

# Learned Selection Strategy for Lightweight Integer Compression Algorithms

Lucas Woltmann  
 Technische Universität Dresden,  
 Dresden Database Research Group  
 lucas.woltmann@tu-dresden.de

Patrick Damme  
 Technische Universität Berlin  
 patrick.damme@tu-berlin.de

Claudio Hartmann  
 Technische Universität Dresden,  
 Dresden Database Research Group  
 claudio.hartmann@tu-dresden.de

Dirk Habich  
 Technische Universität Dresden,  
 Dresden Database Research Group  
 dirk.habich@tu-dresden.de

Wolfgang Lehner  
 Technische Universität Dresden,  
 Dresden Database Research Group  
 wolfgang.lehner@tu-dresden.de

## ABSTRACT

Data compression has recently experienced a revival in the domain of in-memory column stores. In this field, a large corpus of lightweight integer compression algorithms plays a dominant role since all columns are typically encoded as sequences of integer values. Unfortunately, there is no single-best integer compression algorithm and the best algorithm depends on data and hardware properties. For this reason, selecting the best-fitting integer compression algorithm becomes more important and is an interesting tuning knob for optimization. However, traditional selection strategies require a profound knowledge of the (de-)compression algorithms for decision-making. This limits the broad applicability of the selection strategies. To counteract this, we propose a novel *learned selection strategy* by considering integer compression algorithms as independent black boxes. This black-box approach ensures broad applicability and requires machine learning-based methods to model the required knowledge for decision-making. Most importantly, we show that a local approach, where every algorithm is modeled individually, plays a crucial role. Moreover, our *learned selection strategy* is generalized by user-data-independence. Finally, we evaluate our approach and compare our approach against existing selection strategies to show the benefits of our *learned selection strategy*.

## 1 INTRODUCTION

Data compression has been a well-established query optimization technique in database systems for decades [7, 16, 31]. Nevertheless, data compression has experienced a revival in the domain of in-memory column stores [1, 2, 10] to optimize the execution of analytical queries. In this field, lightweight integer compression algorithms play an essential role since all columns are usually encoded as sequences of integer values [5, 13, 35]. Based on that encoding, the whole query processing is done on these integer sequences [1, 2, 5, 13, 35]. Moreover, with the help of some lightweight computations for integer compression, the necessary memory space can be reduced on the one hand [1, 2, 8]. On the other hand, compressed integer values allow for improving the processing performance (i) by increasing the effective bandwidth to reduce the memory wall, (ii) by yielding a better utilization of the cache hierarchy, and (iii) by enabling highly data-parallel processing [1, 2, 8, 10, 24]. However, choosing the right compression

algorithm is not trivial. Compression selection has many objectives, like query memory consumption or query runtime, which can be optimized. Most algorithms only achieve improvements for one objective at a time, so a holistic selection has to look at all combinations of algorithms, data properties, and objectives to reach improved results. To show how complex this topic is, we present an end-to-end experiment and discuss its impact on selecting the right algorithm.

### 1.1 Motivational End-to-End Experiment

As mentioned before, the different objectives of query optimization lead to high complexity in selecting the right algorithm for a given data set. To back this assumption, we use a simple analytical query `SELECT SUM(Y) FROM R WHERE X = c` and measure its end-to-end performance for different integer compression algorithms within the analytical query engine MorphStore [10]. MorphStore uses an operator-at-a-time processing model with on-the-fly de/recompression, i.e., operators decompress the inputs and recompress the outputs on the fly while processing uncompressed data internally. Thus, the (de)compression runtimes determine the query runtime difference between two alternative query plans with different compressed formats selected. All compression algorithms are explained in detail in Section 4.1. In our experiment, performance is differentiated into memory footprint and query runtime. Figure 1 illustrates the difficulty of finding the best integer compression algorithm for our example query given different data inputs. Every bar in this figure shows the performance for uncompressed data or a different

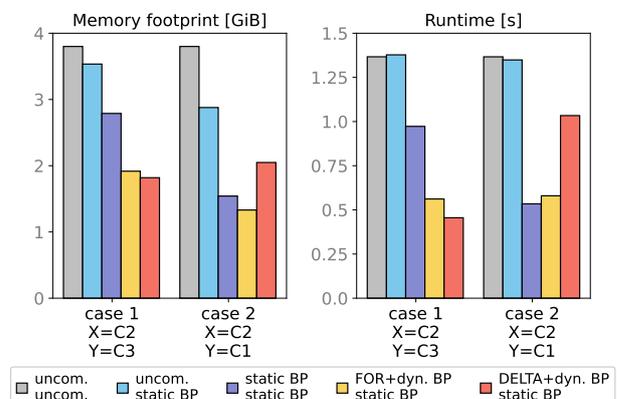


Figure 1: The importance of choosing the right algorithms for better end-to-end performance.

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-092-9 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

combination of compression algorithms on the two input data columns (first line in legend) and any intermediates (second line in legend). We use two cases to model different data inputs and properties, each using two out of three columns C1 to C3, with different data distributions for the aggregated column Y and the filtered column X. Case 1 uses column C3 with sorted values as the aggregated column Y and case 2 uses column C1 with very small values as the aggregated column Y. The filtered column X in both cases is column C2, which contains enormous outliers in its data distribution. Whereas case 1 can always rely on FOR+dynamic bit packing (BP) for best results in both memory footprint and runtime, the second case needs a finer distinction. Here, FOR+dynamic BP is the best option for the memory footprint, but static BP is the best for query runtime. So, it is crucial for query performance to choose the right algorithm or an algorithm close to the performance objective optimum. Therefore, a key feature of a selection strategy is the inclusion of general knowledge about data distributions and their influence on the algorithms to make a data-independent decision and generalize over different inputs.

## 1.2 State-of-the-Art and Shortcomings

The large number of proposed integer compression algorithms in the literature and the insight that there is no single-best algorithm highlight the complexity of finding the best algorithm for every data input [2, 8, 9]. The best integer compression algorithm depends on data as well as hardware properties [8, 9, 24]. However, as shown in [2, 10, 12, 20], analytical query processing in in-memory column stores can be optimized if the best integer compression is selected. To select the best-fitting compression algorithm, an appropriate selection strategy is needed. For that, three general approaches have been proposed: (1) rule-based selection, (2) cost-based selection, and (3) ML-based selection.

Regarding rule-based techniques, Abadi et al. [2] have hand-crafted a decision tree based on an empirical evaluation of a small number of compression algorithms in 2006. One advantage of this approach is that making a decision is very cheap as it requires only a few steps through the tree. However, as the field of lightweight compression has evolved significantly since then, their decision tree does not cover (i) the diversity of the algorithm landscape nowadays and (ii) the hardware dependency.

Damme et al. [9] have gone a step further by developing a cost model for lightweight integer compression algorithms. This cost model adopts a *grey-box approach* by explicitly modeling as much knowledge about the algorithms as possible and implicitly capturing the impact of data and hardware characteristics using a small number of calibration measurements for each algorithm. While this approach is able to generalize to a wide range of lightweight integer compression algorithms, a certain amount of manual effort is required to embrace a new algorithm. Furthermore, making a single decision with the cost model involves much more calculations than traversing a decision tree.

A major shortcoming of both approaches is the manual effort required to incorporate algorithms into the selection process. So, a third category of selection strategies based on ML has been proposed. ML-based selection strategies are a relatively new contribution to the field of compression [4, 6, 20, 21]. They model the general selection process as a black box. Data properties are put into a learned model, e.g., neural networks, and the potentially best algorithm is predicted by the model. These models produce good-quality decisions fast. However, contrary to the other two

types of strategies, these ML-based approaches do not explain why an algorithm is picked and most of them are still rigid when it comes to expanding the approach to new algorithms.

