Learning over Sets for Databases

Angjela Davitkova
angjela.davitkova@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

Damjan Gjurovski
damjan.gjurovski@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

Sebastian Michel
sebastian.michel@cs.rptu.de
RPTU Kaiserslautern-Landau
Kaiserslautern, Germany

ABSTRACT

In this work, we consider using deep learning models over a collection of sets to replace traditional approaches utilized in database systems. We propose solutions for data indexing, membership queries, and cardinality estimation. Unlike relational data, learned models over sets need to be permutation invariant and able to deal with variable set sizes. The proposed models are based on the DeepSets architecture and include per-element compression to achieve acceptable accuracy with modest model sizes. We further suggest a hybrid structure with bounded error guarantees using guided learning to mitigate the inherent challenges when working with set data. We outline challenges and opportunities when dealing with set data and demonstrate the suitability of the models through extensive experimental evaluation with one synthetic and two real-world datasets.

1 INTRODUCTION

Recently, many approaches have suggested the improvement of data structures for relational data by substituting or tuning them with deep learning models. If data preprocessing and parameter tuning are done right, learned models can provide several benefits over traditional approaches. The prevailing research in this area mainly focuses on learned models over structured data, specifically relational datasets [9, 10, 14, 23].

In this paper, we address the problem of learning over sets which, in contrast to relational data, requires models to be independent of the order of the elements within a set and to support sets of different sizes. Examples of set data include (flat) JSON documents, DNA sequences, tagged resources like images, and hashtags contained in Twitter tweets. Typical tasks over such resources involve keyword queries and statistics computation, particularly over subsets of the underlying sets. For instance, keyword queries over a collection of hashtags are posed daily by millions of users searching through trending topics to discover new and relevant content. Performing analysis and gathering statistics over such query logs can bring significant benefits for data analysts which can directly use these insights to improve the user experience. To support such tasks efficiently, we extend the idea of using learned models for operating on sets and specifically propose solutions for *membership queries, indexing,* and *cardinality estimation.*

For example, consider the collection of four tweets (sets) of hashtags shown in Figure 1. To determine the popularity of the hashtags in the collection, one can create a learned model specialized in estimating the number of occurrences of the hashtag subsets. For the given example query $Q = \{\#pizza, \#dinner\}$, the model answers that the query is present in three sets, i.e., T_1, T_3, and T_4. As a different task, one can consider filtering tweets depending on some particular hashtags. For answering such membership queries, the approach will give information about the presence or absence of the respective query. A model can also be created to produce an estimate corresponding to the position in the collection of sets where the query subset appears, corresponding to a traditional index.

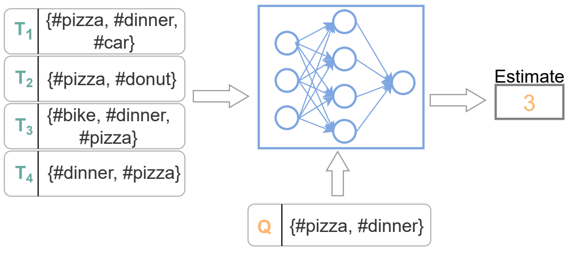


Figure 1: Example for cardinality estimation over a sample collection of four sets of hashtags.

Working with a collection of sets leads to new challenges. Unlike relational data, the nature of set data requires a different approach than handling records of ordered elements. More precisely, models that operate on set data need to be *permutation invariant* and to be able to deal with *variable set sizes.* Furthermore, when considering the problem of indexing a collection of sets, another challenge emerges. The collection of sets is often stored in an *arbitrary order* since an order over the sets cannot be easily defined. Consequently, using any index structure over a collection of sets has the task of mapping an item in an unordered list, which is challenging, especially for a deep learning model.

Finally, creating deep learning models requires appropriate training data. Related research in this field often assumes that such training data is already gathered and sometimes even processed before training the model. Intuitively, the presence of training data largely facilitates the creation of well-tuned replacement models. However, *creating training data* is not a straightforward task. This is especially apparent when working with set data (and multidimensional data), as it is often required to create combinations of the sets. Moreover, the problem of proper training data generation increases with the collection size and the maximal size of the sets.

To address the before mentioned problems and use the benefits of deep learning models, we propose, explore, and test the limits of learned approaches for replacing data structures when working with a collection of sets. Our approach is based on the DeepSets [24] architecture, which produces models that are permutation invariant and can deal with variable set sizes. To create models smaller than traditional data structures, we employ a *per-element compression strategy* [1, 2]. Combining these two techniques requires a modification of the DeepSets architecture to preserve the permutation invariant property.

Furthermore, many indexing approaches work by finding a position of a key in a sorted sequence. Handling keys that are not previously sorted is challenging, and simple regression models cannot solve this. Therefore, we propose a *hybrid approach with* *an iterative training process* to overcome the challenging learning procedure. The hybrid structure incorporates a learned model and a traditional structure, whose creation is guided by the learning error of the model. The hybrid structure has error-bounded guarantees and can be used for the tasks of indexing and cardinality estimation. Furthermore, multiple local errors are stored to improve the search through the hybrid structure, binding the sequential search to smaller ranges.

1.1 Problem Statement

We consider a collection $S = [X_1, X_2, ..., X_N]$ of N sets, where each set $X_i = \{x_1, x_2, \ldots, x_k\}$ contains elements x_i, where $1 \leq k \leq M$ and M is the number of unique elements appearing in the sets of the collection. While S can contain duplicate sets, each set X_i does not contain duplicate elements. A query q is given as a subset of elements from any of the sets of S.

We consider three tasks, indexing sets for subset and equality queries, cardinality estimation, and membership queries:

- The indexing task for a query set q is to determine the first position i in S such that q is a subset of X_i.
- The cardinality *card* of query q is the number of sets S_q in the collection in which the query set q appears.
- A membership query q answers whether the query set is a subset of any of the sets in S, i.e., $\exists X_i \in S : q \subseteq X_i$.

1.2 Contributions and Outline

The main contributions of this paper are as follows:

(1) We formulate the problem of learning over a collection of sets by using a learned supervised approach for the tasks of indexing, cardinality estimation, and membership queries involving any of the subsets of the considered sets (Section 4).
(2) To create compact models, we utilize per-element compression (Section 5).
(3) We propose a hybrid structure by guided learning and outlier removal (Section 6), allowing the best possible application of learned models for cardinality and indexing.
(4) We thoroughly analyze the approaches (Section 7) and report the results of a comprehensive experimental study over one synthetic and two real-world datasets (Section 8).

2 RELATED WORK

Models over Sets: DeepSets [24] is a deep neural network where the model's input and output can be sets. Set Transformer [12] is another architecture learning over sets. It is an attention-based neural network consisting of an encoder and decoder, designed to model interactions between elements in a set. In our work, we use DeepSets due to the faster execution time and smaller memory footprint required for replacing the data structures.
Learned Index Structures: Kraska et al. [10] first propose a recursive learned model performing a regression task as a replacement of a B-Tree and a hash index. Following this idea, multiple improvements suggest different architectures and models [7], discussing the possibility of updates [4] and proposing learned models suitable for multidimensional data [3, 5]. Unlike our approach, related work has an easier task as it leverages the monotonic relation between the keys and their positions as it either considers one-dimensional or metric multidimensional data that can be ordered or scaled.
Learned Cardinality Estimation: Dutt et al. [6] use neural networks and tree-based ensembles for selectivity estimation of multidimensional range predicates. Woltmann et al. [21] suggest a local-oriented approach to improve estimates. Naru [23] is an unsupervised data-driven synopsis using deep autoregressive models and a Monte Carlo integration technique called progressive sampling. An extension of Naru is NeuroCard [22] which addresses join cardinality estimation in relational databases. Hasan et al. [8], use autoregressive models and supervised models as cardinality estimators. Others [11, 15, 16, 18] instead of estimating cardinalities focus on reinforcement learning for optimal plan generation. Differently, LMKG [2] considers both supervised and unsupervised models for cardinality estimation in knowledge graphs. In this paper, we utilize the compression introduced for LMKG to build our approaches for learning over sets.

