the ordered section, resulting in more ordered dictionaries. However, this incurs the cost of a larger key space which can result in a larger file (Sec. VI-B8) and worse query performance [5].

3) 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. The tests in Figure 16 show that increasing the layer count increases ordered percentage as each successive layer creates more space for encoded values to be inserted into. Figure 16 also shows that increasing the MOP layer ratio

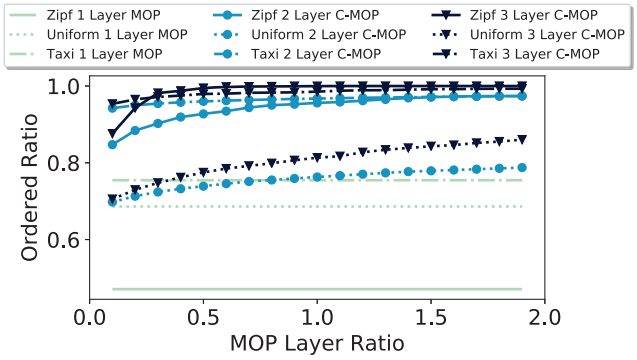


Fig. 16: Evaluating the effect of MOP layer ratio on MOP ordered percentages using uniform, zipf, and taxi workloads.

parameter increases the ordered percentage of the dictionary. As successive layer sizes are set by multiplying the previous layer size by the MOP layer ratio, 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. 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.

4) 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.

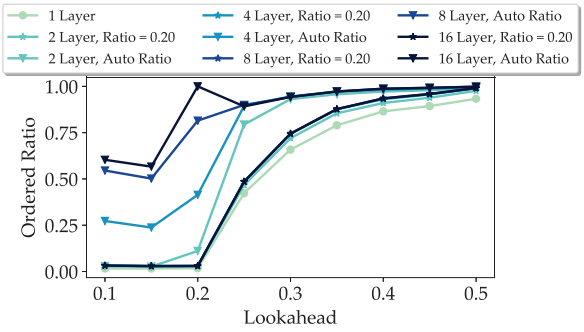


Fig. 17: Evaluating the effect of dynamic MOP layer ratios when correcting for data distribution changes.

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 17 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.

5) Cardinality Estimation: Table I compares our simple cardinality estimator that divides the number of distinct values in the sample by the lookahead and a more advanced adaptive estimator [31] 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. 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.

6) 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

		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 I: Evaluating cardinality estimation techniques.

90% ordered. The average key space needed to achieve these ordered percentages are shown in Figure 18. 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.

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. 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.

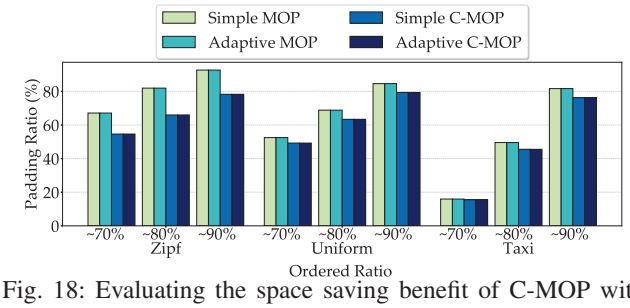


Fig. 18: Evaluating the space saving benefit of C-MOP with regard to padding ratio.

7) Column Sortedness: As described in Section IV-A, 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 [33] ≥ 0.8 or ≤ -0.8). In Figure 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.

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).

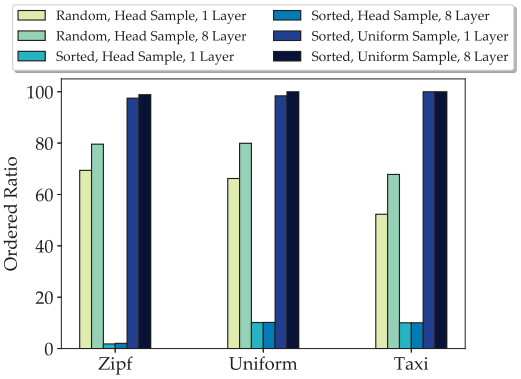
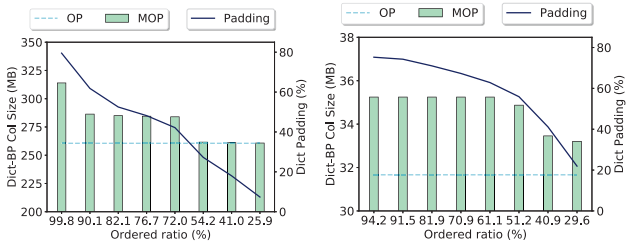


Fig. 19: Evaluating the impact of sorted data and sampling strategy on a MOP's ordered ratio.