## 1.3 Our Contribution and Outline

To achieve a (user-)data-independent selection approach, our main contribution within this paper is to introduce and evaluate a way to approach the integer compression algorithm selection problem with a *learned selection strategy*. To eliminate the manual effort, human engineering, and data dependency for covering compression algorithms, our *learned selection strategy* views compression algorithms as black boxes. This black-box approach ensures broad applicability and generalization over different data but requires machine learning-based methods to model the required knowledge for decision-making. A supervised machine learning-based approach enables us to use the advantage of training models once on a synthetic data set and then apply them repeatedly to new unknown data with very little overhead through their fast forward passes. The prerequisites are split into two challenges for every machine learning-based approach: *feature engineering* and *data generation*. In detail, our contributions are as follows:

- (1) We start by providing the necessary background on lightweight integer compression and related work for selecting a compression algorithm in Section 2.
- (2) Then, we present our machine learning-based *learned selection strategy*, which is based on gradient boosting, in detail in Section 3. In particular, we introduce the necessary concepts for feature engineering and data generation to provide an extensible, data-independent concept with little human interaction for further algorithms.
- (3) Afterwards, we evaluate our novel strategy and compare it to existing strategies in Section 4 by showing that a synthetic data set can be generated once during training and brings generalization and little overhead to the decision-making.

Finally, we discuss our findings in Section 5 and conclude the paper in Section 6.

## 2 RELATED WORK

The general idea of data compression is to invest some computational cycles to reduce the physical data size. This computational effort can usually be amortized by an increased effective bandwidth of data transfers to and from storage mediums. Thus, data compression – in particular *lossless* methods – has been successfully applied in many areas, such as database systems [7, 16, 31].

Depending on the storage medium, different classes of lossless compression methods can be distinguished. On the one hand, classical *heavyweight algorithms*, such as Huffman [19], arithmetic coding [38], variants of Lempel-Ziv [37, 41], and Snappy [18] support arbitrary data but are relatively slow. Thus, they are usually employed to amortize disk access latencies. On the other hand, *lightweight compression algorithms*, which have been developed especially for columnar data [2, 24], are much faster while still achieving great compression rates. This combination makes them suitable for in-memory columnar processing. To give a comprehensive insight into the field of lightweight compression as a foundation for this paper, we summarize those algorithms and review existing selection strategies in this section.

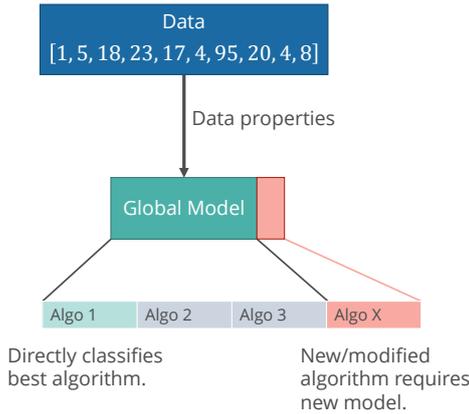


Figure 2: A *global model* selection strategy.

## 2.1 Lightweight Integer Compression

Lightweight compression algorithms usually focus on integer sequences, which is a natural match for columnar data since it is state-of-the-art to represent all values as integer keys from a dictionary [3]. The unique properties of lightweight integer compression algorithms result from their exploitation of specific data characteristics, such as the data distribution, sort order, or the number of distinct values in a column. A large variety of lightweight integer compression algorithms has been proposed and a recent study showed that there is no single best one [2, 8, 9].

All lightweight integer compression algorithms are more or less cascades of one or more basic techniques, whereby five basic techniques are known and heavily applied: run-length encoding (RLE) [2, 32], frame-of-reference (FOR) [17, 42], delta coding (DELTA) [24, 32], dictionary coding (DICT) [2, 3, 32, 42], and null suppression (NS) [2, 24, 32]. The techniques can be described as follows:

**FOR, DELTA:** represent each value as the difference between a given reference value or its predecessor value.

**RLE:** tackles uninterrupted sequences of occurrences of the same value, so-called runs, and each run is represented by its value and run length.

**DICT:** supplants each value by its unique key given by a dictionary.

**NS:** is the most well-studied technique and its basic idea is the omission of leading zeros in the bit representation of small integers.

The goal of FOR, DELTA, and DICT is to represent the original data as a sequence of small integers on the one hand, which is then suited for the actual compression using NS. On the other hand, the ability of these techniques to generalize is used to tailor lightweight integer compression algorithms to different data characteristics, as shown in [8, 9].

## 2.2 Selection Strategies

For this intelligent usage, a selection strategy is required to choose reasonable compression algorithms for the base columns and intermediate results [2, 10, 12, 20]. As already shown in [9], the selection is non-trivial and depends on (i) data characteristics, (ii) the hardware properties, and (iii) the objective. The objective means that the best algorithm concerning the compression rate is not necessarily optimal regarding (de)compression speed.

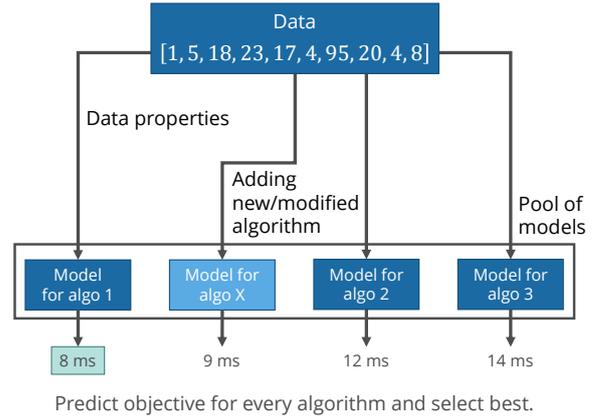


Figure 3: A *local model* selection strategy.

Selecting according to compression rate *minimizes the overall memory footprint*, while selecting according to (de)compression speed *minimizes the query runtime*.

**Selection Objectives.** Several objectives can be regarded for defining the optimal selection outcome. From our experiment in Section 1.1, we can derive that different objectives can be beneficial for query optimization depending on the use case. The two major goals are reduced query memory consumption and query runtime. For integer compression in column stores, this translates to the compression rate to optimize memory consumption and compression or decompression runtimes to optimize the overall query runtime. In detail, the five most important selection objectives (SO) in our scenario are:

**SO1:** compression rate,

**SO2:** runtime compression (ram2ram),

**SO3:** runtime decompression (ram2ram),

**SO4:** runtime decompression (ram2reg),

**SO5:** runtime compression (cache2ram).

Regarding the runtimes, we need to distinguish different contexts in which compression and decompression are typically executed. The classical case is the (de)compression of an entire column, whereby the input is loaded from main memory (RAM) and the output is stored in RAM again, assuming a sufficiently large column. In that respect, objective **SO2** represents the compression of an entire column, e.g., during the initial compression of the base data. Many column store systems keep the data in compressed form during query processing as long as possible [11, 23, 29, 42]. However, as soon as a query operator cannot process the compressed data directly, the data is fully decompressed before the remaining processing happens entirely on uncompressed data. To cover that, we define the selection objective **SO3**. Besides the (de)compression of an entire column, more fine-grained approaches become important when smartly integrating compression into the query execution. In [10], we presented a novel holistic compression-enabled processing model where all intermediate results are represented using a lightweight compression algorithm. To enable that, we developed a morphing wrapper inspired by the transient decompression concept [7]. Our morphing wrapper surrounds an operator internally processing uncompressed data with a fine-grained decompression of the inputs and recompression of the outputs. More precisely, instead of materializing the output of the decompression in RAM,

we forward it immediately to the operator processing one vector register (reg) at a time. This case is represented by selection objective **SO4**. On the output side, a small amount of an operator’s output is buffered in a cache-resident buffer (cache) from where it is forwarded to the recompression. This case is represented by selection objective **SO5**. Since accessing registers or cache is much faster than accessing RAM, the main difference between **SO4**, **SO5** to **SO2**, **SO3** is the different ratio between compute and load/store cost.

**Selection Techniques.** Any selection technique for integer compression algorithms needs to predict the best-fitting algorithm concerning the different objectives from a collection of algorithms  $A$  given the data properties, hardware properties, and algorithm properties. In this context, three selection techniques have been proposed: (1) rule-based, (2) cost-based, and (3) Machine Learning (ML)-based selection techniques. Rule-based techniques are typically modeled as a decision tree or graph guiding through a number of questions to arrive at the suggested selection solution. Abadi et al. [2] manually derived such a decision tree for lightweight integer compression algorithms for a limited number of algorithms. It uses data properties, workload characteristics, and a set of rules to determine the best-fitting compression algorithm. Contrarily to their simplicity, decision trees still need a lot of preliminary research, i.e., feature engineering. Moreover, Abadi et al. [2] only considered the compression rate objective **SO1**. Finally, a major drawback of their hand-crafted decision tree is that it is unclear how to extend it to novel compression algorithms and how to integrate the hardware dependency.