The DeepSets architecture has also been previously used in the task of cardinality estimation. Kipf et al. [9] create a multi-set convolutional network, termed MSCN, for cardinality estimation over relational data. In MSCN, the query representation consists of multiple modules (sets), tables, joins, and predicates, where for every module, a different DeepSet neural network is used. Different from our work, they focus solely on relational data and cardinality estimation. Similarly, Lu et al. [13] create table summaries by using DeepSets per table to summarize over rows, treating the row as an element in a set. Later, the summary, which resembles a conditional autoencoder, is used to compute the query cardinality. Unlike their work, we use DeepSets per set and do not create a summary over multiple sets (table). Our model directly predicts cardinalities and does not resemble a conditional autoencoder. Furthermore, as MSCN, they focus solely on cardinality estimation over relational data and we explore different applications over sets.
Learned Bloom Filter: Kraska et al. [10] propose the Learned Bloom Filter (LBF) as a replacement for traditional Bloom filters (BF) for improved performance and memory savings, at the price of losing updates and the introduction of false negatives. For a query, the classification model computes the probability of the presence or absence of the query. Sandwiched LBF [17] adds another BF before the LBF to improve the performance. Vaidya et al. [20] use multiple LBFs based on classification score segments. Neural Bloom Filter [19] uses memory-augmented neural networks to work with input that arrives at high throughput. Others [1, 14] extend the idea of LBF for multidimensional relational data. Although replacing indexes, cardinality estimators and Bloom filters with their learned counterparts has been widely researched, so far, the problems of indexing, answering membership queries, and estimating cardinality over a collection of sets have *not* been considered.

3 PRELIMINARIES

3.1 Why Relational Learned Models Fail?

Neural networks for sets and relational data are different because they operate on different inputs and require input-specific architectures to model the relationships between the elements. Consider the collection of sets: $[\{A, B, C\}, \{D\}]$. To adapt existing learned models over relational data, we have to first *fix the input size to the maximal set size* and *sort the sets* according to a prespecified order. This leads to creating as many embedding matrices as the maximal set size, i.e., number of columns. For the example collection, we need 3 embedding matrices, all containing an embedding for the same elements (A, B, C, D) to be able to represent each element at a different position. Because the relational learned model has a fixed input size, to allow

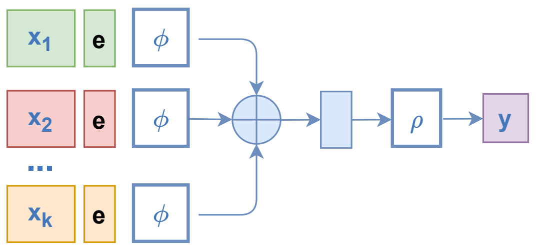


Figure 2: DeepSets architecture where x_i are set elements, e is an embedding or an encoding, ϕ and ρ are suitable transformations.

for variable set sizes, we have to introduce a special element *. Thus, for every set, we have to learn over the combinations. For $\{A, B, C\}$ we need to learn over: $\{A, B, C\}$, $\{A, B, *\}$, $\{B, C, *\}$, $\{A, C, *\}$, $\{A, *, *\}$, $\{B, *, *\}$, $\{C, *, *\}$. Consider the sets $\{A, B, *\}$ and $\{B, A, *\}$. The relational learned model cannot recognize that they are the same, as the elements appear at different positions. A naive solution is to create all possible permutations (incl. *), which is extremely challenging, even for smaller sets.

To solve the above-mentioned problems, we propose a solution that works directly on sets by using and extending the DeepSets architecture. DeepSets allows us to have a single shared embedding, perceiving the elements as same no matter in which position they appear. It further allows variable set sizes without the need for a special element.

3.2 Permutation Invariant Architecture

In contrast to techniques operating on vectors with fixed dimensions and fixed order, by definition, the size of a set or order of the elements in a single set is not specified. Therefore, when considering a model over sets, the main requirements are handling *variable set sizes* and learning a *permutation invariant* function f that transforms an input X (set) into y. Consequently, the output y should not be impacted by any different ordering of the same elements in the input data, i.e., a single set.

One of the existing model architectures capable of dealing with these requirements is the DeepSets [24] architecture. DeepSets states that a function over a set X is a permutation invariant function iff it can be decomposed in the form $\rho(\sum_{x \in X} \phi(x))$ and the elements are from a countable universe. The functions ρ and ϕ are suitable universal approximators that can be learned. The architecture of DeepSets is depicted in Figure 2. Given a set X, each element x_i from X is independently embedded and transformed into a representation $\phi(x_i)$. The embedding parameters are shared for all elements of the set. The independently transformed elements are then aggregated with any permutation invariant pooling operation, e.g., max, mean, sum, or log-sum-exp. The shared embedding parameters, together with the permutation invariant function, produce the same output regardless of the order of the elements in the input. The aggregated set representation is then propagated to another neural network ρ that considers interactions between elements, using further nonlinear transformations. DeepSets naturally supports a variable number of elements in the set and is proven to be a universal set approximator under specific conditions [24].

DeepSets stores a shared embedding for all elements in the collection. The size of the embedding is impacted by the embedding dimension and the number of elements. Therefore, having a large number of elements greatly limits its application as a substitute for traditional database structures, as the embedding matrices will contribute to a large memory footprint.

Table 1: DeepSet setting for the different tasks.

Task	Model Type	Activation Func.	Output	Loss
Index	Regression	Sigmoid	Position	Q-Error
Card. est.	Regression	Sigmoid	Cardinality	Q-Error
Bloom filter	Classification	Sigmoid	Existence	Binary Cross-Entr.

The depicted set problem is termed a **set-to-vector problem**, where we want to learn the probability of a vector y conditioned on specific elements in the set, i.e., $p(y|X)$, not impacted by the order of the elements in X. One common problem of this type is point cloud classification. The set-to-vector problem is also explored by other approaches, like Set Transformers [12]. Although the Set Transformer has a slightly better accuracy than the DeepSets model for some more complicated tasks, for simpler tasks, they perform similarly. However, the DeepSets model is superiorly faster and smaller, which is crucial when replacing traditional data structures and is thus our architecture choice.

In the following, through the set-to-vector problem, we define how to improve and create learned Bloom filters, index structures, and cardinality estimators when dealing with set data and thoroughly discuss the advantages and the encountered challenges.

4 LEARNED DATA STRUCTURES FOR SETS

Data in the form of a collection of sets is widely present. Graph network analysis often represents nodes and edges as sets. Social media platforms represent user followers or the groups a user belongs to as a set. E-commerce websites use sets to represent products a customer has purchased, allowing recommendations of other products. In machine learning and computer vision, an image can be seen as a set of pixels, a video as a set of frames, or a document as a set of words. Compactly storing and analyzing sets is vital in all these scenarios.

We consider the set-to-vector problem in the form of **two model types applied to three database tasks**, summarized in Table 1. Depending on the output y, the set-to-vector problem is formulated as a **classification or regression problem**. Both models use the DeepSets architecture but differ in the final prediction, loss function, and training data. We will consider the regression model as a learned counterpart of a cardinality estimator or an index because, *for a given input, it predicts a real value corresponding to cardinality or index position*. A learned parallel to the traditional Bloom filter is a classification model that *returns the probability that a particular element is present or absent in a collection*. In the following, we will show how DeepSets can be used for these tasks.

4.1 Learned Set Index

We create a learned index structure over an unordered collection of sets that can be used for two scenarios. The index structure can answer equality queries where it will return the position i of the query set q in the collection of sets. As a second search type, the index structure can provide the first position i where the query set q appears as a subset in the collection of sets S, i.e., $q \subseteq S[i] \land \neg\exists j, q \subseteq S[j]$ where $j < i$.

We use the DeepSets architecture, where the transformation ρ (Figure 2) is a neural network having as input a permutation-invariant representation of the set and outputs a real value corresponding to the set position. During training, the DeepSets model receives subsets of the original sets and target values y corresponding to the position of the subsets in the collection S. The position is log-transformed and scaled using the minimum and maximum observed positions during training, i.e., $y_i = \frac{y_i - min(y)}{max(y) - min(y)}$. This makes it suitable for the sigmoid function $f(x) = 1/(1 + e^{-x})$ used as an output of the neural network. The position of the first or last subset occurrence can be easily generated by iterating through the collection of sets. For the regression task, the chosen loss is a relation between the estimate and the actual position, also known as the q-error, i.e., $q\_error(y, \hat{y}) = max(\hat{y}/y, y/\hat{y})$. Other losses, such as MSE and MAE, can also be considered.

4.2 Learned Set Cardinality Estimation

We define the set-to-vector problem in terms of a cardinality estimator, where given a query q representing a set of elements, we predict an estimate of its cardinality in the collection of sets S. More specifically, given a subset q, we estimate the number of occurrences of the subset in the collection of sets S, i.e., $|q \subseteq S[i]|$ for $0 \leq i < |S|$.

The transformation ρ outputs a real value, representing the cardinality. At the time of training, the DeepSets model receives as input subsets of the original sets and target values y corresponding to their number of occurrences (cardinality). The cardinality is log-transformed and scaled, with the minimum and maximum cardinalities. Knowing that a superset always has a cardinality equal to or smaller than that of the elements present inside it, the maximum observed cardinality is easy to detect and is always the largest cardinality of any individual element. As for the set index, the model uses sigmoid as an activation function, and the q-error as a loss function.

