An Empirical Analysis of Deep Learning for Cardinality Estimation

Jennifer Ortiz[†], Magdalena Balazinska[†], Johannes Gehrke[‡], S. Sathiya Keerthi[+]
University of Washington[†], Microsoft[‡], Criteo Research[+]

ABSTRACT

We implement and evaluate deep learning for cardinality estimation by studying the accuracy, space and time trade-offs across several architectures. We find that simple deep learning models can learn cardinality estimations across a variety of datasets (reducing the error by 72% - 98% on average compared to PostgreSQL). In addition, we empirically evaluate the impact of injecting cardinality estimates produced by deep learning models into the PostgreSQL optimizer. In many cases, the estimates from these models lead to better query plans across all datasets, reducing the runtimes by up to 49% on select-project-join workloads. As promising as these models are, we also discuss and address some of the challenges of using them in practice.

1 INTRODUCTION

Query optimization is at the heart of relational database management systems (DBMSs). Given a SQL query, the optimizer automatically generates an efficient execution plan for that query. Even though query optimization is an old problem [34], it remains a challenging problem today: existing database management systems (DBMSs) still choose poor execution plans for many queries [25]. Cardinality estimation is the ability to estimate the number of tuples produced by a subquery. This is a key component in the query optimization process. It is especially challenging with complex queries that contain many joins, where cardinality estimation errors propagate and amplify from the leaves to the root of the query plan. One problem is that existing DBMSs make simplifying assumptions about the data (e.g., inclusion principle, uniformity or independence assumptions) when estimating the cardinality of a subquery. When these assumptions do not hold, cardinality estimation errors occur, leading to sub-optimal plan selections [25]. To accurately estimate cardinalities, optimizers must be able to capture detailed data distributions and correlations across columns. Capturing and processing this information, however, imposes space and time overheads and adds complexity.

To support cardinality estimation, DBMSs collect statistics about the data. These statistics typically take the form of histograms or samples. Because databases contain many tables with many columns, these statistics rarely capture all existing correlations. The manual process of selecting the best statistics to collect can help but requires significant expertise both in database systems and in the application domain.

Recently, thanks to dropping hardware costs and growing datasets available for training, *deep learning* has successfully been applied to solving computationally intensive learning tasks in other domains. The advantage of these type of models comes from their ability to learn unique patterns and features of the data that are difficult to manually find or design [14].

Given this success, we ask the following fundamental question: Should we consider using deep learning for query optimization? Can a deep learning model actually learn properties about the data and learn to capture correlations that exist in the data? What is the overhead of building these models? How do these models compare to other existing machine learning techniques? In this work, we implement a variety of deep learning architectures to predict query cardinalities. Instead of relying entirely on basic statistics and formulas to estimate cardinalities, we train a model to automatically learn important properties of the data to more accurately infer these estimates. In this paper, we seek to understand the fundamental capabilities of deep neural networks for this application domain. For this reason, we focus on the performance of basic neural network architectures.

Our community has recently started to consider the potential of deep learning techniques to solve database problems [41]. There still is, however, limited understanding of the potential and impact of these models for query optimization. Previous work has demonstrated the potential of using deep learning as a critical tool for learning indexes [22], improving query plans [28], and learning cardinalities specifically through deep set models [19], but we argue that that the accuracy should not be the only factor to consider when evaluating these models. We also need to consider their overheads, robustness, and impact on query plan selection. We need a systematic analysis of the benefits and limitations of various fundamental architectures.

In this experimental study, we focus on the trade-offs between the size of the model (measured by the number of trainable parameters), the time it takes to train the model, and the accuracy of the predictions. We study these trade-offs for several datasets. Our goal is to understand the overheads of these models compared to PostgreSQL's optimizer. To do this, we build several simple neural network as well as recurrent neural network models and vary the complexity by modifying the network widths and depths. We train each model separately and compare the overheads of these models to PostgreSQL and random forest models (based on an off-the-shelf machine learning model).

To summarize, we contribute the following:

- We show how deep neural networks and recurrent neural networks can be applied to the problem of cardinality estimation and describe this process in Section 3.
- We comparatively evaluate neural networks, recurrent neural networks, random forests, and PostgreSQL's optimizer on three real-world datasets in Section 4. For a known query workload, we find that, compared to PostgreSQL, simple deep learning models that are similar in space improve cardinality predictions by reducing the error by up to 98%. These models, however, come with high training overheads. We also find

that, although random forest models usually require a larger amount of space, they are fast to train and are more accurate than the deep learning models.
- In Section 4.3, we study these models in more detail by evaluating the robustness of these models with respect to query workload changes. We find that random forest models are more sensitive to these changes compared to the neural network and recurrent neural network models. Although, decision tree models are known to have high variance and random forest models should reduce this problem [16], we still find that the neural networks are generally more robust to changes in the data.
- In Section 4.4, we visualize the embeddings from the models to understand what they are learning.
- Finally, we study these models from a practical perspective in Section 5, where we evaluate how predictions from these models improve query plan selection. We find that these models can help the optimizer select query plans that lead from 7% to 49% faster query executions. In addition, we study how *active learning* can help reduce the training overheads.

2 BACKGROUND AND PROBLEM STATEMENT

Many optimizers today use histograms to estimate cardinalities. These structures can efficiently summarize the frequency distribution of one or more attributes. For single dimensions, histograms split the data using equal-sized buckets (equi-width) or buckets with equal frequencies (equi-depth). To minimize errors, statistics about each bucket are also stored including but not limited to the number of items, average value, and mode [9].

These histograms are especially relevant in cases where there are simple single query predicates. For more complex predicates, the system extracts information from these histograms in conjunction with "magic constants" to make predictions [25]. Optimizers typically do not build or use multidimensional histograms or sampling due to the increased overheads [12, 44]. As the estimates from these optimizers are not theoretically grounded, propagating these estimates through each intermediate result of a query plan can result in high cardinality errors, leading to sub-optimal query plans.

In this paper, we use PostgreSQL's optimizer as representative of this class because it is a mature optimizer available in a popular open source system.

Our goal in this paper is to apply deep learning to the cardinality estimation problem and compare the performance of this approach empirically to that of a traditional query optimizer.

We consider the following scenario: A database system is deployed at a customer's site. The customer has a database D and a query workload Q. Both are known. We compare the following approaches:

- **Traditional:** In the pre-processing phase, we build histograms on select attributes in D. We select those attributes following standard best practices given the workload Q. Simple best practices include collecting statistics for all frequently joined columns and for non-indexed columns frequently referenced as a selection predicate, particularly if the column contains very skewed data[38]. We then measure the accuracy of cardinality estimates on queries in Q (and queries similar to Q) and the overhead of creating and storing the histograms. We measure both the time it takes to build the histograms and the space that the histograms take.
- **Deep Neural Networks:** In the pre-processing phase, we execute all queries in Q to compute their exact cardinalities. We use the results to train deep neural networks. We encode all queries in Q into inputs for the models, and evaluate how accurately these models are able to learn the function between the input, X and the cardinality value, Y. As above, we measure the overhead of building and storing the model and the accuracy of cardinality estimates for queries in Q and queries not in Q but similar to those in Q. To compare different architectures, we build several models by varying the width and depth.

As a simplifying assumption, in this paper, we focus on select-project-join queries and only use single-sided range predicates as selection predicates. The join predicates consist of primary key and foreign key relationships between tables, as defined by their schema.

3 MACHINE LEARNING-BASED CARDINALITY ESTIMATION

The first contribution of this paper is to articulate how to map the cardinality estimation problem into a learning problem. We show the mapping for three types of models: neural networks, recurrent neural networks, and random forests.

For ease of illustration, in this section, we use a simple running example comprising a database D with three relations, $D : \{A, B, C\}$. Each relation has two attributes where relation A contains $\{a_1, a_2\}$, relation B has attributes $\{b_1, b_2\}$, and relation C has attributes $\{c_1, c_2\}$. In this database, there are two possible join predicates. Attribute a_2 is a foreign key to primary key attribute b_1, while b_2 serves as a foreign key to primary key attribute c_1.

3.1 Neural Networks

Deep learning models are able to approximate a non-linear function, f [14]. These models define a mapping from an input X to an output Y, through a set of learned parameters across several layers with weights, θ. Each layer contains a collection of neurons, which help express non-linearity. During training, the behavior of the inner layers are not defined by the input data X, instead these models must learn how to tune the weights to produce the correct output. Since there is no direct interaction between the layers and the input training data, these layers are called *hidden layers* [14].