Cost-based selection techniques are based on a cost model and provide a cost function estimating the cost of an alternative solution. Then, the selection problem is solved by choosing the alternative solution incurring the minimum cost according to the cost function. Formally, a cost model is a function modeling an abstraction from a data point  $x$  to a single *cost value*  $y$  for a given algorithm  $a$ .

$$f_a : x \mapsto y \text{ with } x \in \mathbb{R}^n, y \in \mathbb{R} \quad (1)$$

For integer compression, the inputs  $x$  for a cost model are manifold. For example, these can be bit width histograms, statistical properties of the data, or specific properties of the algorithms. Given a cost model, we can find the best algorithm by choosing the one with the lowest costs. This requires calculating the costs for all available algorithms  $A$  for a data point  $x$  and then comparing them to each other.

$$\operatorname{argmin}_{a \in A} f_a(x) \quad (2)$$

There have been some cost-based attempts to select a lightweight integer compression algorithm based on its estimated compression rate [23, 28, 30], but these works do not consider our runtime objectives **SO2** to **SO5**. Damme et al. [9] have recently presented a novel cost model for lightweight integer compression algorithms. This cost model adopts a *grey-box approach* by explicitly modeling as much knowledge about the algorithms as possible and implicitly capturing the impact of data and hardware characteristics using a small number of calibration measurements for each algorithm. While this approach can generalize to a wide range of lightweight integer compression algorithms, a certain amount of manual effort is required to embrace a new algorithm. This still requires human engineering because more profound knowledge about the new algorithm and its inner workings is needed. Furthermore, making a single decision involves much

more calculations than traversing a decision tree. Given the multitude of available compression algorithms, scoring each of them to select the best one can become very expensive.

ML-based techniques have been introduced as a third large area of research for selection strategies. One common point for most ML approaches is that they are modeled as classification tasks and are not extendable with new algorithms. Additionally, they are trained on one data set and tailored to work on that particular use case. This reduces the generalization capabilities of the ML models and makes them data-dependent.

Boissier and Jendruk [4] employ regression models to select a suitable lightweight compression algorithm in Hyrise [12]. Unfortunately, crucial details of their approach are left unexplained in their short paper. Besides that, Jin et al. [21] model the selection problem as a classification task. During training, they exhaustively determine the best algorithm for a set of training data blocks. Then a test data block is assigned to the training data block with the most similar data properties and the known best algorithm for that training block is selected. A classification task like this is rigid and cannot easily be expanded to new algorithms.

Another example for ML-supported selection is CodecDB [20]. CodecDB uses a learned global model classification approach to find the best algorithm according to the compression rate. However, this is the only objective that this model considers. Other objectives that would be crucial for query performance, like **SO2** to **SO5**, are not modeled in the original paper, but for our evaluation, we extended the model to be usable with all objectives. The model is a neural network with 14 neurons in one hidden layer and hyperparameters as described in [20].

Furthermore, LEA [6] is an ML-based compression selection approach for column-stores. LEA trains models to predict the best column data representation for optimized query execution. LEA also uses synthetic data, like our generators, for training. However, like CodecDB, LEA currently only focuses on the data compression rate and also relies on cardinality estimates, which can negatively influence the optimizer due to their high estimation errors. Additionally, it uses sampling from the input data during the forward pass, which might slow down its application during runtime.

Modern hardware has also become the focus of compression acceleration. Especially the transfer of computations to the GPU seems promising [14]. In this work, faster compression is reached by using a compression planner to gain a speedup for GPU-based compression. In contrast, we propose a hardware-agnostic approach. Mainly, we use machines for data collection and model training with no co-located additional hardware, i.e., GPUs. Even our ML models can be trained very fast on a CPU and do not require a GPU, unlike other neural network approaches.

## 2.3 Lessons Learned

A large corpus of integer compression algorithms has been proposed [2, 24, 27, 33, 34, 36, 40] and no single-best algorithm exists [8, 9]. The best integer compression algorithm depends on data and hardware properties as well as on the objective [2, 8, 9, 24]. The selection of the best-fitting algorithm is essential to enable the smart use of lightweight integer compression in in-memory column stores. With all shortcomings of the available selection strategies discussed above, we argue that a *learned selection strategy* with a collection of smaller models trained on data-independent synthetic data offers a high-potential alternative. We see advantages in machine learning (ML) characteristics

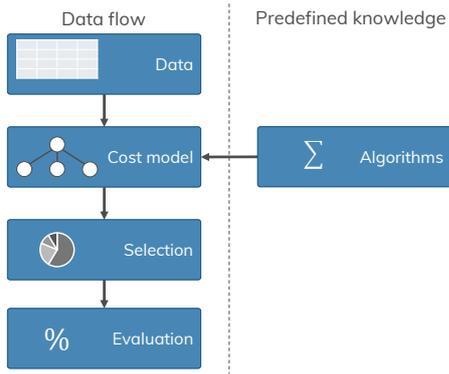


Figure 4: The cost-based selection strategy process.

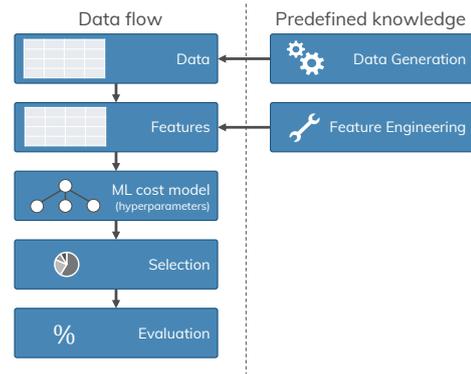


Figure 5: The learned selection strategy process.

because it requires less knowledge and effort for cost model engineering. With ML models, we are able to transform the high-quality grey-box approach by Damme et al. [9] to be adaptive to a more extensive collection of algorithms with a smaller overhead for deriving algorithm properties. Additionally, we can combine the expressive nature of the analytical cost model with the lightweight computation of decision trees by Abadi et al. [2] for a better and faster algorithm selection. We achieve all this without losing sight of the other challenges by solving them accordingly.

### 3 LEARNED SELECTION STRATEGY

As an alternative to existing selection strategies, we want to examine the potential of local ML models for a *learned selection strategy*. In general, ML opens up the possibility of generating more lightweight cost estimators more quickly with higher quality. So, it seems natural to model the cost functions for integer compression with a collection of ML models. Given the nature of such a cost model, a mapping from data properties to a continuous target variable, the modeling is called a *supervised regression problem*. However, an ML model must follow the same general process for selection strategies.

We deliberately decided not to establish the cost model as a classification problem as other ML selection strategies do [4, 21]. From the data properties, these classification models are *global models* predicting a vector with probability distributions over a set of target classes, i.e., the algorithms. The class or algorithm with the highest probability is then chosen as the best. Our primary concern is that the number of classes per *global model* is fixed. So, if we add a new algorithm, we need to rebuild and train the whole ML model from scratch. This process is depicted in Figure 2. With regression and a collection of *local models*, we can build a model for each objective and algorithm making the selection strategy expandable. Any new algorithm just adds a limited number of models—one for every objective and algorithm—with a small training effort. This is necessary because new compression algorithms are introduced or updated very often due to their iterative implementation process. Therefore, we support an ever-changing *pool of algorithms* within a system. A collection of  $|A| \cdot |\text{objectives}|$  models is called a *selection strategy design*. This approach also makes the decision process partially explainable. Given a pool of algorithms, one can identify the predicted cost of every single model and draw conclusions similarly to traditional approaches. The *local model* approach for a *selection strategy design* is shown in Figure 3. *Local models* have been proven to work

in other areas of database optimization, like learned indexes [22] or cardinality estimation [26, 39].

Given the properties of ML models, we require some changes to the general selection strategy engineering process. Altogether, we can establish the basic process for a selection strategy, as defined in Equations (1) and (2). An overview of this process is depicted in Figure 4. Given the data and algorithms, a cost model calculates the costs for all sample and algorithm combinations. Then, we choose the algorithm with the lowest costs for each sample separately. The algorithms and their behavior are seen as predefined knowledge that is often complex and deeply incorporated into the cost model by preliminary human input. The adapted process for an ML approach can be found in Figure 5. For ML-based approaches, the complex integration of algorithms is not necessary anymore. Instead, we defined two new steps: *data generation* and *feature engineering*:

- **Data generation** is used to sample representative example data for our supervised learning problem. Supervised learning always requires labeled data exposing the same properties as the data in the evaluation. Additionally, the use of synthetic training data makes our approach data-independent. It is important that generated and evaluation data are disjoint.
- **Feature engineering** transforms and reduces the data properties to key representations. These can be statistical or derived properties.