4.3 Learned Set Bloom Filter

We expand the idea of learned filters over a given collection of sets S, where we answer a boolean membership query q, which outputs whether a given set is a subset of any of the sets in S, i.e., $\exists i, q \subseteq S[i]$ for $0 \leq i < |S|$.

The model that performs a simple classification task mimics the behavior of a Bloom filter. In the DeepSets model, ρ outputs the probability of the presence or absence of the set. Therefore, the learned Bloom filter outputs whether a subset is present in any of the sets while also likely filtering negative subsets. Let S be the considered collection of sets. $S_{q_{pos}}$ denotes the collection of all subsets present in S. $S_{q_{neg}}$ denotes the collection of subsets that are not present in the sets of S. The training is performed over a given set of queries $S_q = \{(s_{q_i}, y_i = 1) | s_{q_i} \in S_{q_{pos}}\} \cup \{(s_{q_i}, y_i = 0) | s_{q_i} \in S_{q_{neg}}\}$, by minimizing the binary cross-entropy loss, defined as follows: $L = \sum_{(s_q, y) \in S_q} y \log f(s_q) + (1 - y) \log(1 - f(s_q))$. As in related work, to solve the problem of false negatives, we introduce a backup Bloom filter [10, 17].

5 COMPRESSED DEEPSETS ARCHITECTURE

In the DeepSets architecture, the set elements are represented through embeddings from a shared embedding matrix. Intuitively, the embedding matrix scales with the number of distinct elements in the collection of sets. When working with an extensive collection of sets having many distinct elements, the shared embedding matrix will consume a large space, counteracting the memory

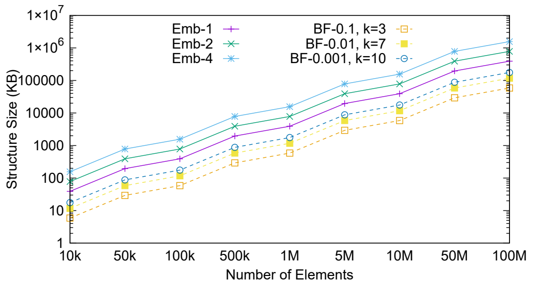


Figure 3: Size comparison of embedding vs. Bloom filter for different embedding dimensions and false positive rates.

benefit, especially in tasks that require replacing already compact structures, such as the Bloom filter. To better understand what this means, Figure 3 compares the size of the embedding matrix to the size of the Bloom filter for different parameter settings. The size of a Bloom filter depends on the false positive rate, the number of elements to be inserted, and the hash functions used. The size of an embedding matrix depends on the size of the vocabulary and the embedding vector size. For example, for a vocabulary of size 1000 elements and an embedding vector of size 100, the embedding matrix will be 1000×100. As expected, as the number of items increases, the size of both the embedding matrix and the Bloom filter also increases. Based on Figure 3, using a learned model with embeddings in its original state instead of a Bloom filter does not bring any benefits since, as the number of items increases, the Bloom filter always occupies less memory. If we use the architecture without any modification, we lose one critical advantage of the learned over the non-learned structures: having less memory. In some situations, even when the DeepSet model is used as a replacement for a B-Tree or a HashMap, the memory benefits are negligible.

To create more compact models, inspired by related work [1, 2, 22, 23], we modify the architecture to include *lossless input compression per element*. The details of the approach are presented in Algorithm 1. To compress an element into ns subelements, we rely on the compression technique introduced for the LMKG framework [1, 2]. To perform the compression, first, the elements of the sets need to be represented as integer values. When this is fulfilled, initially, the maximal integer value $max_{v_{id}}$ from the elements is determined. If the elements are split into $ns = 2$ subelements, we calculate the divisor as $sv_d = \lceil \sqrt[ns]{max_{v_{id}}} \rceil$. To compress an element x_i from the set X, we divide the element with the

Algorithm 1 Compression of elements

```
1:  function COMPRESS_ELEM_NS($elem$, $sv_d$, $ns$)
2:      $elem_c = elem$; $elems_c = []$; $i = 0$
3:      while i < (ns - 1) do
4:          $sv_q, sv_r = compress\_elem(elem_c, sv_d)$
5:          $elems_c.append(sv_r)$
6:          $elem_c = sv_q$; $i++$
7:      $elems_c.append(sv_q)$
8:      return $elems_c$
9:  function COMPRESS_ELEM($elem$, $sv_d$, $ns$)
10:     $sv_q = floor(elem/sv_d)$
11:     $sv_r = elem \% sv_d$
12:     return $sv_q, sv_r$
```

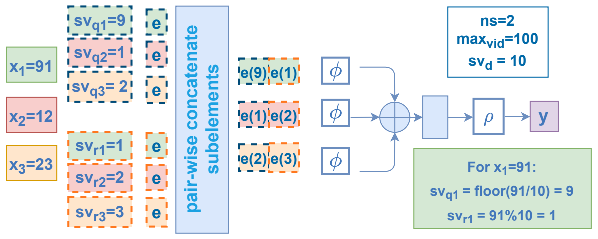


Figure 4: Compressed DeepSets architecture where x_i are set elements, sv_{q_i} and sv_{r_i} are compressed elements for x_i, e is an embedding or an encoding, ϕ and ρ are fitting transformations.

divisor sv_d and obtain the quotient sv_{q_i} and remainder sv_{r_i} (Algorithm 1, Lines 9–12). Correspondingly, the compression of set X of size k will result in $(sv_{q_1}, sv_{r_1}), (sv_{q_2}, sv_{r_2}), ..., (sv_{q_k}, sv_{r_k})$. For $ns > 2$, the same procedure is repeated until ns subelements are created by always taking sv_{q_i} as the next element to be divided with the divisor (Algorithm 1, Lines 3–6). A detailed example of the compression for $ns = 2$ and $max_{v_{id}} = 100$ for the set $\{91, 12, 23\}$ is depicted in Figure 4.

We modify the DeepSets architecture as depicted in Figure 4. Initially, every element from the set is decomposed into two subelements, i.e., quotient sv_q and remainder sv_r. All quotients share a single encoder, as do all the remainders. Every quotient (remainder) is independently encoded (embedded). Each of the independently transformed quotients is then concatenated with the representation of their corresponding remainders. For the example shown in Figure 4, this will result in $(e(sv_{q_1}), e(sv_{r_1}))$, $(e(sv_{q_2}), e(sv_{r_2}))$, and $(e(sv_{q_3}), e(sv_{r_3}))$.

Once concatenated, the pairs of elements are independently forwarded to the transformation ϕ. The transformation is crucial for capturing the interconnection between the two subelements. If this part is omitted, then we consider two sets of independent subelements, which leads to incorrect results and disrupts the DeepSets model. The output is then forwarded to a permutation invariant pooling operation, in this case, the sum, and finally forwarded to the transformation ρ. The modified architecture provides one crucial benefit, and that is the **reduction of memory**. This is because, instead of a large embedding matrix mapping the elements to an n dimensional vector as in the original DeepSets architecture, we decompose the elements into subelements, resulting in multiple embedding matrices whose sizes are drastically smaller than the initial one.

When considering the compression of relational data, columns have a given order. In this case, once a column is split into subcolumns, each subcolumn will be considered a separate feature. That means subcolumns are independently embedded, concatenated, and forwarded for learning. The order of the subcolumns itself imposes their interconnection. However, when working with sets and considering the DeepSets architecture, we cannot interpret every subelement as completely independent when we perform a compression over elements in sets. Due to the permutation invariance property of the model, such consideration of the subelements will lead to incorrect results. Thus, to obtain accurate results, we must preserve the interconnection between the subelements while maintaining the permutation invariance.

For clarification, let us consider a set X with elements x_1, x_2 which are compressed to $x_1 = (sv_{q_1}, sv_{r_1})$ and $x_2 = (sv_{q_2}, sv_{r_2})$ and set Z with elements z_1, z_2, which are compressed to $z_1 = (sv_{q_2}, sv_{r_1})$ and $z_2 = (sv_{q_1}, sv_{r_2})$, such that the set Z has not been given to the model for training or has a completely different estimate than X. The problem has now moved from a set of elements towards a set of pairs. Because our primary goal is to reduce the model size affected by the large element embedding matrix, having an encoding for all pairs will not yield any benefits since, still, the number of unique pairs is the same as the number of unique elements before compression. Therefore, all quotients (sv_q) will be sent to one shared encoder and all remainders (sv_r) to another one. Thus, we have bounded the size of the input dimension creating drastically smaller models. Next, the individual encodings sv_{q_i} need to be concatenated with their respective sv_{r_i} encodings to preserve the original subelement pairs. However, if we just apply a pooling operation, like sum, on them, we still would not preserve the interconnection between the quotients and the remainders. This is due to the permutation invariance property of the model. In other words, if the set X with elements $x_1 = (sv_{q_1}, sv_{r_1})$ and $x_2 = (sv_{q_2}, sv_{r_2})$ appeared, the model will assume that also set Z with elements $z_1 = (sv_{q_2}, sv_{r_1})$ and $z_2 = (sv_{q_1}, sv_{r_2})$ has appeared since the sum per dimension for x_1 and x_2 and z_1 and z_2 is equal. However, once decomposed, the sets will have different elements, and the model should have estimated a different value for Z. *To capture the interconnection between sv_q and sv_r, we forward them through a transformation ϕ*. Thus, we are certain that the sets X and Z are considered as different.