Training occurs through a series of steps. First, during forward propagation, a fixed-sized input X is fed into the network through the input layer. This input is propagated through each layer through a series of weights [37] until it reaches the final output, Y. This process is illustrated in Figure 1. After a forward pass, the back-propagation step then evaluates the error of the network and through gradient descent, modifies the weights to improve errors for future predictions.

There are several architectures we could consider for the model. As shown in Figure 1, a neural network can have a different number of layers (depth) and a different number of hidden units in each individual layer (width). Determining the correct number of hidden

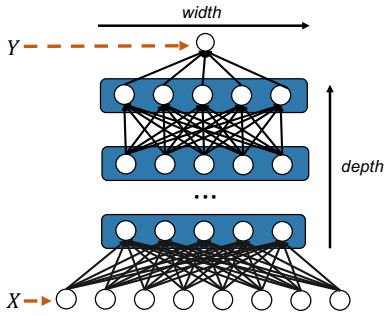


Figure 1: Illustration of a Deep Neural Network: The input consists of an input vector X. This is then fed into a network with n hidden layers, which then makes a prediction for the cardinality of the query, Y.

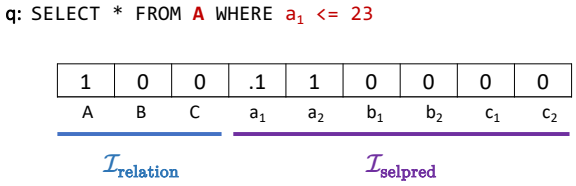


Figure 2: Query Encoding for selections: We encode a selection query by specifying the underlying relations and all selection predicate values.

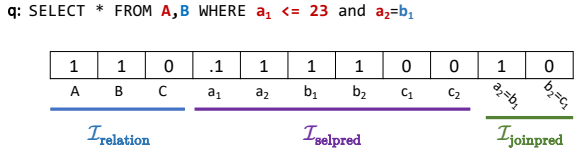


Figure 3: Query Encoding for Joins+Selections: We encode a join+selection query based on the joined relations, the selections predicate values and join predicates.

units is currently an active area of research and does not have strong theoretical principles [14]. Although a network with only a single wide hidden layer is capable of learning a complex function, deep networks are able to use a smaller number of training parameters to achieve the same goal. Unfortunately, deep networks are difficult to train and to optimize [14]. In this work, we focus on evaluating a variety of network architectures. We focus on simple architectures comprising a small number of fully connected layers. We vary the width and the depth of the network. More complex architectures are possible [19] and are also interesting to study. Our goal, however, is to understand the performance of basic architectures first.

Given a model, a query q and a fixed dataset D, we define an encoding for the input, X. The input X should contain enough information for the model to learn a useful mapping. There are several ways to represent a query as an input vector. The encoding determines how much information we provide the network. In this work, we define X as a concatenation of three single dimensional vectors: $\mathcal{I}_{relations}$, $\mathcal{I}_{selpred}$, and $\mathcal{I}_{joinpred}$. To explain this encoding, we first describe how to encode selection queries.

Modeling Selection Queries

With selections queries, the goal is to have the network learn the distribution of each column and combinations of columns for a single relation. To encode a selection query, we provide the model with information about *which* relation in D we are applying the selections to, along with the attribute values used in the selection predicates. We encode the relation using vector $\mathcal{I}_{relations}$, and a binary one-hot encoding. Each element in $\mathcal{I}_{relations}$ represents a relation in D. If a relation is referenced in q, we set the designated element to 1, otherwise we set it to 0.

We encode the selection predicates in q using vector $\mathcal{I}_{selpred}$. As described in Section 2, selection predicates are limited to single-sided range predicates. Each element in this vector holds the selection value for one attribute. The vector includes one element for each attribute of each relation in D. As an example, assume we have the following query: SELECT * FROM A WHERE $a_1 \leq 23$. In this case, we set the corresponding element for a_1 in $\mathcal{I}_{selpred}$ as 23. Otherwise, if there is no selection on an attribute, we set the element with the maximum value of the attribute's domain. This captures the fact that we are selecting all values for that attribute.

Neural networks are highly sensitive to the domain of the input values. Having unnormalized values in the input will highly impact the error surface for gradient descent, making the model difficult to train. Instead, we encode these selection predicates as values ranging from 0 to 1, where the value represents the percentile of the attribute's active domain as shown in Figure 2. The output of the model is also normalized to represent the percentage of tuples that remain after the selection query is applied to the relation. Using this type of normalization, instead of learning query cardinalities, we are learning in fact predicate selectivities.

Modeling Join Queries Introducing queries that contain both joins and selections requires the model to learn a more complex operation. Joins essentially apply a cartesian product across a set of relations followed by additional filters that correspond to the equality join and selection predicates. As we later show in our measurement analysis, deeper networks are usually more successful at predicting join cardinalities than shallower networks. We encode existing *join predicates* with the vector $\mathcal{I}_{joinpred}$ using a binary one-hot encoding. As we now include joins, the output Y now represents the fraction of tuples selected from the join result. Hence, once again, the model will learn the selectivity of the join operation.

We illustrate this encoding using an example in Figure 3. Given the following query from our running example dataset, SELECT $*$ FROM A, B WHERE $a_1 \leq 23$ and $a_2 = b_1$, we show the encoding in the figure. For $\mathcal{I}_{relations}$, there are three possible elements, one for each relation in D. For this query, we only set the elements corresponding to relations A and B to 1. The vector $\mathcal{I}_{selpred}$ contains the encoding for the selection predicates. Since relation C is not referenced in q, we set all of its attributes in $\mathcal{I}_{selpred}$ to 0. We set to .1 the element corresponding to attribute a_1, as it represents the percentile of the active domain for a_1. The rest of the attributes from A and B are set to 1, as we are not filtering any values from these attributes. Finally, the vector $\mathcal{I}_{joinpred}$ encodes the join predicate $a_2 = b_1$ with a 1.

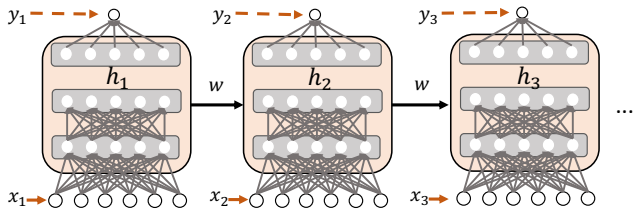


Figure 4: Illustration of a Recurrent Neural Network: The input consists of a sequence of inputs $\{x_1, x_2, ..., x_t\}$. Each input, along with the hidden state of the previous timestep, is fed into the network to make a prediction, y_i.

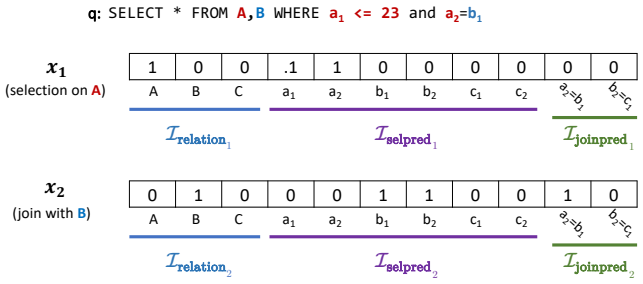


Figure 5: Query Encoding for the RNN model: In this example, the input consists of two inputs: $\{x_1, x_2\}$. The first input represents the subquery that scans and filters relation A. The second represents the join with relation B.

3.2 Recurrent Neural Networks

As we describe in our earlier short paper [29], if we focus on left-deep plans, we can model queries as sequence of operations, and we can leverage that structure when learning a model. Recurrent neural networks (RNN) in particular are designed for sequential data such as time-series data or text sequences [37]. Compared to neural networks where the input is a single vector X, the input to RNNs is a sequence with t timesteps, $X = \{x_1, x_2, ..., x_t\}$. For each timestep t, the model receives two inputs: x_t and h_{t-1}, where h_{t-1} is the generated hidden state from the previous timestep [37]. With these inputs, the model generates a hidden state for the current timestep t, where $h_t = f(h_{t-1}, x_t)$ and f represents an activation function. Given this feedback loop, each hidden state contains traces of not only the previous timestep, but all those preceding it as well. RNNs can either have a single output Y for the final timestep (a many-to-one architecture) or they can have one output for each timestep (many-to-many) where $Y = \{y_1, y_2, ..., y_t\}$.