Furthermore, transforming the task into an ML problem requires several prerequisites and naming conventions. In ML, the cost value is called *target value* and the derived data properties are called *features*. Any supervised ML model has two phases: a *training phase* and a *forward pass*. In the training phase, the model is shown example data from which it learns a function to predict the target value. This function is applied to new or unknown data in the forward pass to calculate its target value. The forward pass is usually much faster than the training phase by some orders of magnitude. Therefore, training should only be done once, whereas the forward pass can occur often, which is beneficial during model application because it adds only minimal overhead to the decision-making. To get the most out of the training phase, the example data should be representative, especially for the model to abstract to new or unknown data. Therefore, data generation should produce representative synthetic data. We show how we generate example data in Section 3.2. The other important part is feature derivation. We present our *feature engineering* in Section 3.1.

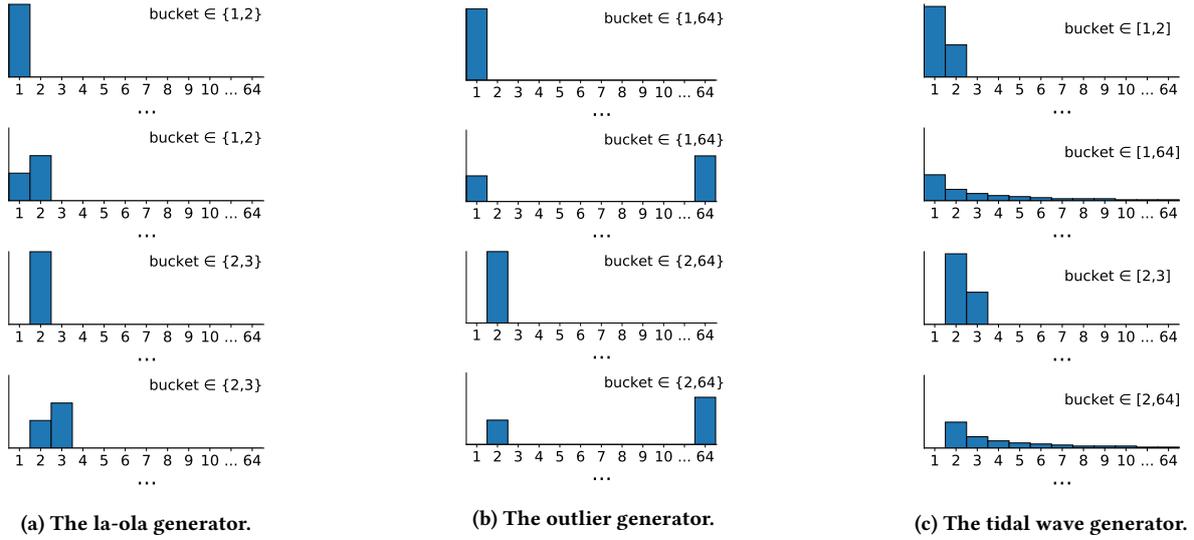


Figure 6: Different data generator techniques.

With the trained local ML cost models, we receive predictors  $\hat{f}_a$ , which try to approximate the actual costs of an algorithm  $f_a$ . So, we need to adapt our definition of a selection strategy to the predictors.

$$\operatorname{argmin}_{a \in A} \hat{f}_a(x) \quad (3)$$

The quality of the selection directly depends on the quality of the models for each algorithm. Errors are only introduced by the wrong cost model estimates and not by the selection strategy itself.

### 3.1 Feature Engineering

Feature engineering is an essential part of our locally learned model setup pipeline. Here, data representations are formulated given by varied instructions. We opted for the standard properties of lightweight integer compression algorithms because their influences are generally well-studied. So, it is easiest to take into account all properties of the algorithms and then derive features for our learned approach.

The first step we use is the *blocking* of data into chunks with a fixed number of values each. This is a state-of-the-art procedure in in-memory column-stores [10, 23]. Given the different lengths of columns or integer arrays, this is required because blocking reduces the input to a fixed size for integer compression. Moreover, this means that our models only work on these blocks. However, this is also an advantage because it enables our selection strategy to work on single segments of a column or data array at a time. This allows for a different compression algorithm for every block of a column.

To process whole columns, the selection strategy needs to be called on every block of the column. This does not lead to a performance loss because the forward pass is very fast, as our evaluation will show. Additionally, the model needs only to be trained on block-level meaning the problem abstraction is much lower than it would be for individual length input.

However, blocking is not enough to generate meaningful input. A common abstraction in integer compression is to use *bit width histograms* (bwhist) [10]. They collect integer values with the same bit width into buckets and aggregate a collection of integers

into a single histogram distribution. Therefore, the histograms have a fixed length for a given data type, i.e., 64 buckets for 64-bit integers. Bwhists further reduce the level of abstraction for an ML model. They reduce the complexity of any collection of integers to a number of bucket values. In our case, we model any input as a 64-bit bwhist where the  $m$ -th bucket contains the percentual share of the  $m$ -th bit width in the data.

Again, bwhists are not the final feature set. With 64 values, bwhist are still very large as input for small, efficient ML models. Fortunately, lightweight integer compression algorithms are very well studied and documented. Their inner processes can be directly associated with different data properties. So, particular important data properties for the algorithms' performance can be derived. We identified the following complete set of single value features from other works in the compression research area, which argue through their experiments that some of these features are most important for compression [8, 9].

- (1) Is the data sorted? (*sorted*)
- (2) Minimum bit width in the data (*minBucket*)
- (3) Maximum bit width in the data (*maxBucket*)
- (4) Percentage of data in the min. bucket (*Min*)
- (5) Percentage of data in the max. bucket (*Max*)
- (6) Average bit width of the data (*Avg*)
- (7) Number of filled buckets (*numBuckets*)
- (8) Standard deviation of the data (*Std*)
- (9) Skew of the data (*Skew*)
- (10) Kurtosis of the data (*Kurt*)

The last nine features can be directly derived from the bwhists. However, we need to inspect the data for the first feature, but most column stores store this information in their metadata. Our evaluation will show that this set of features is enough to produce low errors for all objectives and with competitive runtimes for training and forward passes. We also ran experiments to explore the possibilities of reducing the feature set, but the runtime of all optimization methods did not justify the marginal quality improvements. Therefore, we use the complete set of features for all experiments.

### 3.2 Data Generation

Besides the derivation of features, data generation is an important aspect of supervised models. Some even say that obtaining high-quality training data is the most crucial part of designing ML models for data processing [25]. The main contribution of data generation is to generate synthetic data sets that can be used during training and resemble data distributions that we expect to occur in real-world data. This makes our approach data-independent because the models are able to generalize from a well-defined synthetic data set to any other data set. So, we need to generate artificial data sets representing any unknown data. For this, we need to test several generation techniques to find the best one.

Additionally, every training example needs to be run for each integer compression algorithm at least once on the target hardware system to get the necessary labels for all objectives. This is an expensive operation, even for smaller data sets. However, this is necessary to capture the hardware behavior and properties in combination with the integer compression algorithms. Therefore, the main objective is to use as little data as possible to shorten this execution but to use as much representative data as possible to improve the model’s quality. To find this balance and the best generalizing data-independent generator, we incorporate three different generation techniques, each representing another idea to generate data with increasing complexity. The three approaches are:

- (1) La-ola<sup>1</sup>
- (2) Amount of outliers
- (3) Tidal wave

As mentioned in Section 3.1, the generators do not directly produce integer data but different *bwhists* from which the data and features are derived. All generators are presented in Figure 6. Note that it is not possible to generate every possible *bwhist* because a 64-bit *bwhist* where every bucket can assume values between 1% and 100% (percentual share) has  $100^{64}$  different combinations. Reducing the number of values per bucket to 10% brackets and grouping the buckets in pairs of two only reduces the number of combinations to  $10^{32}$ . So, our generators reduce this problem space even further.