Finally, to emphasize the importance of the compression, let us revisit the initial motivation. Let us consider limiting the embedding size of the original model to the smallest possible size, i.e., *embedding_size* $= 1$. Additionally, we assume that there are 1000000 different elements in the set. The dimension of the embedding matrix will be 1000000×1. When using the compression for $ns = 2$ subelements, we create two embedding matrices of dimensions 1000×1 and 1001×1. Consequently, we substantially reduce the required memory and make it possible to consider the DeepSets model as a substitute for Bloom filters. The same remarks and conclusions hold if we consider the one-hot encoding.

The drastic reduction of memory from the compression comes at the cost of introducing additional complexity in the model due to the increased interconnection between the subelements that needs to be detected by the neural network. In our experimental evaluation, we show the trade-off between the model size and the accuracy and discuss when it is beneficial to consider the compressed model instead of the non-compressed one.

6 HYBRID STRUCTURE WITH ERROR BOUNDS

Typically, learned index structures assume a monotonic correlation between keys and their corresponding positions in the dataset, which is possible by sorting the keys in a specific manner. This advantage allows even small neural networks or simpler models, such as regression or interpolation, to capture data interdependencies efficiently. However, when considering more complicated scenarios, such as mapping keys to positions that do not preserve the monotonic relation, the complexity increases, producing a significant accuracy error or requiring enormous models, rendering the learned index impractical. These problems naturally appear when working with set data since sorting the keys and achieving an easier correlation is not possible.

To overcome these problems, we propose the use of **guided learning with outlier removal**. The resulting *hybrid structure*

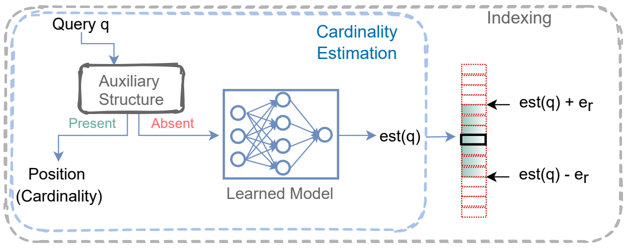


Figure 5: Querying the hybrid architecture consisting of an auxiliary structure, a learned model, and error bounds.

incorporates a learned model and an additional structure for storing the outliers. The inclusion of this auxiliary structure is beneficial because deep learning models typically provide approximate results before being corrected by specific error bounds. The benefits of the proposed structure are especially evident in the case of learned Bloom filters, where the backup structure is crucial for guaranteeing no false negatives. Backup structures are also useful in the case of updates. They maintain the updates until the retraining of the model is initiated. By incorporating an additional structure, we implicitly allow the inclusion of updates in our proposed approach.

Driven by the idea of solving the false negatives problem of the learned Bloom filter with a backup structure, we explore and suggest a generalization of an auxiliary structure beyond the learned Bloom filters. This additional structure serves as a correction strategy for remedying misestimates in **learned estimators and learned indices**. The auxiliary structure can be any traditional or learned structure already proven efficient in such scenarios but lacks the benefit of a small memory consumption. Thus, the hybrid architecture incorporates a learned model that captures as most of the data as possible and an auxiliary structure working on the data that creates a problem for the learned model.

Unlike the typical approach used in learned Bloom filters that assumes an already trained model for which it calculates the wrong estimates, we suggest an iterative guided training process to exclude outliers. We observe that the iterative training process and such exclusion of outliers allow us to apply the model as best as possible to more complicated, difficult-to-learn distributions under prespecified constraints, which balance out the memory consumption and accuracy of the approach. Since in the hybrid structure, the learning is affected by the data distribution, in the best case, when the learning error is small, guided learning renders only a learned model with prespecified error bounds and, in the worst case, a non-learned index structure.

The process of outlier elimination requires several parameters. Initially, the model learns for several epochs until the function can be estimated sufficiently. After a predefined number of epochs, the error is calculated for each subset. All the outliers or subsets whose prediction causes a larger error than a **given threshold** are removed from the training process and placed into the outlier structure. The threshold is guided by a defined error that we want to reach and can be set manually or automatically. The automatic setting is needed more for the indexing task, where we set the error to always reach a q-error in the range of [1, 1.4], depending on the dataset size. Since the goal is to achieve a small error, depending on the requirements, it is possible to continue the process in iterations. The whole process is tuned according to the balance we want to achieve between the memory consumption and the speed of the structure.

Algorithm 2 Search in the hybrid structure for indexing

```
1: function FULL_PREDICTION($set, min\_val, range\_length$)
2:     $res = auxiliary\_structure.search(set)$
3:     if $res$ is $None$ then
4:         $est_q = model.predict(set)$
5:         $r = ceil((est_q - min\_val)/(range\_length + 1))$
6:         $e_r = errors[r]$
7:         $res = search\_in\_range(set, est_q, est_q - e_r, est_q + e_r)$
8:     return $res$
```

For *searching through the structure*, we need to calculate the maximal error produced by the model. The maximal error is the maximal absolute error between the estimate and the real position of the subset X_i, i.e., $abs(est_{X_i} - pos_{X_i})$ calculated for every X_i in the collection of sets. This computation is necessary since the model gives an estimate rather than the correct value. To guarantee that we find the set X_i, we have to search within the specified error bounds $[est_{X_i} - max\_error, est_{X_i} + max\_error]$. However, calculating single error bounds over the whole dataset will lead to unnecessary scanning of large portions of the collection of sets. Let us assume that the model always predicts accurate results for the whole dataset except for one set X_j. Although the accuracy is almost perfect, during the search, the range boundaries will be defined by the largest error encountered, which in this case corresponds to the one for set X_j. This will lead to unnecessary scans and an increased execution time even though almost all the points are accurately predicted. Therefore, to avoid unnecessary scans, we calculate errors per equally sized ranges of the possible predicted values and achieve a drastic reduction in the search time. For the previous example, the large error produced by X_j will only impact the search for a small number of sets during query time rather than the complete indexed collection.

The resulting architecture, depicted in Figure 5, includes the learned model, the outlier backup structure, and the error bounds. The details of the search algorithm are shown in Algorithm 2. When querying, first, the auxiliary structure will be checked for the existence of the query set (Algorithm 2, Line 2). If the query set is not present, it will be forwarded to the model. The querying for cardinality is less involved because it requires only the prediction of the model. The index needs an additional local search around the predicted position (Algorithm 2, Lines 5–7). When performing an index search for query q, the range that needs to be searched is defined by the estimated position est_q and the local error bound, calculated as $e_r = errors[\lceil (est_q - min\_val)/range\_length + 1 \rceil]$. The resulting search range is $[est_q - e_r, est_q + e_r]$. The equality search starts from est_q and goes in both directions, whereas the search for the first position starts from the left position $[est_q - e_r]$.

7 CHALLENGES AND LIMITATIONS

When replacing traditional structures with learned models, we can encounter challenging scenarios, such as processing complex distributions that are difficult to learn, for which the model will produce large errors. In such cases, the hybrid structure will fall back to an auxiliary structure not having the benefits of the learned model. Additionally, for some of the tasks, especially visible for larger sets, the creation of the training data can be complicated. Finally, learned data components are sensitive to changes in the data distribution. Recreating the models is computationally expensive because the learned models already incur a larger creation time than the traditional structures. In the following, we discuss the main challenges, addressing the data preprocessing, the practical application of the approaches, and the changes in the data distribution.

7.1 Training Data Creation and Practical Application

7.1.1 Regression Model. While creating training data for the regression model, subsets from the sets and their respective cardinalities or index positions need to be generated. When considering the indexing task, we have to generate the complete dataset of subsets to guarantee that all subsets will be found. For cardinality estimation, subset generation is required since often the suggestion that a supervised model generalizes to unseen queries can be conflicting.

When working with set data, one can typically observe that many items are very infrequent or even appear only once. A prominent example are Twitter tweets, where the hashtag frequency distribution follows Zipf's law. As a consequence, the extracted subsets from such sets will have a skewed cardinality distribution, as the presence of a rare element will result in supersets that are also infrequent, i.e., have a cardinality of 1. We can make use of this information in the data generation. Therefore, when dealing with larger sets, one can limit the generation of all the training data if their subsets are already infrequent. For the datasets in the experiments, we observed that subsets above size six are already infrequent, and thus, we generate only the subsets up to this size.