In our context, we model queries as a series of actions, where each action represents a query operation (i.e. a selection or a join) in a left-deep query plan corresponding to the query. With a sequential input, RNNs incrementally generate succinct representations for each timestep, which are known as *hidden states*, and which represent a subquery. Recurrent neural networks rely on these hidden states to infer context from previous timesteps. More importantly, h_t is not a manually specified feature vector, but it is the latent representation that the model learns itself. We illustrate how these hidden states are generated in Figure 4. Information from each hidden state, h_t is fed into the next timestep, $t + 1$ through shared weights, w. In our context, hidden representations are useful, as they capture important details about the underlying intermediate result. The information learned at the hidden state is highly dependent on the input and output of the network.

To generate the input for this model, we concatenate three input vectors for *each* action x_i. That is, for each action x_i, we concatenate vectors $\mathcal{I}_{relation_i}$, $\mathcal{I}_{selpred_i}$ and $\mathcal{I}_{joinpred_i}$. In Figure 5, we show the representation for our running example, SELECT $*$ FROM A, B WHERE $a_1 \leq 23$ and $a_2 = b_1$. We break down this query into two operations: the scan and selection on relation A, followed by a join with relation B. Alternatively, we could have the first action represent the scan on relation B (with no selections applied), followed by a join and selection with relation A.

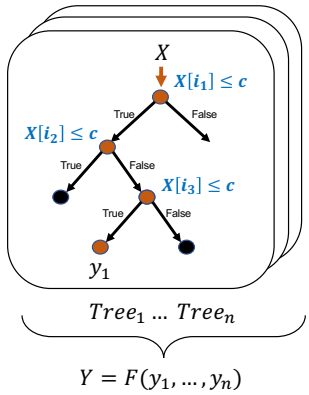


Figure 6: Illustration of a Random Forest Model: Each input X is tested against each criteria in each node (where each $X[i]$ represents an attribute in X). The predictions from each tree are aggregated for the final prediction, Y.

3.3 Random Forest

In this study, we also include random forest models, as they are fast to build and are well suited for regression and classification problems [10]. A random forest model is a combination of predictions from several independently trained trees as shown in Figure 6. Each tree, $Tree_i$, is a sequence of decisions that leads to a prediction at the leaf nodes, y_i. With each tree, the data is repeatedly split based on different attributes from X. Each tree partitions the input space into subregions, where each of these subregions either contains a linear model [32] or a constant to make a prediction [3]. Finding these subregions requires finding an optimal set of splits, which is computationally infeasible. Instead, these models use a greedy optimization to incrementally grow the trees one node at a time. To help generalize the model and to help add randomness, the random forest model uses a bootstrapped dataset from the training data to build many trees [11]. Based on the predictions from all these trees, the random forest model uses a function F (usually the mean) to compute a final prediction Y.

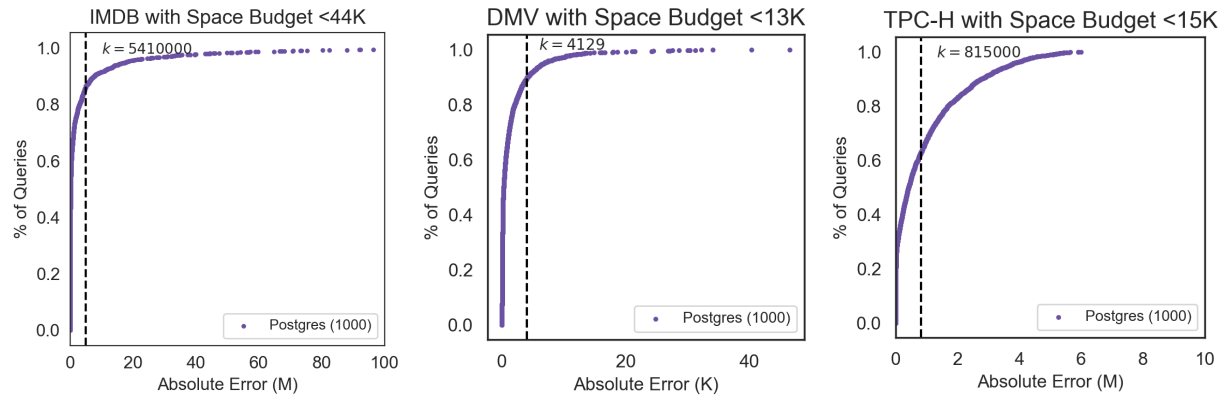


Figure 7: CDF of PostgreSQL absolute errors with storage budget: For each curve, we show the knee, k, which defines the split between Easy(PostgreSQL) and Hard(PostgreSQL).

4 MEASUREMENT ANALYSIS

In this section, we evaluate all models and their architecture variants on three datasets. We start with a description of the experimental setup, which includes the implementation of the models and generation of the training datasets. We then evaluate the accuracy, time, space trade-offs in Section 4.2, followed by a study of the robustness of the models in Section 4.3 and a look into their latents in Section 4.4.

4.1 Experimental Setup

Datasets: We evaluate the models on three datasets:

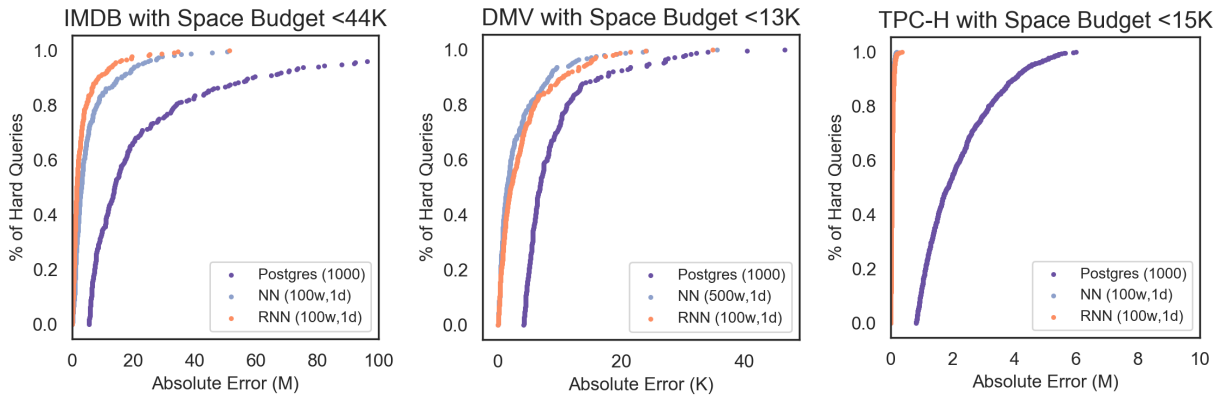
- **IMDB**: The *Internet Movie Data Base* is a real dataset that contains a wide variety of information about actors, movies, companies, etc. This dataset has 21 relations. The dataset is based on the 3.6GB snapshot from Leis et. al. [24].
- **DMV**: This dataset contains 6 relations (61MB) and is based on a real-world dataset from a Department of Motor Vehicles [18]. Relations include accidents, owners, cars, location, demographics and time.
- **TPC-H (skewed)** : This is a standard benchmark dataset with 8 relations and a scale factor of 1 (1GB). We adjust the skew factor to $z = 1$ [8].

Model Architectures: For the recurrent and neural networks, we build several models that vary in width (w) and depth (d). To minimize the number of possible architecture combinations, we assume that all layers within a model have the same width. We annotate the models as a pair (x, y), where x represents the width and y represents the depth. For example, (100w, 1d) represents a model with 100 hidden units in a single hidden layer. For the random forest models, we vary the number of trees from 1 to 500. No additional pruning takes place. We compare these models to estimates from PostgreSQL version 9.6 [31]. To fairly compare PostgreSQL to these models in terms of space, we modify the PostgreSQL source to allow for a larger number of bins in each histogram. For each relation in each dataset, we collect statistics from each join predicate column and each selection column. We vary the number of bins from the default size (100 bins) up to 100K.