8) 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

Fig. 20: Encoded Column Size and Dict Padding

As is shown in Figure 20a, 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 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.

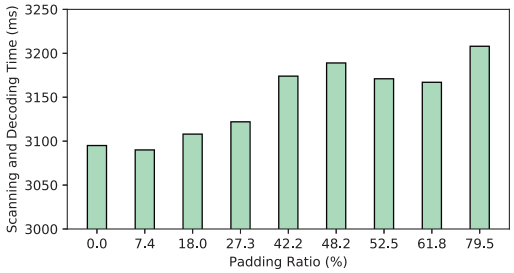


Fig. 21: Scanning and decoding 180 million values.

VII. 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.

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

References

[1] D. Abadi, S. Madden, and M. Ferreira, "Integrating Compression and Execution in Column-oriented Database Systems," in *SIGMOD*, pp. 671–682, 2006.

[2] G. Graefe and L. D. Shapiro, "Data compression and database performance," in *Symposium on Applied Computing*, pp. 22–27, Apr 1991.

[3] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "UDP: A Programmable Accelerator for Extract-transform-load Workloads and More," in *IEEE/ACM Micro*, pp. 55–68, 2017.

[4] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in SAP HANA database: the end of a column store myth," in *ACM SIGMOD*, pp. 731–742, ACM, 2012.

[5] H. Jiang and A. J. Elmore, "Boosting data filtering on columnar encoding with simd," in *DAMON*, p. 6, ACM, 2018.

[6] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A Column-oriented DBMS," in *Very Large Data Bases*, pp. 553–564, VLDB Endowment, 2005.

[7] Apache Foundation, "Apache Parquet." https://parquet.apache.org/.

[8] "Apache CarbonData." https://carbondata.apache.org/.

[9] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *PVLDB*, vol. 3, no. 1-2, pp. 330–339, 2010.

[10] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *ACM SIGMOD*, pp. 1917–1923, ACM, 2015.

[11] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[12] I. Müller, C. Ratsch, F. Faerber, *et al.*, "Adaptive string dictionary compression in in-memory column-store database systems.,"

[13] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in *ACM SIGMOD*, pp. 283–296, ACM, 2009.

[14] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, "Storing and querying ordered XML using a relational database system," in *ACM SIGMOD*, pp. 204–215, ACM, 2002.

[15] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Trans. Inf. Theor.*, vol. 22, pp. 75–81, Sept. 1976.

[16] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theor.*, pp. 337–343, Sept. 1977.

[17] B. R. Iyer and D. Wilhite, "Data Compression Support in Databases," in *VLDB*, pp. 695–704, 1994.

[18] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theor.*, pp. 530–536, Sept. 1978.

[19] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems," *SIGMOD Record*, vol. 30, no. 2, pp. 271–282, 2001.

[20] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krueger, "How to juggle columns: an entropy-based approach for table compression," in *IDEAS*, pp. 205–215, ACM, 2010.

[21] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, "MonetDB/X100 - A DBMS In The CPU Cache," *IEEE Data Eng. Bull.*, pp. 17–22, 2005.

[22] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar ram-cpu cache compression," in *ICDE*, pp. 59–59, 2006.

[23] G. Ray, J. R. Haritsa, and S. Seshadri, "Database Compression: A Performance Enhancement Tool," 09 2004.

[24] G. Antoshenkov, D. Lomet, and J. Murray, "Order preserving string compression," in *ICDE*, pp. 655–663, Feb 1996.

[25] G. Antoshenkov, "Dictionary-based order-preserving string compression," *VLDB Journal*, vol. 6, no. 1, pp. 26–39, 1997.

[26] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3.9, pp. 490–499, 1960.

[27] T. C. Hu and A. C. Tucker, "Optimal computer search trees and variable-length alphabetical codes," *SIAM Journal on Applied Mathematics*, vol. 21, no. 4, pp. 514–532, 1971.

[28] I. H. Witten, A. Moffat, and T. C. Bell, "Managing gigabytes: Compressing and indexing documents and images," *IEEE Transactions on Information Theory*, vol. 41, pp. 2101–, Nov 1995.

[29] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[30] S. Kanda, K. Morita, and M. Fuketa, "Practical string dictionary compression using string dictionary encoding," in *Big Data Innovations and Applications*, pp. 1–8, IEEE, 2017.

[31] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya, "Towards estimation error guarantees for distinct values," in *PODS*, pp. 268–279, 2000.

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

[33] M. G. Kendall, "A New Measure Of Rank Correlation," *Biometrika*, vol. 30, no. 1-2, p. 81, 1938.