7.1.2 Classification Model. When considering traditional Bloom filters, only the present elements need to be indexed by the filter. Differently, the learned Bloom filter, in addition to the positive training data, requires the negative training data, i.e., non-existing combinations of the existing set elements, to ensure the false positive rate and no false negatives. The positive training data $S_{q_{pos}}$ consists of not only the existing sets but also all the subsets of the present sets. The negative training data $S_{q_{neg}}$ consists of sets whose elements co-occurrence is not present as a subset in the given collection of sets.

Consequently, using the learned Bloom filter is most beneficial when there are already existing negative training samples. When they are not present in advance, the generation of negative training data is challenging as it has to provide enough representative sets which are not a subset of the sets in the collection. Although relatively trivial for one-dimensional data, in the case of set or ordered multidimensional data, generating such data is problematic since it creates a combinatorial problem. To overcome this problem, and following the assumptions that the queries are typically smaller than the maximal possible set size, the learned Bloom filter can be restricted to capture subsets until a predefined size. Even if this impacts the accuracy for larger sets, restricting the space of possible negative training data provides guarantees up to a specific set size.

Although the requirement of having negative training data in addition to the positive data limits the use cases for learned Bloom filters, it does not render them entirely useless. The model can be used in cases where the negative training data is present in advance, is gathered through time, or is small enough to be generated entirely. For instance, when working on the detection of malicious messages, the negative training data will contain the malicious messages that should be filtered. In contrast, the positive training samples can be drawn from the existing messages that already qualified as relevant.

Table 2: Datasets specification.

Datasets	**RW**			**Tweets**	**SD**
Details	$n = 200k$	$n = 1.5M$	$n = 3M$	$n = 1.9M$	$n = 100k$
Uniq. Elem.	30324	231954	346893	73618	5661
Max Card.	52905	638488	968112	513696	99280
Min/Max Set Size	2/8	2/8	2/8	1/> 10	6/7

7.2 Change in Data Distribution

Although static data is ideal for learned models, we next discuss how incremental updates are handled. To decide if the accuracy of the model is deteriorating, after a prespecified number of updates, the accuracy is measured. If a significant drop in the accuracy is detected, the models are retrained. While minor updates are not of high impact for the cardinality estimator and Bloom filter, for the set index, this impacts the correctness of the approach. Therefore, for the index task, updates can be handled by modifying the error boundaries used to find each of the indexed sets. However, this impacts the search time and will trigger the recreation of the model. We directly benefit from the auxiliary structure in the hybrid model by avoiding continuous retraining. Consider the update $\{A, B\}$ to $\{A, C\}$ for the set index. The index position for B and C in the sets will change. If the change is smaller than the current error boundaries, everything stays the same. If the new position is outside of the boundaries, the new subsets are inserted into the auxiliary index. Upon searching, the auxiliary index, already containing the updated version, is queried first. After a considerable number of updates, the whole structure can be rebuilt. When we want to avoid handling updates, i.e., prohibit the rebuilding of the model, the structure will eventually fall back entirely to the traditional index.

8 EXPERIMENTS

8.1 Experimental Setup

We implemented the models for our proposed approaches in Keras[1] following the DeepSets implementation [24]. We have performed the experiments on an NVidia GeForce RTX 2080 Ti GPU. In the following, we investigate the performance of the proposed learned data structures, and we report on experiments for both the non-compressed and the compressed (Section 5) models. For the evaluation, we varied the following parameters:

- the *embedding size* from 2, 4, 8, 16, to 32
- the *number of neurons* from 8, 16, 32 (Bloom filter and indexing task) and 64, 128, to 256 (cardinality estimation)
- the *number of layers* from 1 to 2

Since increasing *ns* creates more complicated patterns between the subelements that need to be learned, for the compressed models, we use embeddings with $ns = 2$ subelements. This setting for *ns* already produces much smaller models than the original one, with better accuracy and speed than models with larger *ns*. **Note that** the learned Bloom filter, learned index, and the learned cardinality estimator can be either non-compressed (**LSM**) or compressed learned set models (**CLSM**). To improve the models' performance by enabling them to efficiently handle outliers, for the cardinality estimation and indexing task, we generate the proposed hybrid structure using guided learning (**-Hybrid**). We measure the accuracy of the subsets of the sets by estimating the cardinality or index position. For each subset, we calculate

[1]The source code and datasets are available at https://git.cs.uni-kl.de/dbis-public/clsm/learning-over-sets-for-databases

the q-error using the estimate and the real value and group the results by result size. For the Bloom filter task, we measure the binary accuracy. For the non-learned competitors, the accuracy is one as they always return the exact result.

For all the considered tasks, the models are trained between 50 and 100 epochs. The training time in seconds per epoch, grouped per dataset and task, i.e., cardinality, indexing and BF (without compression, with compression), is as follows: RW-200k: (2.6, 2.6), (2.7, 2.8), (7.3, 6.5); RW-1.5M: (14, 8), (14.4, 8.3), (16, 10); RW-3M: (15, 8), (15.6, 9), (17, 11); Tweet: (5.6, 5.2), (5.8, 5), (12, 11); SD: (2.4, 2.5), (4, 4.1), (2, 2).

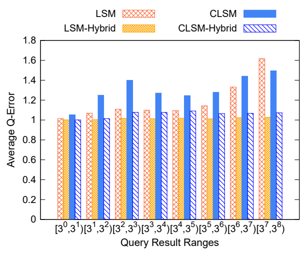
8.1.1 Datasets. The experiments were performed over *one* synthetic and *two* real-world datasets (cf. Table 2 for dataset statistics). The **Tweets** dataset consists of hashtags from a tweets excerpt that was gathered over a specific period of time through the public Twitter API. The size of the excerpt is 50 GB. The (**RW**) dataset consists of company server logs that contain information about file accesses and user logins. Three alternate versions of different sizes have been gathered on different days, containing a different number of unique elements, with at most 8 elements per set. The elements of this dataset are diverse, causing the cardinality to follow a skewed distribution, with most of the elements appearing only in a small number of sets. To test other distributions of elements, we have created a synthetic dataset (**SD**). SD is generated by randomly combining subsets of elements up to a prespecified size (6–7 elements) to demonstrate the effects of having fewer unique elements that appear often in different sets. Since the RW and Tweets datasets have sets of different sizes, we restrict SD to sets with similar sizes. With that, we also show how the compression works when the number of unique elements is smaller.

For the cardinality and indexing task, we generate all subsets of the sets as training data. The index requires all possible subsets to guarantee finding the actual position. The query workload for all datasets is created using subsets of the original sets having both few and many elements. For the Bloom filter task, the negative data contains combinations of elements not appearing in the original sets. However, creating all negative data is impossible. For this reason, the used negative training data is only a subset of the complete dataset.

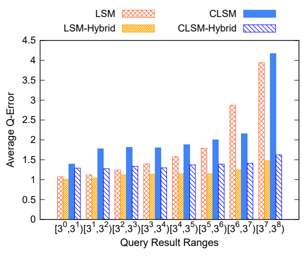
8.1.2 Competitors. We modify traditional structures to fulfill the permutation invariant property. To support sets, one can either concatenate sorted elements and hash them or use a permutation invariant hash function and index the combinations of set elements. All the competitors are implemented in Python as in-memory structures.

- **Cardinality estimation task:** Due to the permutation invariance property, existing learned estimators for relational data cannot be applied. Thus, for this task, we create combinations of the elements in the sets and store them in a HashMap.
- **Set index task:** We use a B+ Tree, where as a key we use a hash function over the set also allowing duplicate keys.
- **Bloom filter task:** We use the traditional Bloom filter where we index all the combinations of present elements.

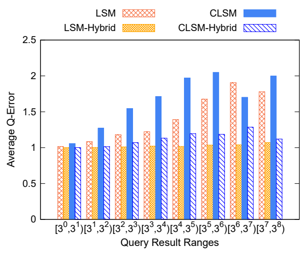
The creation time in seconds for the B+ Tree with branching factor 100 for the indexing task, the HashMap for the cardinality task, and the traditional Bloom filter with an fp-rate of 0.1 per dataset is as follows, RW-200k: (4.4, 1.5, 6.7); RW-1.5M: (39, 10, 44); RW-3M: (63, 18, 74); Tweet: (8.3, 3.2, 15); SD: (5, 1.6, 2.7).