Training Data: For each dataset, we generate various training sets with different levels of query complexity. We define three complexity levels: $2Join$, $4Join$ and $6Join$. $2Join$ is the case where we generate a training set with joins that consist of any 2 relations in the dataset, $4Join$ represents joins with 4 relations, and $6Join$ represents joins with 6 relations. In addition, for each dataset, we manually select a set of columns as candidates for selection predicates. We select columns with small discrete domain sizes, as these are generally the columns that contain more semantically meaningful information about the data, unlike columns that contain a sequence of identifiers. As we generate the workload, selection predicate values are randomly drawn from the domains of the selected candidate columns. We generate 100K training samples for each query complexity and each dataset. We randomly select the desired number of tables and pick the selection columns from the joined relations. For the RNN, because it requires an input for each timestep, we extend these training sets by adding more training samples for all the subqueries. For example, for a query that joins six relations, we extend the training set with additional examples representing the subquery after each intermediate join. For each query complexity training set, we select 1K samples to serve as the test set.

Hyperparameter Tuning: We tune each model architecture for each dataset. We separate 10% of the training data as the validation data. We run a basic grid search over the learning rate and batch size. A larger batch size (although faster to train, especially on a GPU) might lead to sub-optimal results, while a small batch size is more susceptible to noise. Larger learning rates also have the tendency to oscillate around the optimum, while smaller learning rates might take a long time to train. We set the number of epochs to 500 for all learning rate and batch parameter combinations. Based on the combination that leads to the lowest learning rate, we continue to train for more epochs as long as the validation loss keeps decreasing. We stop the training once the validation loss plateaus or increases.

Model Implementation Details: The neural network is implemented in Tensorflow [1] and is implemented as a residual network with leaky RELU activation functions, as it is a default recommendation to use in modern neural networks [14]. Weights are initialized from a random normal distribution with a small standard deviation.



(a) IMDB with Storage Budget

CDF Percentiles			
	25%	50%	75%
PostgreSQL	7.8M	13.8M	29.7M
NN (100w,1d)	1.25M	2.97M	6.6M
RNN (100w,1d)	.71M	1.49M	3.67M

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	6.8M	12.8M	31.3M
NN (100w,1d)	.80M	4.1M	7.0M
RNN (100w,1d)	.58M	2.2M	4.1M

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.39	.75	.95
NN (100w,1d)	.04	.22	.20
RNN (100w,1d)	.03	.11	.13

(b) DMV with Storage Budget

CDF Percentiles			
	25%	50%	75%
PostgreSQL	5.2K	6.8K	10.3K
NN (500w,1d)	.50K	1.4K	4.1K
RNN (100w,1d)	.76K	2.1K	5.1K

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	9.4K	9.4K	8.9K
NN (100w,1d)	4.9K	3.0K	3.1K
RNN (100w,1d)	7.9K	2.9K	4.0K

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.10	.20	.23
NN (100w,1d)	.03	.03	.04
RNN (100w,1d)	.06	.02	.07

(c) TPCH with Storage Budget

CDF Percentiles			
	25%	50%	75%
PostgreSQL	1.2M	1.8M	2.9M
NN (100w,1d)	.01M	.02M	.05M
RNN (100w,1d)	.01M	.03M	.06M

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	2.9M	2.2M	1.8M
NN (100w,1d)	35K	40K	32K
RNN (100w,1d)	27K	60K	41K

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.99	.99	.99
NN (100w,1d)	.01	.02	.01
RNN (100w,1d)	.01	.03	.02

Figure 8: Error Analysis for all Models : We show the curve for Hard(PostgreSQL) and show the corresponding errors from the best models below the storage budget. Below each graph, we show tables detailing the percentiles, the average absolute error and average relative error.

	% Queries Easy (Models)					
	IMDB		DMV		TPC-H	
	NN(100w,1d)	RNN(100w,1d)	NN(500w,1d)	RNN(100w,1d)	NN(100w,1d)	RNN(100w,1d
Easy(PostgreSQL)	99.5%	99.8%	90.5%	94.5%	100%	100%
Hard(PostgreSQL)	71.4%	83.5%	75.6%	69.4%	100%	100%

Table 1: Percentage of Queries that are Easy for the Models: For each Easy(PostgreSQL) query batch, we find the percentage of queries that are also easy for the models. We also show the percentage of queries that are easy based on the Hard(PostgreSQL) batch

Biases are initialized to .01. The input X is normalized as explained in Section 3.1 and centered using a StandardScaler. The output Y is log transformed and also normalized with a StandardScaler. The model's goal is to minimize the mean squared error between the real outputs and the predictions. We use the AdamOptimizer as the optimizer for the model. The recurrent neural network is also implemented in Tensorflow. For deep recurrent neural networks, we use a ResidualWrapper around each layer, to mimic the residual implementation of the neural networks. Both the neural network and recurrent neural networks are run on a GPU on p2.xlarge instances on Amazon AWS [2]. Finally, the Random Forest model is based on an implementation from sklearn's RandomForestRegressor module [30].

4.2 Learning Cardinalities for Selections + Joins

In this section, we vary the architecture of the models and evaluate them on the three datasets. We study the trade-offs (space, time, and accuracy) for these models.

First, we evaluate the prediction accuracy for each model. As described in Section 2, we make the assumption that the query workload is known in advance (we relax this assumption later in this section). In this case, the models overfit to a specific set of queries.

For each query complexity, we train six neural network (NN) and six recurrent neural network models (RNN) based on the following widths and depths: (100w, 1d), (100w, 5d), (500w, 1d), (500w, 5d), (1000w, 1d), (1000w, 5d). We separately train four random forest models (RF) with 1, 5, 50 or 500 trees. Larger models generally use up more space, but result in more accurate cardinality predictions.

To make this analysis comparable to PostgreSQL, we first limit the storage budget for the models to be no more than the storage budget for PostgreSQL histograms. We compute the size of a model as the size of all its parameters. For the NN and RNN models, we thus measure the number of trainable variables and for PostgreSQL, we measure the number of parameters used in the *pg_stats* table. We compare PostgreSQL cardinality estimates to those produced by models that are smaller in size compared to PostgreSQL's histograms. We specifically study the PostgreSQL scenario where each histograms builds at most 1K bins. Setting the number of bins to 1K for PostgreSQL results in 13385 parameters for the DMV dataset, 15182 parameters for the TPC-H dataset and 44728 parameters for the IMDB dataset. We purposely set PostgreSQL as the storage upper bound size. Given these storage budgets, we then select the best neural network architecture, the best recurrent network architecture, and the best random forest model. If no model meets the budget, we do not display them on the graphs. If more than one model architecture meets the storage budget, we display the best model, where the best model is defined as the one with the lowest median error.

Limited Storage CDFs and Outlier Analysis: For PostgreSQL, as for other relational DBMSs, cardinality estimation is easy for some queries and hard for others. As expected PostgreSQL yields more accurate predictions for queries with a low complexity, particularly those with no selection predicates. To help distinguish between these "easy" and "hard" queries (labeled as Easy(PostgreSQL) and Hard(PostgreSQL)), we plot the absolute errors from PostgreSQL as a cumulative distribution (cdf) as shown in Figure 7. We use the knee (k) of the cdf curve to split the queries into an "easy" category (those with errors less than the knee, k) and a "hard" category (those with errors greater than the knee k). For the TPC-H dataset, the distribution of errors is wide. To ensure we retain enough queries in the Hard(PostgreSQL) category (for later more in-depth analysis), we compute k and half the corresponding error.

We first focus on the Hard(PostgreSQL) queries. These are the more interesting queries to study as these are the queries for which we seek to improve cardinality estimates. We plot the distribution of errors for the best performing models for each dataset in Figure 8. Overall, both types of models outperform PostgreSQL on all three datasets. We also find the performance of both types of models to be similar.

First, in Figure 8a, we show the cdf for the Hard(PostgreSQL) queries from the IMDB dataset. From the entire set of IMDB queries, 11% of the queries fall in the Hard(PostgreSQL) category. The y-axis represents the percentage of queries and the x-axis represents the absolute error. In addition to the PostgreSQL error curve, we show the cdf for the corresponding queries from the best neural network and recurrent neural network models. We do not show the random forest models here, as the smallest model does not meet the storage budget. Both the neural network and recurrent neural network have comparable cardinality estimation errors. On average, the neural network reduces estimation error by 72%, while the recurrent neural network reduces the error by 80%. Below Figure 8a, we include additional details that show the percentiles of the model cdfs, the average absolute error for each query complexity, and the average relative error.