The first generator, the *la-ola*, is a cycling bucket generator. For each new *bwhist*, a fixed portion of a bucket is redistributed to the next highest neighboring bucket. This happens until the bucket is empty. Then, the next bucket, which was getting filled before, is emptied. Therefore, the first *bwhist* must have a filled first bucket with otherwise empty buckets. Figure 6a details this process for the first two buckets. The *la-ola* generator models edge conditions, i.e., overflow of buckets into the next higher bit width.

The second generator tries to model distributions with outliers. Outliers are larger integer values than the rest of the integer sequence. This scenario is important for integer compression algorithms because a high amount of outliers might introduce the relocation of data to a different level in the memory hierarchy, i.e., from the cache to the main memory. With a data-independent approach, we want to be able to cover the eventuality of this use case. For this, it uses the same approach as the first generator, but instead of putting the portion of a bucket into the neighboring one, it always uses the last bucket. After emptying a bucket, the next higher bucket is emptied into the highest bucket. For our

use cases, the highest bucket has a bit width of 64. This *outlier* generator produces distributions that represent highly skewed data, as pictured in Figure 6b.

The last generator, called *tidal wave*, is a cycling Zipf distribution. For an increasing number from one to 64 buckets, a Zipf distribution is sampled in these buckets and then used as a *bwhist*. If one run from one to 64 is completed, the start bucket is increased and the process is repeated until every bucket has been the start bucket. Figure 6c shows the generations where the first and second bucket are used as the initial buckets. Over several iterations, the Zipf distribution is stretched over an increasing number of buckets for the first bucket. The same is repeated for the second bucket. This gives us a wide range of different *bwhists* modeling integer distributions where specific bit widths are more distinctive without losing the focus on outliers.

With these three generators, we want to examine the potential to find representative integer data. This data should produce high-quality learned models but also have as few distributions as possible to reduce performance load. Every integer compression algorithm needs to be executed with every example distribution on the target hardware system to be useful. So, every distribution we generate without quality gain can be directly attributed to a performance loss. Conversely, every distribution that brings quality improvement but is not generated is a quality loss. We show in our evaluation how the different generators influence this balance and how well they generalize to different data sets.

### 3.3 Training and Hyperparameters

An important part of learned approaches is their training phase. Usually, this step for setting up a global model is very time-consuming. Even though generating the example data takes the longest time of the training process (cf. Section 3.2), learning the model still takes some additional time. To keep training as short as possible, we decide not to use neural networks (NNs) because they would induce long training times. Additionally, we want to generate as little example data as possible and NNs perform best with lots of training data. Given these prerequisites, we decided to use Gradient Boosting (GB) as local models. This technique has the advantage of being lightweight because its modeling is based on weak predictors (i.e., decision trees) that have a fast training phase and forward pass. Every weak predictor is trained on the residuals of its predecessors. Generally speaking, for  $P$  predictors  $F$  our cost models is derived as:

$$\hat{f}_a(x) = \sum_{p=1}^P \lambda_p F_p(x) + c \quad (4)$$

In our case,  $F$  are the tree predictors. These weak predictors have two parameters that mainly influence model quality: the number of estimators  $P$  and the maximum depth per tree estimator  $d$ . Any change in these has an impact on the model’s final quality.

As a result, we need to find the best combination of  $P$  and  $d$ . This is called *hyperparameter tuning* and is done by repeating the training of the ML model with different parameter combinations. A validation data set is used to assess the model’s quality with every parameter combination and the best model configuration is chosen. The validation data has to be disjoint from the training data. We can directly incorporate the hyperparameter tuning into the training setup because we see two advantages. Firstly, our local models have a very low single training time, so repeating the training is not as time-consuming as it would be for NNs. Secondly, we only have two parameters to keep track of limiting

<sup>1</sup>Spanish: *the wave*, also: *stadium wave*

**Table 1: Rel error[%] for objectives (avg over all algorithms).**

Generator	compr ram2ram	decompr ram2ram	compr cache2ram	decompr ram2reg	compr rate
la-ola	11.81%	2.32%	17.80%	19.20%	12.84%
outliers	13.70%	1.23%	17.48%	16.82%	16.74%
tidal	<b>6.03%</b>	<b>1.17%</b>	<b>7.68%</b>	<b>6.87%</b>	<b>7.05%</b>

**Table 2: Rel error[%] for algorithms (avg over all objectives).**

Generator	static bp	dynamic bp	group simple
la-ola	14.22%	10.55%	11.34%
outliers	7.71%	11.89%	19.95%
tidal	<b>1.02%</b>	<b>5.04%</b>	<b>11.25%</b>

the search space significantly. Therefore, we can still train all necessary models in seconds and report smaller overall errors.

Furthermore, the tree structure of the chosen python implementation<sup>2</sup> also has a very small memory footprint compared to other ML models. The trees are stored as one-dimensional float arrays giving us the advantage of less memory consumption. Furthermore, the *sklearn* implementation transfers some of its components to C positively influencing its performance. A detailed empirical discussion of all this can be found in our evaluation.

## 4 EVALUATION

In this section, we evaluate the primary methods of our *learned selection strategy* for integer compression algorithms, like the data generation and hyperparameter tuning, and the overall performance of our approach against other techniques. For the first part, we extensively compare the results of the three different generators when used for training ML models and discuss their performance. Additionally, we detail the final hyperparameter configurations, the total runtime of the hyperparameter tuning and training, the time for the forward pass, and the memory footprint of the final models. This is done for each algorithm and each objective, as defined in our learned selection strategy setup. Next, we show the performance of our approach on 201 real-world data sets from the publicBI benchmark [15] compared to a baseline, the cost model of Damme et al. [9], and the ML-based strategy of CodecDB [20]. Additionally, we show the flexibility of our approach by adding algorithms to our selection strategy design and evaluating the performance and the required training times. The last experiment looks at the modeling of hardware properties by using our approach on a different set of hardware.

To assess the quality of our *learned selection strategy*, we need some indicators. We use three standard metrics for error evaluation. Firstly, there is the relative error for a model under a given algorithm  $a$  on a data set  $X$  consisting of  $|X| = n$  samples.

$$rel(a, X) = \frac{200\%}{n} \sum_{i=1}^n \frac{|\hat{f}_a(X_i) - f_a(X_i)|}{|\hat{f}_a(X_i)| + |f_a(X_i)|} \quad (5)$$

This is equal to the Symmetric Mean Absolute Percentage Error (SMAPE) between the original cost for every objective  $f_a(x)$  and the cost estimated by the model  $\hat{f}_a(x)$ . The second metric is the accuracy: Out of  $|X| = n$  samples, how many times did the

strategy predict the correct algorithm?

$$acc_A(X) = \frac{\text{true positives}}{n} \quad (6)$$

The higher the accuracy, the better the selection strategy. The last indicator is the loss or slowdown. This metric shows how much performance we lose due to misclassification. If our strategy does not select the correct algorithm, we want to quantify the loss we get for any objective. From our motivational experiment, we argue that misclassifications are not severe if the slowdown is low. For a single sample  $x$ , the slowdown is defined as the Symmetric Absolute Percentage Error (SAPE) between the costs of the best algorithm  $f_b(x)$  and the costs of the chosen algorithm  $f_a(x)$ .

$$SAPE(b, a, x) = 200\% \frac{|f_a(x) - f_b(x)|}{|f_a(x)| + |f_b(x)|} \quad (7)$$

The slowdown is only defined for  $a \neq b$ . For a complete data set  $X$ , we require the best ( $B$ ) and chosen algorithms ( $A$ ) for all samples and average the SAPE to receive the SMAPE for the slowdown.

$$SMAPE(B, A, X) = \frac{200\%}{n} \sum_{i=1}^n \frac{|f_{A_i}(X_i) - f_{B_i}(X_i)|}{|f_{A_i}(X_i)| + |f_{B_i}(X_i)|} \quad (8)$$

With these metrics, we can compare our *learned selection strategy* to other strategies and evaluate its overall quality.

### 4.1 Setup

All ML models are trained on an Intel(R) Xeon(R) Gold 6136 system with 64GB memory. However, retrieving the labels for all objectives for the example data from the generators is executed on an Intel(R) Xeon(R) Gold 5120 system with 377GB memory. This roughly depicts a real-world scenario where the ML models are trained asynchronously on a different machine not to disturb operations on the system where they should be used after training. The generated labeling benchmarks have to be executed on the main system because we need their execution objectives on this particular hardware if we want to use our selection strategy on it.