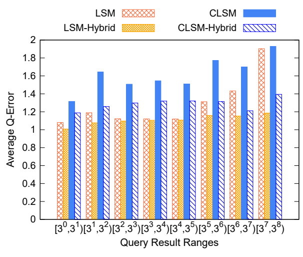
(a) RW-200k



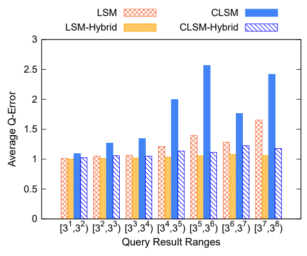
(b) RW-1.5M



(c) RW-3M



(d) SD



(e) Tweets

Figure 6: Accuracy (q-error) per query result size for cardinality estimation task.

8.2 Cardinality Estimation Task

8.2.1 Accuracy. We next show the accuracy of the cardinality estimation task by comparing the compressed estimator (CLSM) against the non-compressed (LSM) one. We further analyze the inclusion of an auxiliary structure that represents the hybrid version of the estimators.

Compressed vs. Non-compressed: To measure the accuracy when estimating cardinalities, we use the average q-error. The larger the q-error, the more far-off the estimated cardinality is. The results grouped by query result size ranges are shown in Figure 6. The models for each dataset are trained with a similar number of neurons and epochs. The outliers are removed at the same epoch step.

When analyzing the results for the RW collection (Figures 6a, 6b, 6c), it is visible that CLSM produces slightly less accurate estimations, evident for the larger datasets. However, for the outliers, i.e., larger query result size ranges, CLSM sometimes outperforms LSM. CLSM has mainly worse accuracy for the smaller query result size ranges. The reason for this is the increased input complexity which means that the model needs to learn many complex interconnections between the subelements. This specifically occurs in these ranges because most of the subset cardinalities belong to them. When the dataset size and the number of unique elements increase, both LSM and CLSM have a larger (worse) average q-error. For the SD (Figure 6d) and the Tweets datasets

Table 3: Memory consumption (MB) for cardinality estimation task.

Datasets	LSM	LSM-Hybrid	CLSM	CLSM-Hybrid	HashMap
RW-200k	3.817	4.917	0.079	1.079	32.644
RW-1.5M	29.021	37.721	0.156	8.456	249.263
RW-3M	43.388	61.588	0.182	17.082	429.891
Tweets	9.285	11.385	0.168	1.968	66.410
SD	0.791	1.791	0.119	1.119	30.123

Table 4: Execution time (ms) for cardinality estimation task.

Datasets	LSM	LSM-Hybrid	CLSM	CLSM-Hybrid	HashMap
RW-200k	0.08	0.053	0.09	0.07	0.00059
RW-1.5M	0.11	0.077	0.14	0.13	0.00054
RW-3M	0.19	0.180	0.20	0.19	0.00056
Tweets	0.179	0.168	0.20	0.19	0.00056
SD	0.174	0.163	0.196	0.183	0.00050

(Figure 6e), as for RW, CLSM has lower estimation accuracy than LSM. The difference in the estimation accuracy is especially noticeable for the higher ranges. When analyzing the estimation results and considering the memory (Table 3), it is clear that for SD, there is no need for compression since, without it, the model is of acceptable size and produces better estimates. Thus, when the number of unique elements is small, using a non-compressed model has clear benefits.

Hybrid vs. Baseline: In the following, we will discuss the benefits of having a hybrid structure for cardinality estimation. To show the benefits, as depicted in Figure 6, for each estimator, we also include the hybrid version (-Hybrid), constructed by removing the outliers responsible for the errors larger than the 90 percentile error. The outliers are removed at the same epoch step for both LSM and CLSM. Even removing a small number of outliers improves the accuracy of the approach but results in a slight increase in the memory footprint. In this scenario, the auxiliary structure represents a HashMap; however, other more compact structures can also be considered. When looking into the results produced by the hybrid structure, we observe that the accuracy drastically improves in all scenarios since the auxiliary structure produces exact results, and the model fits better to the data without the outliers. LSM-Hybrid produces the best results, followed by CLSM-Hybrid. CLSM generally produces much smaller models (Table 3). Thus, *CLSM-Hybrid is deemed the best option* when considering both accuracy and memory.

Comparison: Finally, we compare the learned set cardinality estimator with a traditional estimator. As other approaches do not work with sets, we store all the subsets from the sets together with their cardinality. Intuitively, since the HashMap stores all the possible subsets, the accuracy is always one. However, indexing all combinations occupies an enormous amount of memory (Table 3), which renders the HashMap structure unsuitable in many realistic applications.

8.2.2 Memory Consumption. We next show the memory consumption of the learned cardinality estimation models both with and without compression (Table 3), the hybrid structures with the same thresholds as in the accuracy experiments, and the HashMap when indexing all the possible combinations. From the model, we extract the weights, and together with the auxiliary structure, we store it in Python's pickle format. We do the same for the HashMap and report on the memory consumption of the files. When the learned set model takes the role of a cardinality estimator, both models are viable replacement options, each providing different benefits when considering the trade-off between memory and accuracy. From the depicted results, it is evident that the learned models are far smaller than the HashMap index. The hybrid structure introduces a small overhead while producing a much better accuracy.

8.2.3 Execution Time. To avoid the overhead from the used framework when measuring the *execution time of the queries*, we extract the weights from the model, and we do a pure numpy model implementation. We execute the estimators over 10000 queries and report the average execution time per query. We execute each query separately and not in batches to mimic the behavior of a real query system, although this is a disadvantage for the model. For the considered datasets and set sizes, we depict the results in Table 4.

The overhead of the element compression is small, resulting in a difference of 0.001 ms. The compressed model also has more complicated operations, such as the concatenation of the subelement embeddings, which impacts the execution time. This overhead is acceptable because of the size benefits of the compressed version over the non-compressed one. In comparison, a HashMap reaches a speed of around 0.0005 ms for each dataset, as the set is directly retrieved from the map. The search time is much faster than the models' prediction; however, this comes at the cost of a much higher memory footprint. The hybrid model, both for LSM and CLSM, almost always improves the query time. Thus, when a balance between the memory and the execution time is needed, *the hybrid model is the best option.*

8.3 Set Index Task

8.3.1 Accuracy. As next, we show the performance of learning over sets when considering the indexing task. All models are trained with a small number of neurons to allow for compact memory and faster execution.

Hybrid vs. Baseline: In Table 5, we show the accuracy upon comparing the two models using the average q-error and average absolute error when the outliers having an error above the prespecified percentile are removed (row named *Percentile Threshold*). To emphasize the importance of the hybrid structure, we also include the *No Removal* version, which is the model without using any auxiliary structure. When analyzing the hybrid model, it is observable that for some datasets (RW-200k and RW-3M), the model error is drastically reduced when only a small number of the data is placed in a separate (outlier) structure. Since the errors directly impact the number of sets that will be examined around the predicted position, the hybrid structure makes the model applicable in the indexing scenario. Although we can remove the outliers, when we have a restricted memory budget, the models cannot always learn each distribution to the required accuracy. This is the case for RW-1.5M, having element interconnections that do not follow specific patterns, where the error is still large and results in an increased sequential search for the index position. For such cases, the hybrid structure will fall back to the auxiliary structure. For the smaller datasets, such as SD, due to the small number of records, a learned index is not required, as the B+ Tree already performs well.

Compressed vs. Non-compressed: We next look into the performance of LSM versus CLSM and their suitability for the set indexing task. For many datasets, the accuracy is similar for both variants, keeping in mind that CLSM drastically reduces the

Table 5: Accuracy (q-error/abs-error) for index task.

Errors	Datasets	LSM-Hybrid					CLSM-Hybrid				
Percentile threshold		<50%	<75%	<90%	<95%	No Removal	<50%	<75%	<90%	<95%	No Removal
Avg. q-error	RW-200k	1.0012	1.0027	1.0050	1.0069	1.0538	1.0022	1.0050	1.0097	1.0155	1.0728
	RW-1.5M	1.0872	1.2247	1.4175	1.5449	1.9821	1.2448	1.5492	1.8836	2.0846	2.7380
	RW-3M	1.0041	1.0082	1.0162	1.0229	1.0651	1.0003	1.0005	1.001	1.0033	1.0203
	Tweets	1.0128	1.0292	1.0548	1.0774	1.2118	1.0413	1.0938	1.1978	1.2863	1.6057
	SD	1.2603	1.5509	1.8868	2.0891	2.6825	1.4177	1.8647	2.3709	2.6808	3.5656
Avg. absolute error	RW-200k	109.86	187.10	265.40	314.69	532.57	155.20	345.77	601.85	805.64	1654.40
	RW-1.5M	25618.50	60855.56	106977.40	132363.49	170479.19	69128.51	129852.72	195271.46	227799.38	269449.98
	RW-3M	6862.21	13748.58	29496.86	44187.90	72197.05	548.11	930.90	1749.29	5729.40	25262.81
	Tweets	2211.97	5837.48	12459.96	17736.75	33232.34	7771.05	18041.04	31840.64	42417.40	70110.22
	SD	2629.62	5384.69	8946.67	10983.82	13554.35	3630.74	6549.95	11299.99	13653.25	16407.14

Table 6: Impact of compression factor sv_d for index task for Tweets dataset.

sv_d	Full comp.	500	1000	5000	10000	No comp.
Accuracy (Q-error)	1.6	1.57	1.47	1.35	1.26	1.21
Memory (MB)	0.012	0.014	0.02	0.08	0.16	1.15
Training Time (s)	709	842	843	867	872	1071

Table 7: Memory consumption (MB) for index task.