In Figure 8b, we show the cdf for Hard(PostgreSQL) from the DMV dataset. Approximately 10% of the queries are labeled as hard for PostgreSQL. The NN reduces the errors by 75% on average and the RNN by 73%. As shown in the tables below the figure, the complexity of the queries does not heavily impact the average error. In fact, the relative errors for the NN across all query complexities have a small standard deviation (σ = .004), compared to IMDB (σ = .08). Compared to DMV, the IMDB dataset contains several many-to-many primary/foreign key relationships, so joining relations significantly increases the size of the final join result.

We also observe a significant error reduction in Figure 8c (TPC-H), where the NN improves estimates by 98% and the RNN by 97%. For TPC-H, 30% of the queries are hard for PostgreSQL.

Table 1 shows the percentage of queries that are easy for the models given that they are either Easy(PostgreSQL) or Hard(PostgreSQL) for PostgreSQL. In the case of Hard(PostgreSQL) queries, 70% or more become easy with the models. For the Easy(PostgreSQL) queries, the simple NN and RNN models also find a majority of these queries to be easy (>90%). For IMDB and DMV, there are some queries from the Easy(PostgreSQL) batch that the models find to be hard. We highlight some of these hard queries below:

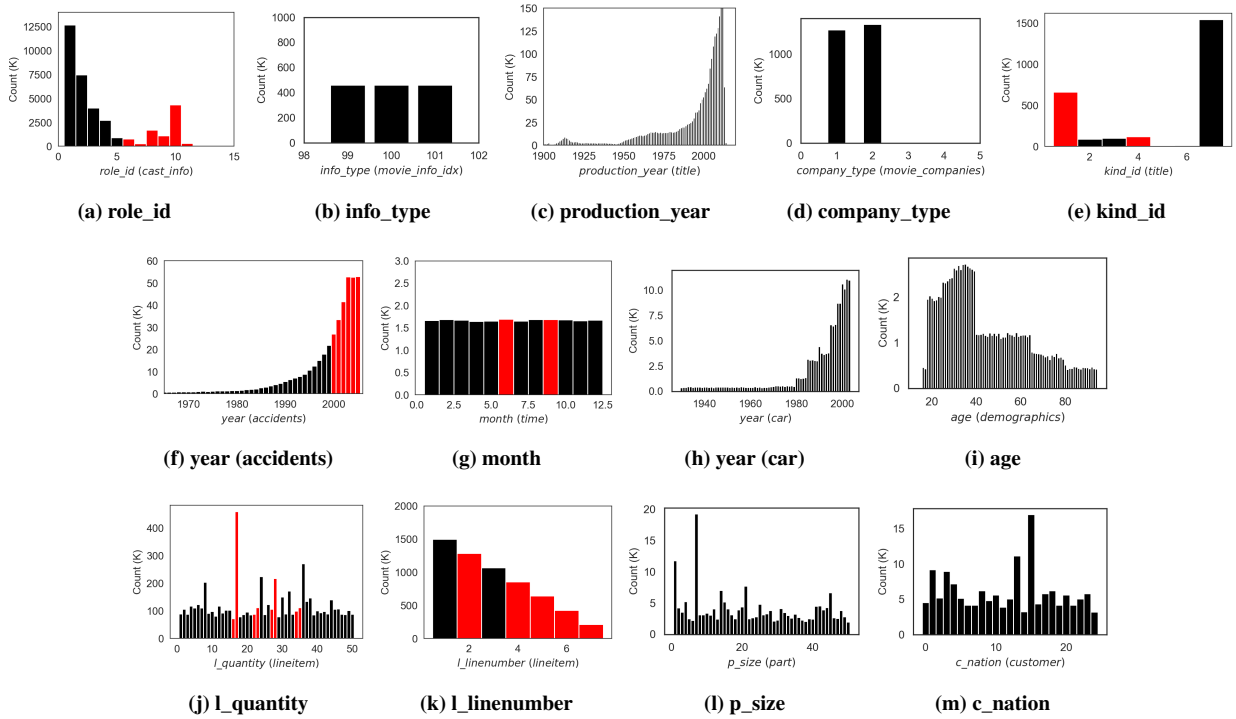
- From the IMDB dataset, approximately 0.4% of the Easy(PostgreSQL) queries are hard for the NN, and we find that the query with the highest error is one with an absolute error of 8.9M. This query joins the *name*, *cast_info*, *role_type*, and *char_name* relations and has a selection predicate on *role_id* <= 8. For the RNN, the query with the highest error is similar. It joins the *name*, *cast_info*, *role_type*, and *char_name* relations, with a selection predicate on *role_id* <= 4.
- For the DMV dataset, approximately 10.5% of the queries are hard for NN, and 6.5% are hard for the RNN. The query with the highest error for the NN is one that joins all relations *car*, *demographics*, *location*, *time*, *owner*, and *accidents* and has several selection predicates: *age_demographics* <= 89, *month_time* <= 12, *year_accidents* <= 2004. For the RNN, the query with the highest absolute error also joins all relations and has selection predicates with similar values, *age_demographics* <= 93,*month_time* <= 9,*year_accidents* <= 2005.

From the Hard(PostgreSQL) queries, there are more queries that remain difficult for the models compared to Easy(PostgreSQL). These hard queries consist of joins of 6 relations (the most complex queries we have in the test set) and up to 5 selection predicates.

Understanding why the NN or RNN fail to accurately predict the cardinality for specific queries is challenging as there are several factors to consider. For example, the error could be caused by a specific join or perhaps a combination of selection attributes. To gain a better understanding of these errors, we now only focus on the queries with a low complexity (i.e. those from the 2*Join* test set). In Table 2, we take the Easy(PostgreSQL) queries and show the queries

	Queries with Highest Errors from $2Join$	
	Best NN per Dataset	**Best RNN per Dataset**
IMDB	(cast_info,role_type) where role_id<= 11 [1.9M] (cast_info,role_type) where role_id <= 10 [1.9M] (cast_info,role_type) where role_id <= 8 [1.9M] (cast_info,title) where kind_id <= 1,production_year <=\ 2019,role_id<= 4 [1.5M] (movie_info,info_type) [1.3M] (cast_info,role_type) where role_id <= 7 [1.2M] (cast_info,role_type) where role_id <= 6 [1.1M] (cast_info,title) where kind_id <= 4,production_year <=\ 2019,role_id<= 6 [1.0M]	(cast_info,name) where role_id <= 9 [2.1M] (cast_info,name) where role_id<= 11 [2.1M] (cast_info,role_type) where role_id <= 11 [1.9M] (cast_info,role_type) where role_id <= 9 [1.8M] (cast_info,role_type) where role_id <= 7 [1.3M] (cast_info,name) where role_id <= 8 [1.2M] (cast_info,role_type) where role_id <= 7 [1.1M] (cast_info,title) where kind_id <= 4,production_year <=\ 2019,role_id<= 6 [.9M]
DMV	(accidents,time) where year<= 2005 and month <= 9 [11K] (accidents,time) where month <= 9 and year <= 2005 [11K] (accidents,time) where year <= 2003 and month <= 6 [10K] (accidents,time) where year <= 2000 and month <= 6 [10K] (accidents,location) where year <= 2001 [7K] (accidents,location) where year <= 2000 [7K] (car,accidents) where year <= 2005 [6K] (car,accidents) where year <= 2004 [6K]	(accidents,time) where year <= 2005 and month <= 9 [33K] (accidents,location) where year <= 2003 [18K] (car,accidents) where year <= 2005 [18K] (car,accidents) where year <= 2003 [18K] (accidents,time) where year <= 2005 and month <= 9 [17K] (accidents,time) where year <= 2003 and month <= 6 [10K] (accidents,time) where year <= 2000 and month <= 6 [10K] (accidents,location) where year <= 2002 [8K]
TPC-H	(lineitem,orders) l_linenumber <= 7 and l_quantity <=16 [116K] (lineitem,orders) l_linenumber <= 6 and l_quantity<= 16 [109K] (lineitem,orders) l_linenumber <= 5 and l_quantity <= 35 [98K] (lineitem,orders) l_linenumber <= 7 and l_quantity <= 34 [84K] (lineitem,orders) l_linenumber <= 7 and l_quantity <= 27 [65K] (lineitem,orders) l_linenumber <= 2 and l_quantity <= 17 [61K] (lineitem,orders) l_linenumber <= 6 and l_quantity <= 27 [61K] (lineitem,orders) l_linenumber <= 7 and l_quantity <= 23 [59K]	(lineitem,orders) where l_linenumber <= 7 and l_quantity <= 34 [89K] (lineitem,orders) where l_linenumber<= 6 and l_quantity <= 38 [75K] (lineitem,orders) where l_linenumber <= 5 and l_quantity <= 35 [72K] (lineitem,orders) where l_linenumber <= 6 and l_quantity <= 28 [62K] (lineitem,orders) where l_linenumber <= 7 and l_quantity <= 22 [55K] (lineitem,orders) where l_linenumber <= 5 and l_quantity <= 28 [52K] (lineitem,orders) where l_linenumber <= 4 and l_quantity <= 28 [51K] (lineitem,orders) where l_linenumber <= 7 and l_quantity <= 23 [50K]

Table 2: The $2Join$ Queries with the Highest Errors from the NN and RNN Models: For each dataset, we show the top eight queries with the highest absolute errors.