We train our models only on the three data sets generated by our generators and test our models on two data sets: (i) the synthetic validation data from the work of Damme et al. [9], and (ii) the real-world test data, which is a reduced excerpt from the publicBI benchmark. Note that the validation data is a second test data set with different properties than the publicBI. We reduced the publicBI benchmark to ca. 1GB by removing duplicated and almost equal blocks. However, we still keep data from each of the 201 example data sets within the publicBI. So, we get a diverse distribution of bwHi sts and properties from real-world data where the mix of bit widths is challenging for any selection strategy. Additionally, we apply an order-preserving dictionary encoding to all data columns in the publicBI benchmark, which are not integers. Therefore, we are able to include the compression of floating point values and strings in a straightforward manner. Our local models can model even the high complexity of a float-to-integer dictionary encoding without losing expressiveness.

<sup>2</sup><https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting>

**Table 3: Evaluation benchmarks.**

generator	forward pass [ $\mu$ s] ( $\sigma$ )	samples	training time [s]	hyperparameter tuning [s]	setup time on target system [min]
la-ola	26 ( $\pm 1$ )	3.151	0.24	4.8	898
outliers	26 ( $\pm 1$ )	3.151	0.24	4.8	899
tidal	26 ( $\pm 1$ )	2.080	0.24	4.8	600
Damme et al.	1126 ( $\pm 1651$ )	64	-	-	11
CodecDB	56 ( $\pm 13$ )	3.151	13.1	-	898

With this processing step, `bwHists` can be applied to all data types and the feature derivation is then applied in the same way. This allows us to evaluate our approaches over all columns and data types in the publicBI data set.

We consider three state-of-the-art lightweight integer compression algorithms from the null suppression (NS) technique as examples. For further experiments, we also included a delta encoding and frame of reference implementation. All five algorithms are vectorized using AVX-512, assume 64-bit uncompressed data elements, and are broadly used [10]:

*Static bit packing (static bp)* represents all data elements in a data set using the number of bits required for the largest data element. This algorithm is straightforward but cannot adapt to local variations in the data distribution.

*Dynamic bit packing (dynamic bp)* divides a data set into blocks of 512 data elements each and represents all data elements in the block with the bit width of the block’s largest value, i.e., an individual bit width is chosen per block.

*Group simple* packs as many data elements as possible into a 512-bit vector register, whereby a common bit width is chosen for a variable number of data elements. In fact, dynamic bit packing and group simple are ports of SIMD-BP128 [24] and SIMD-Group-Simple [40], respectively.

*Delta encoding (DELTA)* only saves the differences between each successive integer value in the data.

*Frame of reference (FOR)* also saves a differences instead of data values. However, in contrast to DELTA, it calculates these differences from a fixed reference point for each data point.

Due to the local model approach, our *learned selection strategy* design requires an individual model for every combination of algorithm and objective. We evaluate the five important objectives from Section 2 for the three NS compression algorithms mentioned above. This leads to 15 models for every run during training, hyperparameter tuning, and the same number of forward passes during model application. Despite this overhead, we show that our strategy is still performing efficiently.

## 4.2 ML Evaluation

For comparing the different generators, we use the average relative error once over all three algorithms and once over all five objectives. Note that this is not the slowdown but the direct relative error for the costs of the 15 models. Tables 1 and 2 contain the relative error on the validation data set for the three generators. The error is consistently averaged over the dimension, i.e., the algorithms or objectives, which are not presented. From these tables, we can derive that it is possible to generate (user-)data-independent decisions with our generated data. Additionally, the tidal generator produces the models with the highest quality. However, even though the la-ola generator is the simplest one, it

also performs reasonably well. Another result is the poor performance of the outlier generator. In conclusion, this generator does not deliver synthetic representative data for the training. From this evaluation, one can say that the tidal generator produces representative data and should be used further to train models. We will show its direct performance for our selection strategy on the validation and test data in Section 4.4.

## 4.3 Hyperparameters and Model Properties

To run the hyperparameter optimization for gradient boosting, we need to define the search space over the two parameters *estimators* and *depth*. The range for the number of estimators is [10, 50] with a step size of 10 and the range for the depth is [3, 6]. This generates a search space of  $5 \cdot 4 = 20$  parameter combinations. During the hyperparameter tuning, for each of the three generators, each of the 20 configurations spawns a new training process and yields an individual model trained on the training data. Then, for each of the three generators, we select the best of those 20 models, i.e., the one with the lowest relative error on the validation data. These errors can be found in Tables 1 and 2. We detail more information about the models’ properties in Table 3. The table contains for each generator, the cost model by Damme et al., and CodecDB: the forward pass per sample, the number of training samples, the total duration of the hyperparameter tuning, and the required time to run the examples on the target system.

So, for each generator, the resulting 15 models—one for each combination of the three algorithms and five objectives—have, on average, a faster forward pass than the competitors. One forward pass for one model takes 26  $\mu$ s. This is faster than most competitors’ forward passes and also independent of the algorithm or objective. The cost model’s forward passes heavily differ for different algorithms and objectives. Additionally, the models are so lightweight that a single training only takes 0.24s. In sum, we need 4.8s to train a whole model including hyperparameter tuning, which is still faster than just the training of the global CodecDB model. The hyperparameter tuning always takes the same amount of time because, for each model, it has to check all 20 different parameter configurations. However, the setup times of all ML approaches are longer than for the cost model because we require the label retrieving on the target system.

A very important note we want to address here is that these setup times are always necessary to generate training data for any supervised ML approach and only need to be run once. For example, using NNs would also require using the synthetic data and its execution on the target hardware system.

All the presented advantages stem from the good-natured properties of gradient boosting and the use of local models for our use case. The lightweight weak tree estimators show high

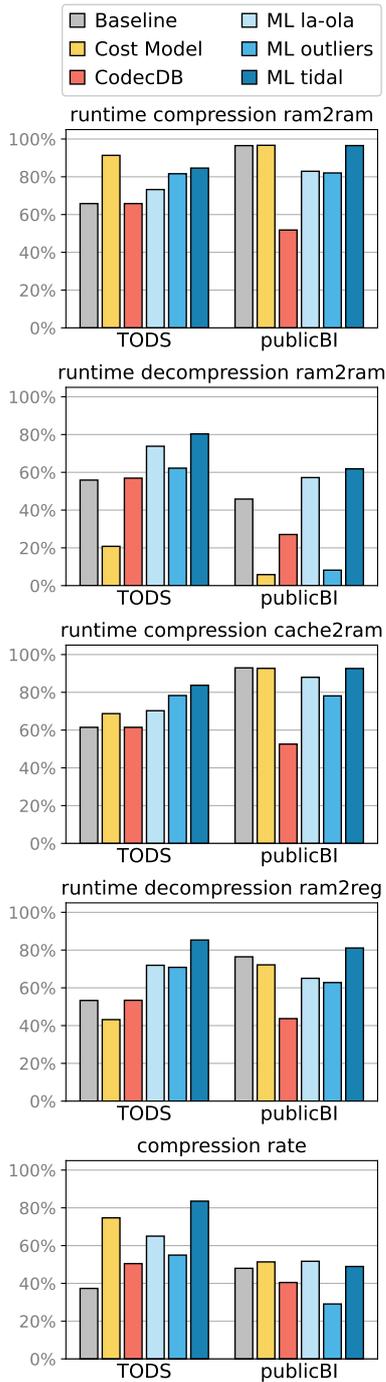


Figure 7: Accuracy (higher is better).