Datasets	LSM-Hybrid	CLSM-Hybrid	B+ Tree
	Model/Aux.Str./Err.	*Model/Aux.Str./Err.*	
RW-200k	0.24 / 0.72 / 0.07	0.005 / 0.72 / 0.07	7.03
RW-3M	2.71 / 8.58 / 1.31	0.01 / 8.58 / 1.31	70.67
Tweets	2.3 / 5.05 / 0.59	0.02 / 5.04 / 0.59	12.33
SD	0.09 / 3.06 / 0.04	0.006 / 3.06 / 0.04	9.99

Table 8: Execution time (ms) for index task.

Datasets	LSM-Hybrid	CLSM-Hybrid	B+ Tree
RW-200k	0.36	0.31	0.004
RW-3M	4.02	5.4	0.006
Tweets	2.1	3.7	0.005
SD	4.02	4.4	0.004

models' size. However, as for the cardinality task, there is still a decline in the accuracy when using the compressed version. We would like to ideally have something in between these two spectrums, i.e., lower memory like CLSM but better accuracy like LSM. To achieve this, *we can tune the compression factor sv_d*. In the optimal case, the compression factor sv_d is set to achieve the maximal compression for a given *ns*. However, one can set sv_d to be any number between the best-case compression or no compression at all. To show the tunable compression, we depict the accuracy for $ns = 2$ by varying sv_d between its optimal case (most compression we can achieve) to no compression (Table 6). Increasing sv_d provides us with more parameters capable of expressing the interconnections better, leading to better accuracy. On the other hand, when increasing sv_d we also increase memory consumption. By tuning sv_d we can find a suitable sweet spot between LSM and CLSM.

Comparison: Finally, as a competitor with the learned set index, we consider a B+ Tree. As in the case of the cardinality estimation, this index produces exact results. Therefore, the most relevant part when comparing learned and non-learned approaches is comparing the memory and the speed reached by the model with the achieved accuracy.

8.3.2 Memory Consumption. When considering the model as a replacement for a set index, we notice that the memory is relatively small. However, we need more than just the model to reach a small enough error to consider it as a direct replacement. Therefore, we discuss the performance of the set index where we only consider the hybrid version of the models with errors specified per ranges of length 100. As error threshold percentiles, we chose 90 for RW-200k and RW-3M, 60 for Tweets and 70 for SD. As an outlier structure, we utilize a B+ Tree. The results in Table 7 inform that the hybrid structure drastically reduces space consumption. CLSM consumes the least amount of space, often less than 1 MB. The error list (Err.), which is currently set to range 100, also consumes a small amount of memory. It is important to note that the memory can be further reduced by increasing the range and storing fewer values. Naturally, this will impact the accuracy and speed of the approach. As depicted, most memory is consumed by the outlier structure (Aux.Str.). The hybrid structure for the RW-1.5M dataset falls back to the worst scenario, which is solely an auxiliary structure. For that reason, it is not present in the results.

8.3.3 Execution Time. We execute the indices over 1000 queries and show the average query execution time (Table 8). For the considered settings in Table 8, the hybrid structure results in an average between 0.3 and 4 ms for perfectly tuned models. For more complicated distributions where the model has to answer for most of the subsets, but the error is large (such as the RW-1.5M dataset), the local search can reach even 20 ms, which is unacceptable. In such scenarios, the hybrid structure does not reach the required accuracy and falls back to the worst case, which is solely a traditional index structure. The hybrid index is crucial when using a learned model for the task of indexing. The model by itself generates enough error producing a large execution time of around 4 ms for the smaller dataset and up to 100 ms for larger ones. Although using a larger embedding size can improve the performance of a deep learning model, having more embedding weights can lead to slower execution time and higher computational costs.

To show the compression benefits during training, we analyze the training time needed for 100 epochs by varying the compression factor sv_d for the Tweets dataset and depict the results in Table 6. Unlike the query time, the **training time** is positively impacted by the compression.

Local error vs. Global error: The larger execution time in the learned set index is caused by the sequential search around the

Table 9: Binary accuracy for Bloom filter task.

Datasets	RW-200k	RW-1.5M	RW-3M	Tweets	SD
LSM	0.9999	0.9999	0.9999	0.9998	0.9998
CLSM	0.9968	0.9987	0.9996	0.9711	0.9970

estimated position to find the actual position of the query. Thus, an essential improvement in the hybrid structure used for indexing is the introduction of local errors. For instance, for the RW-200k dataset, the maximal error is 171853. If we have the range size for the local error set to 100, the search is improved by now having an average error of 11901. By reducing the local error, we examine fewer sets on average, which drastically reduces the search time. In the basic case, the learned index will result in an enormous execution time, rendering the model inefficient as a possible replacement for the indexing structure. Intuitively, the smaller the range, the better the prediction time, at the cost of a slightly larger memory consumption.

8.4 Bloom Filter Task

8.4.1 Accuracy. To test the behavior of the models for replacing Bloom filters, we create positive training data consisting of subsets present in the given sets and a sample of negative subsets consisting of combinations of elements not co-occurring in the sets. As mentioned, the generation of the complete negative training data is not possible, and thus, the negative samples are limited to a specific size. Both LSM and CLSM have embeddings of size two and two layers, each having eight neurons. We chose a smaller model so both LSM and CLSM can compete with the Bloom filter when considering the memory. In Table 9, we show the binary accuracy after 50 epochs. It is observable that in all scenarios, LSM and CLSM perform extremely well, reaching an almost perfect accuracy. As for the before-mentioned tasks, LSM is able to capture the element interdependencies better and, thus, produce better accuracy. **Intuitively, if we consider only the training sets, both models perform exceptionally well.** However, since the complete negative training data cannot be generated, the false positive rate cannot be bound. This is expected as we must create all subsets to ensure that the model can correctly estimate them. Still, in scenarios where both negative and positive training sets are present, a learned Bloom filter is useful as it produces highly accurate results.

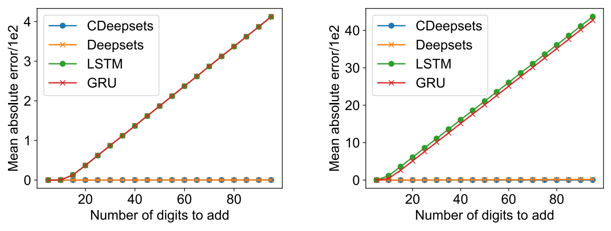
8.4.2 Memory Consumption. We next discuss the memory of the learned models in the role of Bloom filters. The results from the comparison are depicted in Table 10. **Note that the structure of the learned Bloom filter includes a backup structure. The memory of each backup structure is negligible, and it does not impact the comparison.** We can see that the learned non-compressed Bloom filter yields a much larger memory than any of the compressed Bloom filter variants. It is also important to note that a larger number of neurons can sometimes lead to LSM being larger than a traditional Bloom filter. Intuitively, this comes from the large embeddings that scale with the number of unique elements in the dataset. In contrast, the compressed learned Bloom filter reduces the large embedding matrix and drastically reduces the input dimensions, leading to enormous space benefits when considered as a replacement for the traditional Bloom filter. This is even more visible for larger datasets with many unique elements where the size of the compressed Bloom filter is not impacted.

Table 10: Memory consumption (MB) for Bloom filter task.

Datasets	**LSM**	**CLSM**	**BF**		
			0.1	0.01	0.001
RW-200k	0.239	0.005	0.5	1.01	1.5
RW-1.5M	1.814	0.01	3.8	7.5	11.2
RW-3M	2.712	0.011	6.2	12.4	18.6
Tweets	0.577	0.006	0.9	1.8	2.7
SD	0.046	0.003	0.22	0.43	0.66

Table 11: Execution time (ms) for Bloom filter task.

Datasets	**LSM**	**CLSM**	**BF**		
			0.1	0.01	0.001
RW-200k	0.035	0.043	0.006	0.008	0.009
RW-1.5M	0.033	0.044	0.007	0.008	0.009
RW-3M	0.04	0.05	0.007	0.008	0.009
Tweets	0.032	0.045	0.007	0.008	0.01
SD	0.034	0.044	0.006	0.007	0.008



(a) MAE sum up to 10   **(b) MAE sum up to** 100

Figure 7: Accuracy of digit summation with text.