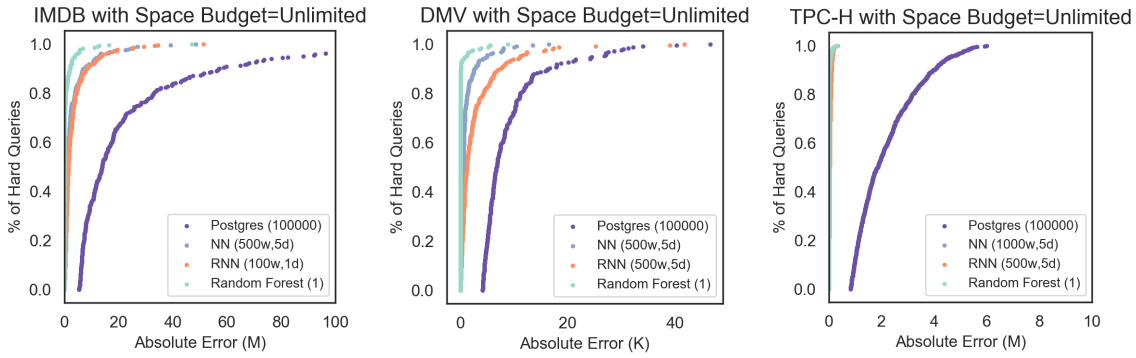


(a) role_id **(b) info_type** **(c) production_year** **(d) company_type** **(e) kind_id**

(f) year (accidents) **(g) month** **(h) year (car)** **(i) age**

(j) l_quantity **(k) l_linenumber** **(l) p_size** **(m) c_nation**

Figure 9: Distributions for all Selection Columns: First row shows all distributions from the IMDB relation. Second shows distributions from DMV, and the third shows TPC-H.



(a) IMDB Unlimited

CDF Percentiles			
	25%	50%	75%
PostgreSQL	7.8M	13.9M	27.5M
NN (500w,5d)	.30M	.97M	2.6M
RNN (100w,1d)	.6M	1.4M	3.6M
Random Forest (1)	1e-6M	6e-6M	.2M

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	6.8M	12.8M	31.3M
NN (500w,5d)	.28M	.45M	4.1M
RNN (100w,1d)	.58M	2.0M	4.1M
Random Forest (1)	1e-6M	.21M	1.0M

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.38	.80	.95
NN (500w,5d)	.01	.03	.11
RNN (100w,1d)	.03	.11	.13
Random Forest (1)	3e-8	.01	.03

(b) DMV Unlimited

CDF Percentiles			
	25%	50%	75%
PostgreSQL	5.28K	6.82K	10.3K
NN (500w,5d)	.16K	.46K	1.0K
RNN (500w,5d)	.02K	1.1K	3.2K
Random Forest (1)	3e-6K	6e-6K	1e-5K

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	9.5K	9.3K	8.9K
NN (500w,5d)	.4K	.9K	1.3K
RNN (500w,5d)	1.1K	3.3K	2.8K
Random Forest (1)	8e-5	.06K	.29K

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.10	.20	.23
NN (500w,5d)	.002	.01	.01
RNN (500w,5d)	.006	.03	.04
Random Forest (1)	7e-8	.0008	.007

(c) TPCH Unlimited

CDF Percentiles			
	25%	50%	75%
PostgreSQL	1.2M	1.8M	2.9M
NN (1000w,5d)	.01M	.02M	.04M
RNN (500w,5d)	.01M	.02M	.05M
Random Forest (1)	4e-8M	8e-8M	1e-7M

Average Absolute Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	2.9M	2.2M	1.8M
NN (1000w,5d)	16K	41K	28K
RNN (500w,5d)	29K	59K	23K
Random Forest (1)	9e-4	3K	15K

Average Relative Errors			
	2Join	*4Join*	*6Join*
PostgreSQL	.99	.99	.99
NN (1000w,5d)	.007	.02	.01
RNN (500w,5d)	.01	.02	.01
Random Forest (1)	4e-8	.001	.009

Figure 10: Error Analysis for all Models : We show the curve for Hard(PostgreSQL) and show the corresponding errors from the best models with an unlimited storage budget. Below each graph, we show tables detailing the percentiles, the average absolute error and average relative error.

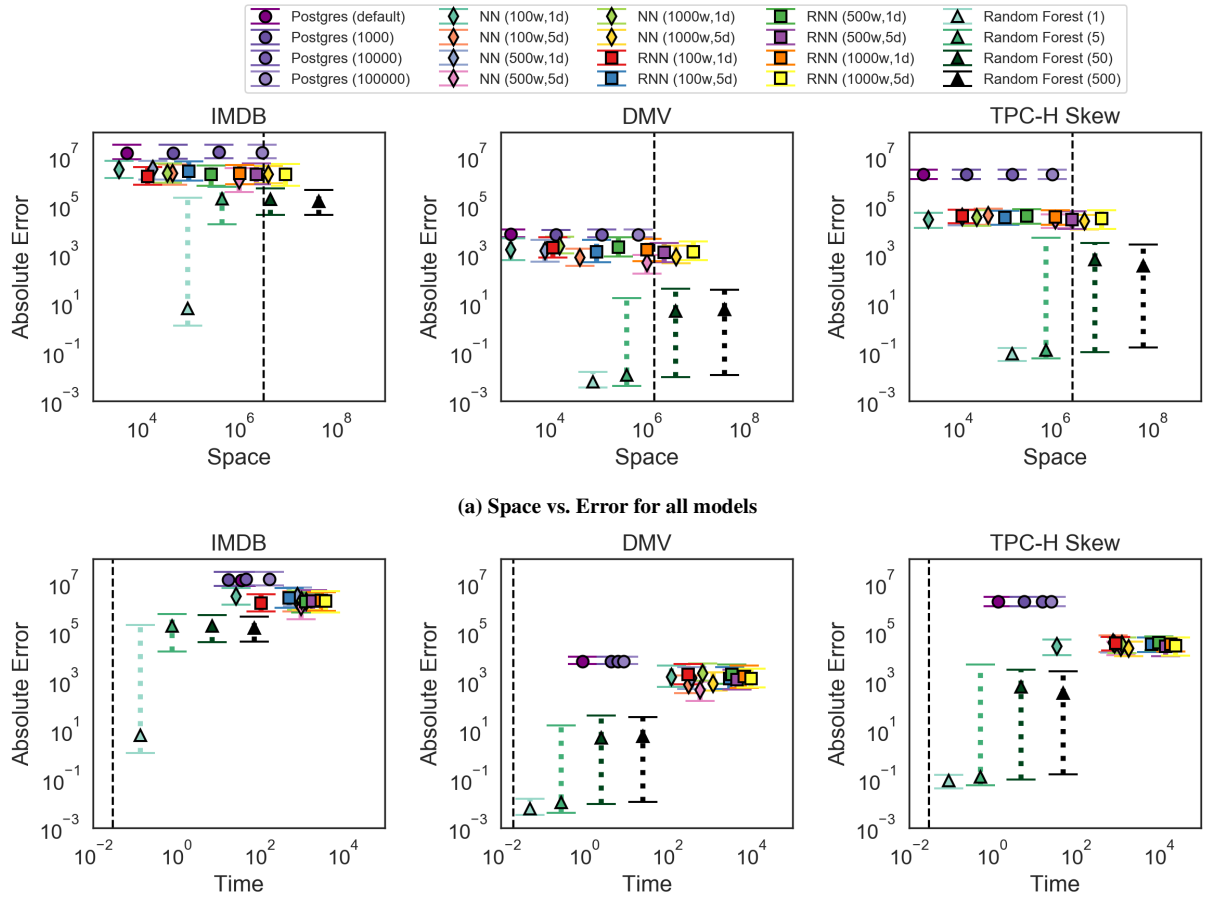
with the highest errors from the $2Join$ set. For succinctness, we annotate each query with the names of the relations it joins (relations are listed in parenthesis) and its selection predicates. We further add the absolute error of the query in brackets.