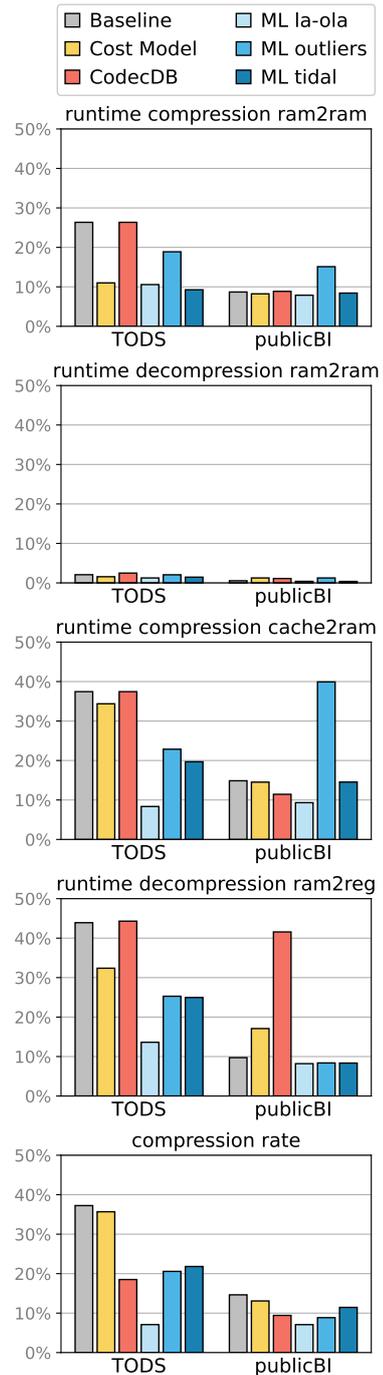


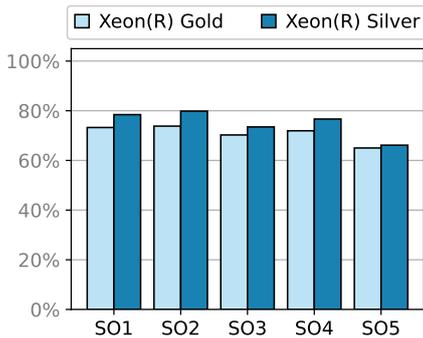
Figure 8: Slowdown (lower is better).

efficiency in training, the forward pass, and memory consumption. Our models require 18MB of memory, which is less than 15 equivalent NNs would use.

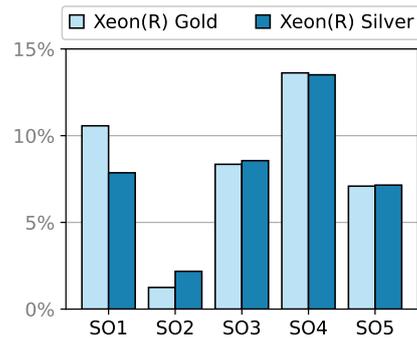
#### 4.4 Comparison

To show its applicability, we compare our *learned selection strategy* to two classical approaches and one learned global model approach. The first one is a *baseline* method, which chooses the simplest algorithm *static bp* for each block we want to compress, whereby the block size is 2,048 values in all experiments. The next

one is the complex cost model introduced by Damme et al. [9]. Note that the original cost model by Damme et al. [9] does not directly define the compression cache2ram and decompression ram2reg selection objectives. However, as the authors mention in [10], we modified the cost model by employing appropriate bit width profiles for the first objective, while the latter can be approximated by using the aggregation runtime objective of the original cost model. The last one is the classification-based global model from CodecDB [20]. This allows us to directly compare the benefits of a local model approach to a state-of-the-art model.



(a) Accuracy for different hardware (higher is better).



(b) Slowdown for different hardware (lower is better).

Figure 9: Scenario: New Hardware.

Table 4: Scenario: New Algorithms.

algorithm	training time per model [s]	training time total [s]	decision time total [ $\mu$ s]
NS	4.8	72	79
+DELTA	4.8	96	105
+FOR	4.8	120	131

We extend the CodecDB model to include the additional objectives SO2 to SO5, which were not included in the original paper. This is done by generating one classification model for every objective. To evaluate our approach properly, we use both the validation (TODS) and test data (publicBI) and report the accuracy and slowdown for all six selection strategies. From Figures 7 and 8, we derive that our approach is better than the three comparing selection strategies with some restrictions.

From the accuracy evaluation in Figure 7, we can derive that our *learned selection strategy* with the tidal generator produces the best (highest) results. It beats its competitors in most cases, except for some objectives for the publicBI benchmark. Even then, the other approaches are only marginally better. The global model from CodecDB suffers from overfitting and therefore generates results that cannot be generalized to the test data. In general, the accuracy of choosing the right algorithm for all data blocks varies widely across the board of techniques. Contrarily, the slowdown evaluation in Figure 8 shows a different story. Here, the la-ola generator outperforms all other approaches by yielding smaller slowdowns. There is one exception for the runtime compression ram2ram for the publicBI data. Again, we argue that these differences are only marginal. The global model shows qualities between the traditional approaches and our local models with a wide range of measured slowdowns. This is consistent with the accuracy evaluation and is again an indicator of overfitting and missing model expressiveness compared to our local models.

All in all, the tidal generator performs better in terms of accuracy and the la-ola generator is better for the slowdown. This means that with the tidal generator, our *learned selection strategy* chooses the right algorithm more often, but if it misclassifies, the resulting slowdown is more severe. This differs from our initial findings from Section 4.2, but we think a lower slowdown is more important than a higher accuracy. For a more comprehensive overview of all data presented for the NS algorithms in this section, we highlight all metrics in Table 5.

#### 4.5 Scenario: New Algorithms

In this scenario, we take the data from the la-ola generator and train local models for two additional algorithms: one DELTA and one FOR algorithm. For testing, we used the TODS validation data. With the additional two algorithms, our selection strategy has two more possible choices for each decision. Table 4 details the cumulative times for expanding the number of algorithms in the selection strategy. Adding a single ML model takes 4.8s per combination of algorithm and objective including hyperparameter tuning. This sums to 24s per additional algorithm for all five objectives SO1 to SO5. The combined forward passes through all models in the pool generate the total cumulative decision time for one entire decision as presented in Figure 3. The maximum decision time over all five algorithms is 131  $\mu$ s. Adding a new algorithm only increases the decision time linearly by 26  $\mu$ s. The DELTA and FOR algorithms are seldom the best selection for any objective. Therefore, the overall accuracy and slowdown stay the same as in Figure 7 and Figure 8. With such a low training time at the same levels of quality, our learned selection strategy is flexible and can adapt to a changing environment with different integer compression algorithms. Additionally, the introduced overhead through the additional forward passes that come with each new algorithm is very small and does not slow down the decision process. We argue that these properties stem from the *local model* approach and cannot be reached easily with a *global model*.

#### 4.6 Scenario: New Hardware

In the last scenario, the whole collection of algorithms, including DELTA and FOR, is tested on different hardware. However, different hardware is only used for collecting labeled training data and applying the learned selection strategy. For this, the hardware contains an Intel(R) Xeon(R) Silver 4214R with 125GB memory. The training is done asynchronously on the same hardware as for the other experiments. The training data again is the la-ola generated data set and the test data is the TODS validation data set. Figure 9 shows the transfer of the selection strategy to the new hardware with an Intel(R) Xeon(R) Silver. For all objectives SO1 to SO5, as defined in Section 2.2, the overall levels of accuracy and slowdown are similar to or better than the ones on an Intel(R) Xeon(R) Gold. This is because the Xeon(R) Silver is the newer CPU with better performance properties. Therefore, we argue that our approach can model properties required for integer compression over different hardware.