8.4.3 Execution Time. Finally, we discuss the execution time of LSM and CLSM when compared to a traditional Bloom filter. The average query result times from the evaluation over 1000 queries are presented in Table 11. As for the previous tasks, we can notice that the learned models do increase the execution time when compared to a traditional Bloom filter. However, in this case, since the task of classifying is easier, fewer neurons are sufficient, resulting in a shorter execution time when compared to the previous learned structures. When comparing LSM with CLSM, the same trend holds. Due to the initial compression and concatenation, contributing to a more complicated structure, CLSM has an increase in execution time compared to LSM.

8.5 Compression Impact & System Integration

8.5.1 Sum of Digits. To showcase the impact of compression, we perform a comparison of the native DeepSets with the compressed DeepSets using the experiment and the publicly available implementation from the original paper [24]. The experiment randomly samples a subset of maximum $M = 10$ digits to create 100000 sets, with labels representing the sum of digits in that set. The testing is performed over sums of M digits, where $M = [5, 100]$ over 10000 samples. For our compressed version we set $ns = 2$ (larger ns has similar performance). In addition, we also include LSTM and GRU as competitors. All models have the same

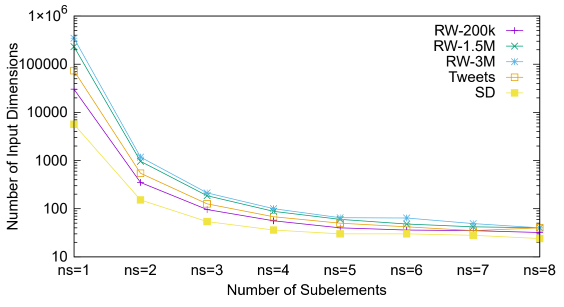


Figure 8: Impact of compression factor.

number of layers and embedding sizes and predict a scalar as the sum of the digits given as input. The results after 100 epochs are depicted in Figure 7. Based on the results depicted in Figure 7a, we can observe that both the default DeepSets model and the compressed DeepSets model (CDeepsets) generalize much better than LSTM and GRU and perform similarly for the task where we sum digits in the range [1, 10]. However, performing a compression over an embedding with 10 entries does not bring significant memory benefits. As the compression impact is not drastically visible for digits in the range [1, 10], unlike the original implementation, we vary the numbers up to 100, and we present the results in Figure 7b. The compressed version has the same MAE as the non-compressed one for a smaller memory. The memory of the compressed version is 0.035 KB, whereas the memory for the original one is 0.053 KB. Although not depicted, a similar trend can be observed in the accuracy when we increase the numbers in the range [1, 1000], with an even more significant difference in the memory, i.e., DeepSets occupies 0.404 KB, whereas the compressed version is 0.052 KB. During training, we can also notice that the compressed model learns faster and converges faster than the non-compressed DeepSets. Note that we previously mentioned that the compressed version increases the model complexity and may impact the accuracy. Although not visible in this simple use case, it was previously apparent for more complex tasks such as cardinality estimation and indexing.

8.5.2 Impact of Compression Factor ns. The compression factor *ns* directly affects the memory footprint of the model, impacting the ability for the models to be applied for various database tasks. Therefore, it is essential to investigate how much the compression factor *ns* has an impact on the input dimensions of the model. The results are depicted in Figure 8. Based on the results, it is evident that when increasing *ns* there is a drastic reduction in the input dimensions. Naturally, when increasing *ns*, we affect not only the size but also the accuracy of the models. Thus, when considering both the memory and the accuracy, we suggest setting *ns* to two or three.

8.5.3 System Integration. To demonstrate how our proposed approaches behave in a real-world system, we have implemented our cardinality estimator as a user-defined function in PostgreSQL 13. We imported RW-3M in PostgreSQL as an hstore data type that allows storing sets of key-value pairs as an attribute value inside a table. In our case, the set elements represent keys. For the evaluation, we considered 5000 queries. We depict the results of exact COUNT queries with and without an index in PostgreSQL and our cardinality estimator in Table 12. As previously claimed, CLSM drastically reduces the memory of the index in PostgreSQL. Furthermore, we notice that CLSM tremendously

Table 12: Cardinality estimator in PostgreSQL (evaluation performed over RW-3M).

	PostgreSQL w/o Index	PostgreSQL w/ Index	CLSM
Avg. Exec. Time (ms)	295.5	1.78	**1.26**
Memory (MB)	–	37.92	**0.182**
Build Time (s)	–	**29.9**	400

speeds up the performance of PostgreSQL without an index and outperforms the built-in hstore index. The model still preserves the low prediction time, but the overhead comes from the data transfer. This experiment further underpins the importance of our proposed learned approaches over sets.

8.6 Lessons Learned

As we have observed from the experiments, depending on the settings and the datasets used, every model can bring different benefits. When comparing LSM against CLSM, we notice a decrease in accuracy but an enormous memory benefit, especially in the presence of many elements. For cardinality estimation, we can use both LSM and CLSM with and without an auxiliary structure, depending on the needed trade-off between memory and accuracy. Including an auxiliary structure improves accuracy when even a small number of outliers are removed. For the indexing task, solely using the model produces enormous errors, leading to large execution times. For that reason, the hybrid option is a necessity. Furthermore, the local range errors are highly beneficial and produce a shorter execution time. Therefore, the best options are both LSM-Hybrid and CLSM-Hybrid with local range errors. Finally, for the membership task, since the drastic space saving is a requirement to compete with the traditional structure, CLSM is the best option. In conclusion, the preferred option for most scenarios is the CLSM-Hybrid variant since it always incurs negligible additional memory but achieves drastically better estimation accuracy.

9 CONCLUSION AND FUTURE WORK

We addressed the problem of learning over sets for subset membership queries, indexing, and cardinality estimation. We put forward models that use the permutation invariant architecture DeepSets for handling sets. Since sets can consist of many unique elements, we utilized a per-element lossless compression to create models of practical size without substantially lowering their accuracy. We appropriately extended the DeepSets architecture while retaining the crucial permutation invariant property. We proposed a hybrid structure incorporating a learned model and an auxiliary structure for indexing and cardinality estimation in challenging distributions. We analyzed the advantages and limitations of the proposed models and performed a comprehensive experimental evaluation that justifies our design decisions and supports our claims.

As future work, the exploration of set-related problems can be broadened by considering multi-set multi-membership querying. This would include analysis of the preprocessing steps and identification of an appropriate learned architecture.

ACKNOWLEDGEMENTS

This work has been partially funded by the German Federal Ministry of Food and Agriculture under grant number 28DE113C18 (DigiVine). The responsibility for the content of this publication lies with the authors.

REFERENCES

[1] Angjela Davitkova, Damjan Gjurovski, and Sebastian Michel. 2021. Compressing (Multidimensional) Learned Bloom Filters. In *Workshop on Databases and AI.* https://openreview.net/forum?id=0BAqBIJAegT

[2] Angjela Davitkova, Damjan Gjurovski, and Sebastian Michel. 2022. LMKG: Learned Models for Cardinality Estimation in Knowledge Graphs. In *EDBT*. OpenProceedings.org, 2:169–2:182.

[3] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT*. OpenProceedings.org, 407–410.

[4] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD Conference*. ACM, 969–984.

[5] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

[6] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (2019), 1044–1057.

[7] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[8] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *SIGMOD Conference*. ACM, 1035–1050.

[9] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

[10] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD Conference*. ACM, 489–504.

[11] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).

[12] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. 2019. Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks. In *ICML (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 3744–3753.

[13] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. *Proc. VLDB Endow.* 15, 3 (2021), 414–426.

[14] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and Ed H Chi. 2018. Lifting the curse of multidimensional data with learned existence indexes. In *Workshop on ML for Systems at NeurIPS*. 1–6.

[15] Ryan Marcus and Olga Papaemmanouil. 2019. Towards a Hands-Free Query Optimizer through Deep Learning. In *CIDR*. www.cidrdb.org.

[16] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[17] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *NeurIPS*. 462–471.

[18] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *DEEM@SIGMOD*. ACM, 4:1–4:4.

[19] Jack W. Rae, Sergey Bartunov, and Timothy P. Lillicrap. 2019. Meta-Learning Neural Bloom Filters. In *ICML (Proceedings of Machine Learning Research)*, Vol. 97. PMLR, 5271–5280.

[20] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. 2021. Partitioned Learned Bloom Filters. In *ICLR*. OpenReview.net.

[21] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. 2019. Cardinality estimation with local deep learning models. In *aiDM@SIGMOD*. ACM, 5:1–5:8.

[22] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Peter Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[23] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.

[24] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. 2017. Deep Sets. In *NIPS*. 3391–3401.