For IMDB, the hardest queries for the NN and RNN are similar. These queries consist of joins with $cast\_info$ and either $role\_type$ or $name$. All queries also have a selection predicate on the $role\_id$ column with values between 6 and 11. Figure 9 shows the value distributions for different attributes. The first row shows all selection columns for IMDB, the second for DMV, and third for TPC-H. The x-axis in each graph represents the column value and the y-axis represents the frequency of the value. In Figure 9a, we show the distribution of the $role\_id$ column. The red bars represent the values for which we see the highest errors for the NN and RNN models, based on Table 2. Compared to the other existing selection attributes, $role\_id$ comes from the largest relation in the dataset, $cast\_info$. We generally observe that the models have the highest errors for columns that belong to the largest relations and specifically at the points where the distribution is irregular.

For the DMV dataset, the hardest queries are those that contain the $accidents$ relation and join with $time$ or $location$. These queries have selection predicates on the $year$ and $month$ columns. We highlight the selection predicate values in Figure 9f and Figure 9g. We note that there is a one-to-one mapping between the $accidents$ and $time$ relation, so the distribution for these columns does not change due to the join. This is also the case for the join between $accidents$ and $location$. The year column in the accidents relation has a high skew and the models have the highest errors for the more frequent values. The accidents relation also happens to be the largest relation in the DMV dataset.

For the TPC-H dataset, most of the errors come from the join between $lineitem$ and $orders$. These contain selection predicates on both the $l\_linenumber$ and $l\_quantity$. The pearson correlation for these two attributes is low (.0002) so these are independent attributes. We highlight the values with highest errors in Figure 9j and Figure 9k. We note that the $l\_quantity$ in particular has an irregular distribution, and also belongs to the relation with the highest number of tuples in the dataset, $lineitem$.

Unlimited Storage CDFs and Outlier Analysis In Figure 10, we show similar graphs across all datasets, but with an unlimited storage budget. The goal here is to understand how more complex models compare against the simpler ones from Figure 8. Given this unlimited budget, we now include the random forest models. The PostgreSQL estimates do not significantly change even with 100K bins, which implies that adding finer granularity to the histograms does not significantly improve estimates. Among all the models, the



(a) Space vs. Error for all models

(b) Time vs Error for all models

Figure 11: Trade-offs between Error, Space and Time: We show the absolute error, space and time for each model and for PostreSQL for different number of bins. The horizontal line represents the space and time for the hash table model.

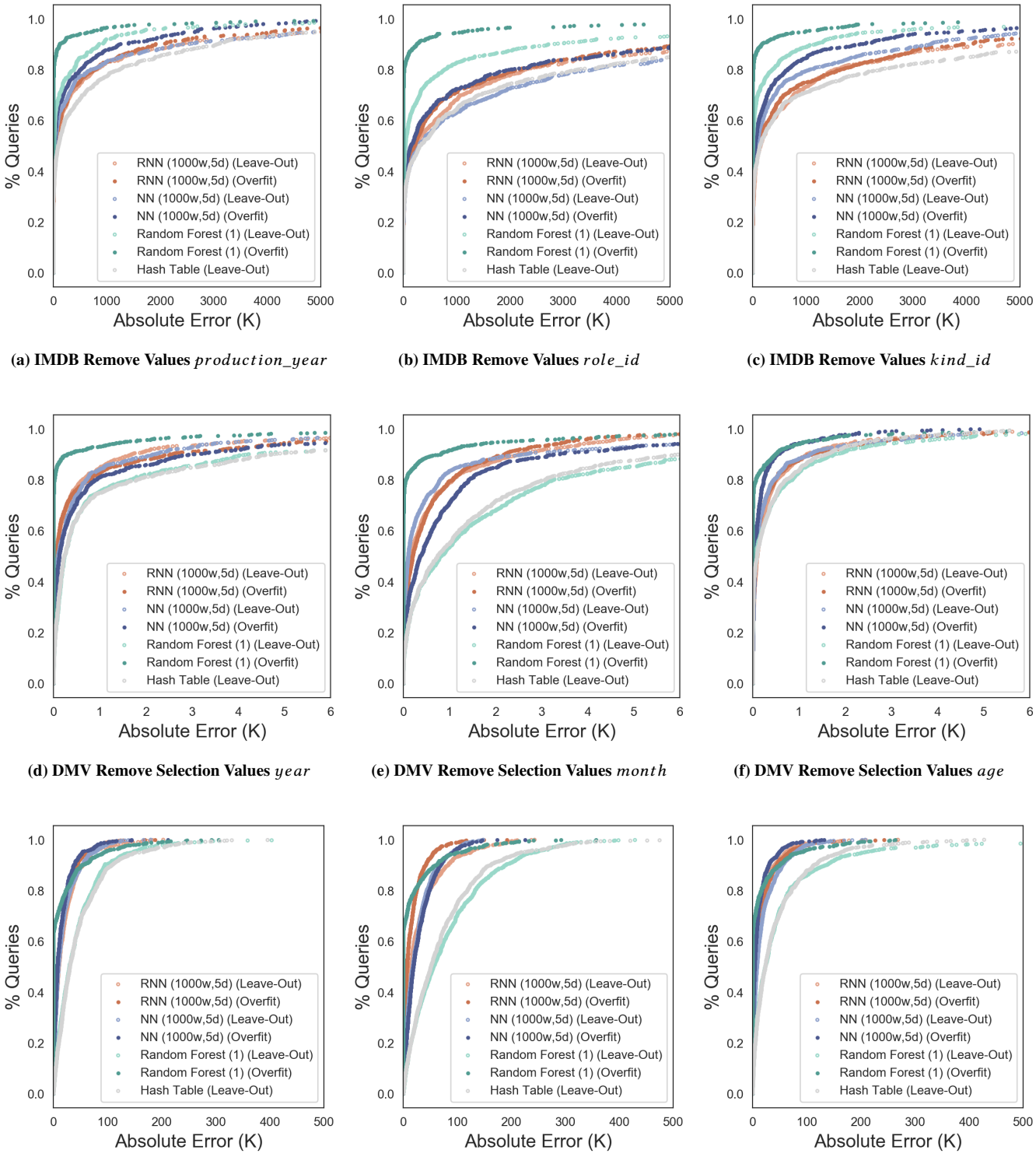
trees have the lowest errors overall across all query complexities and across all datasets.

Time vs Space vs Accuracy Trade-offs In Figure 11, we show the error, space and time trade-offs for each model. First, in Figure 11a we compare the error and space. On the y-axis, we show the absolute error between the predicted value and the real value on a log scale. On the x-axis we show the space of each model on a log scale. Each point represents the median error and the error bars represent the 25th and 75th percentiles. For all datasets, all variants of PostgreSQL have the highest errors and increasing histogram bin granularity does not significantly improve performance. Neural networks and recurrent neural networks are fairly competitive in terms of absolute error. In Section 5, we study whether deeper models actually learn more context about the relations compared to the shallower ones. Models that are deeper are much larger in terms of space, with small error improvement over simpler models.

In Figure 11b, we compare the accuracy to the time (in seconds) it takes to train each model. We do not include the time it takes to run the hyperparameter tuning and we do not include the time it takes to run the training queries. We address the latter overhead in Section 5.

Given their large sizes, an important question is whether the models improve upon simply keeping the entire query workload in a hash table (with query features as keys and cardinalities as values). To answer this question we plot the overhead of such a hash table. Given that our training data consists of only 100K samples for each query complexity, our goal is to understand whether the deep learning models can actually compress information and still provide a good accuracy. For the hash table model, we assume that each feature for each training example is equivalent to one weight when measuring space. To measure time, we measure the time it takes to populate the hash table. We mark this implementation in the graphs as a vertical dashed line. For this model, the error is 0 for each query.