**Table 5: All results from the ML evaluation.**

generator	data set	objective	accuracy	slowdown	accuracy	slowdown	accuracy	slowdown
			baseline	baseline	cost model	cost model	ML	ML
la-ola	TODS	compr ram2ram	65.81%	26.35%	91.35%	11.00%	69.27%	10.59%
		decompr ram2ram	55.89%	2.07%	20.81%	1.56%	72.22%	1.31%
		compr cache2ram	61.46%	37.44%	68.71%	34.37%	70.16%	8.98%
		decompr ram2reg	53.32%	43.91%	43.17%	32.38%	72.78%	13.96%
		compr rate	37.32%	37.24%	74.70%	35.68%	64.36%	7.21%
	publicBI	compr ram2ram	96.51%	8.69%	96.67%	8.23%	45.93%	9.55%
		decompr ram2ram	45.84%	0.54%	5.83%	1.25%	64.71%	0.36%
		compr cache2ram	92.42%	14.87%	92.69%	14.52%	90.41%	13.41%
		decompr ram2reg	76.43%	9.72%	72.17%	17.09%	62.79%	8.38%
		compr rate	47.97%	14.63%	52.36%	13.08%	51.14%	6.74%
outliers	TODS	compr ram2ram					81.62%	18.88%
		decompr ram2ram					62.21%	2.05%
		compr cache2ram					78.30%	22.86%
		decompr ram2reg					70.86%	25.28%
		compr rate					54.96%	20.57%
	publicBI	compr ram2ram					82.03%	15.11%
		decompr ram2ram					8.19%	1.25%
		compr cache2ram					78.11%	39.91%
		decompr ram2reg					62.79%	8.38%
		compr rate					29.08%	8.85%
tidal	TODS	compr ram2ram					84.61%	9.26%
		decompr ram2ram					80.36%	1.45%
		compr cache2ram					83.72%	19.86%
		decompr ram2reg					85.31%	24.96%
		compr rate					83.54%	21.81%
	publicBI	compr ram2ram					96.53%	8.42%
		decompr ram2ram					61.84%	0.37%
		compr cache2ram					92.61%	14.54%
		decompr ram2reg					82.10%	8.35%
		compr rate					48.93%	11.45%

## 5 DISCUSSION

In this paper, we have shown that our *learned selection strategy* works data-independently with local models and little human engineering. Our ML models are better than competing approaches when using the right training data. This data was chosen carefully by comparing different generators representing different sets of data distributions. The *local model* approach makes the selection strategy extendable by design, which is essential for the dynamic life cycles of compression algorithms in systems. This also applies if we switch to other hardware. Through their fast forward passes, our learned approach can be used as a selection strategy as they add minimal overhead during query optimization. Our *learned selection strategy* greatly reduces the human effort in setting up new models because, with the presented generator-benchmark-training pipeline, we can automate the addition of any other integer compression algorithm to our selection design.<sup>3</sup>

We see three potential future extensions of our work, which are out of the scope of this paper. Firstly, our selection strategy also supports cascades of different compression algorithms. Here, every cascade generates a new local model and is handled just like a single algorithm, which works out of the box with our approach. Secondly, we could adapt our approach to other data type compression, like floating point. However, this would require new feature engineering and data generation to obtain the same

levels of quality and automatism as for the integer compression. Lastly, another step forward would be integrating our *learned selection strategy* into an existing database system. A candidate would be MorphStore [10] from our motivational experiment in Section 1.1. However, a full integration and evaluation are beyond the scope of this paper.

## 6 CONCLUSION

Data compression has been a well-established query optimization technique in database systems and has recently experienced a revival. Unfortunately, there is no single-best integer compression algorithm depending on data as well as hardware properties. Thus, in this paper, we presented an ML-based *learned selection strategy* with local models to select the best-fitting integer compression algorithm. Our *learned selection strategy* is characterized by three properties: (i) a (user-)data-independent strategy by training on synthetic representative data, (ii) a fast strategy by providing a decision with low computational and human effort, and (iii) an extendable strategy by using local models.

## ACKNOWLEDGMENTS

This work was partly funded by (1) the German Research Foundation (DFG) by a Reinhart Koselleck-Project (LE-1416/28-1), and (2) the European Union’s Horizon 2020 research and innovation program under grant agreement No. 957407 (DAPHNE).

<sup>3</sup><https://github.com/lucaswo/learned-selection-strategy>

## REFERENCES

- [1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280. <https://doi.org/10.1561/19000000024>
- [2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27–29, 2006*. ACM, 671–682. <https://doi.org/10.1145/1142473.1142548>
- [3] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. 283–296. <https://doi.org/10.1145/1559845.1559877>
- [4] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*. OpenProceedings.org, 674–677.
- [5] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. <https://doi.org/10.1145/1409360.1409380>
- [6] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *Fourth Workshop in Exploiting AI Techniques for Data Management (aiDM '21)*. Association for Computing Machinery, New York, NY, USA, 32–35. <https://doi.org/10.1145/3464509.3464885>
- [7] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query Optimization In Compressed Database Systems. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21–24, 2001*. ACM, 271–282. <https://doi.org/10.1145/375663.375692>
- [8] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21–24, 2017*. OpenProceedings.org, 72–83. <https://doi.org/10.5441/002/edbt.2017.08>
- [9] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46. <https://doi.org/10.1145/3323991>
- [10] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410. <http://www.vldb.org/pvldb/vol13/p2396-damme.pdf>
- [11] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query Optimization in Oracle 12c Database In-Memory. *PVLDB* 8, 12 (2015), 1770–1781. <http://www.vldb.org/pvldb/vol8/p1770-das.pdf>
- [12] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauk, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *EDBT*. 313–324.
- [13] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Found. Trends Databases* 8, 1–2 (2017), 1–130. <https://doi.org/10.1561/19000000058>
- [14] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 670–680. <https://doi.org/10.14778/1920841.1920927>
- [15] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*. www.cidrdb.org, 7.
- [16] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23–27, 1998*. IEEE Computer Society, 370–379. <https://doi.org/10.1109/ICDE.1998.655800>
- [17] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *ICDE*. 370–379. <https://doi.org/10.1109/ICDE.1998.655800>
- [18] Google. [n.d.]. Snappy - A fast compressor/decompressor. <https://github.com/google/snappy>.
- [19] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (1952), 1098–1101.
- [20] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data*. 843–856.
- [21] Yingting Jin, Yuzhuo Fu, Ting Liu, and Lan Dong. 2019. Adaptive Compression Algorithm Selection Using LSTM Network in Column-oriented Database. In *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 652–656.
- [22] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.
- [23] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326. <https://doi.org/10.1145/2882903.2882925>
- [24] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.* 45, 1 (2015), 1–29. <https://doi.org/10.1002/spe.2203>
- [25] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2022. Database Gyms. In *13th Annual Conference on Innovative Data Systems Research (CIDR '23)*. Amsterdam, The Netherlands, January 8–11, 2023, Online Proceedings. www.cidrdb.org, 8. <https://www.cidrdb.org/cidr2023/papers/p27-lim.pdf>
- [26] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. *Proceedings of the 26th International Conference on Extending Database Technology (2023)*, 273–284. <https://doi.org/10.48786/EDBT.2023.22>
- [27] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. 2015. Vectorized VByte Decoding. *CoRR abs/1503.07387* (2015), 7. arXiv:1503.07387 <http://arxiv.org/abs/1503.07387>
- [28] Piotr Przymus and Krzysztof Kaczmarski. 2014. Compression Planner for Time Series Database with GPU Support. *Transactions on Large-Scale Data- and Knowledge-Centered Systems* 15 (2014), 36–63. [https://doi.org/10.1007/978-3-662-45761-0\\_2](https://doi.org/10.1007/978-3-662-45761-0_2)
- [29] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091. <http://www.vldb.org/pvldb/vol6/p1080-barber.pdf>
- [30] Alexander Rasin and Stanley B. Zdonik. 2013. An automatic physical design tool for clustered column-stores. In *EDBT*. 203–214. <https://doi.org/10.1145/2452376.2452402>
- [31] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (1993), 31–39. <https://doi.org/10.1145/163090.163096>
- [32] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Record* 22, 3 (1993), 31–39. <https://doi.org/10.1145/163090.163096>
- [33] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. In *DaMoN@SIGMOD*. 34–40. <https://doi.org/10.1145/1869389.1869394>
- [34] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. 2011. SIMD-based decoding of posting lists. In *CIKM*. 317–326.
- [35] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O’Neil, Patrick E. O’Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 – September 2, 2005*. ACM, 553–564. <http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf>
- [36] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2018. Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action. In *ICDE Workshops*. 96–101. <https://doi.org/10.1109/ICDEW.2018.00023>
- [37] Ross N. Williams. 1991. An Extremely Fast Ziv-Lempel Data Compression Algorithm. In *DCC*. 362–371. <https://doi.org/10.1109/DCC.1991.213344>
- [38] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic Coding for Data Compression. *Commun. ACM* 30, 6 (1987), 520–540. <https://doi.org/10.1145/214762.214771>
- [39] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management*. 1–8.
- [40] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33, 3 (2015), 15:1–15:28. <https://doi.org/10.1145/2735629>
- [41] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory* 23, 3 (1977), 337–343. <https://doi.org/10.1109/TIT.1977.1055714>
- [42] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59. <https://doi.org/10.1109/ICDE.2006.150>