For each dataset, all variants of the tree models result in the lowest error. In particular the trees with the lowest error are those with 1 decision tree. Since we build these models to overfit to a specific query workload, using a single decision tree results in the lowest

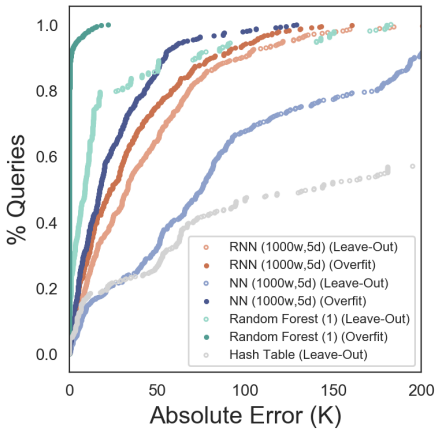


(a) IMDB Remove Values $production\_year$ **(b) IMDB Remove Values** $role\_id$ **(c) IMDB Remove Values** $kind\_id$

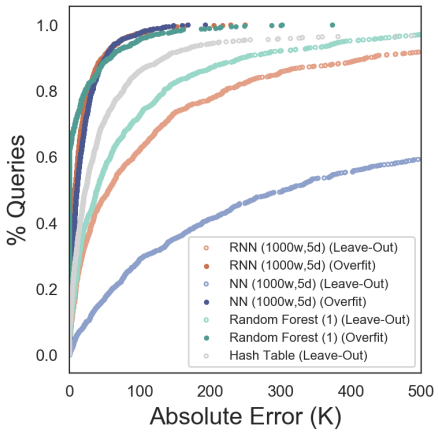
(d) DMV Remove Selection Values $year$ **(e) DMV Remove Selection Values** $month$ **(f) DMV Remove Selection Values** age

(g) TPC-H Remove Selection $l\_quantity$ **(h) TPC-H Remove Selection** $c\_nationkey$ **(i) TPC-H Remove Selection** p_size

Figure 12: Removing 10% selection predicate values across all datasets



(a) IMDB Remove Join



(b) TPC-H Remove Join

Figure 13: Remove Joins from the Training Workload

error. Once more decision trees are introduced, the error is higher as these models no longer overfit and attempt to generalize over the training set. These results suggest that for overfit workloads, the random forest model is able to build these models quickly and more accurately compared to the deep learning models. The deep learning models are able to save in space and although they are not as accurate as the trees, they can still improve errors in some cases by an order of magnitude compared to PostgreSQL.

4.3 Model Robustness

In this section, we study how robust these models are in the face of *unknown* queries. That is, instead of overfitting each model to a specific set of queries, we remove some query samples from the training data. We focus on the most challenging, 6*Join* datasets. We evaluate the most complex models for the RNN and NN (1000w, 5d) as these perform favorably for the 6*Join* set. We also select the best performing version of the random forest model from Figure 11 (Random Forest (1)).

Removing Selections In the first row of Figure 12, we remove 10% of values from three columns: *production_year* in Figure 12a, *role_id* in Figure 12b and *kind_id* in Figure 12c. In each graph, we have two variants of each model where Leave-Out (shown in lighter colors) is the case where we do not include these queries in the training data and Overfit (shown in darker colors) is the case when the models are trained on all possible queries. As shown in Figure 12c, for the IMDB dataset, the accuracy of the Random Forest model degrades significantly compared to the RNN and NN models for all columns, but still outperforms the other models. For both the DMV and TPC-H dataset (shown in rows 2 and 3) of Figure 12, removing values from training significantly decreases the accuracy for the random forest models. In these graphs, we also included the accuracy of the hash table model (Leave-Out) implementation. Since the samples in the test set are not included in the training set, we use a nearest neighbor approach to find the closest sample that exists in the training set (our hash table). We use the nearest neighbor implementation from sklearn [30] which uses the minkowski distance metric. In many cases, the hash model performs similarly to the random forest (Leave-Out) model. For IMDB the hash model is not as accurate. We generate 100K random query samples uniformly from the set of all possible queries, but unlike the other two datasets, 100K queries doesn't fully cover the set of all possible queries for IMDB. As a result, the nearest neighbor, is not always as close for this database as for the other two.

Removing Joins In Figure 13a and Figure 13b, we remove a join with a specific combination of tables from the IMDB and TPC-H 6*Join* datasets. During training, the models observe how certain tables join with each other, but they never see the specific combinations we remove. In Figure 13a, we remove the join between relations: { *complete_cast*, *aka_title*, *movie_info_idx*, *title*, *movie_companies*, *movie_link* } from the training set. The queries shown in the figure correspond to the test set, which includes the removed combination of tables with random selection predicates. When comparing between the Overfit and Leave-Out models, there is a large degradation in accuracy for the NN. In particular, the RNN and the random forest model have an advantage over the NN. For the IMDB dataset, we observe that the random forest model relies heavily on features from $\mathcal{I}_{selpred}$. We found that the IMDB dataset contains combinations of tables in the training data that are very similar (and yield the same cardinality) as the combination of tables we removed from the training. In fact, when observing the splits used in the tree, approximately 16% of the splits on average for all test samples are from the features in $\mathcal{I}_{relation}$. Whereas 67% of the splits are from $\mathcal{I}_{selpred}$. It is not clear why the NN model does not learn to rely more on information from $\mathcal{I}_{selpred}$. The hash table model has the worst accuracy, since the nearest neighbor at times selects queries with selection predicates on the same values but different underlying tables.

In Figure 13b, we observe a similar trend. For this experiment, we remove a join from the TPC-H dataset with the relations: { *customer*, *lineitem*, *partsupp*, *nation*, *part*, *orders* }. Again, the RNN results in lower errors than the neural network, but the tree model still performs better than the RNN and NN. Compared to the IMDB models, the tree model for TPC-H more heavily depends on splits from $\mathcal{I}_{selpred}$. In fact, on average, 88% of the splits are from these selection features. One reason why the random forest might only split

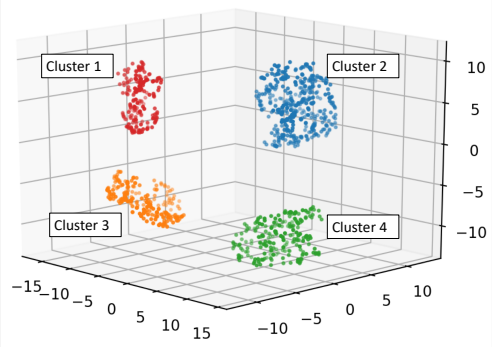


Figure 14: Clustering dimensionality-reduced latents for the NN (100w, 1d) model on the TPC-H dataset.

on selection features is due to TPC-H's schema. For all the queries in TPC-H's 6*Join* set, the join cardinality (without selections) result in approximately 6M tuples. This implies that regardless of the join combination, the maximum number of tuples is always the same, and the the selections are the defining feature that ultimately determine the final number of tuples.

4.4 Model Latents

One challenge of training deep neural networks is the difficulty to understand what the models are actually learning. As discussed in Section 4.3, the random forest models are easily interpretable as we can track path of decision splits to understand how the model is able to predict the outcome given the input. For neural networks, diagnosing why a model arrives at a specific answer is a harder problem. There are several existing approaches, which include masking or altering the input to measure the predication change and studying hidden unit activation values [17, 35, 45].

We study the activation values of the hidden layers for the NN and RNN models. During training, these models take the input, X, and propagate it through a series of transformations that represent the data at different levels of abstraction [14]. Taking a close look at the activation values (also referred to as *latent representations* or *embeddings*) can help diagnose what the model is learning from the inputs. For example, if we cluster training samples based on their latents, we can determine whether models are in fact generating similar representations for queries that are semantically similar.

We use the t-SNE technique to cluster latents, which is a dimensionality reduction technique that is often used to visualize high-dimensional data [40]. This approach has an objective function that minimizes the KL-divergence between a distribution that measures pairwise similarities of the objects in high-dimensional space and a distribution that measures the pairwise similarities of the corresponding low-dimensional points [40]. Compared to principal component analysis (PCA), t-SNE is non-linear and relies on probabilities.

We cluster latent vectors from the (100w, 1d) NN model for the 6*Join* training set from each dataset. In Figure 14, we reduce the dimensionality of the latents from the (100w, 1d) model on the TPC-H dataset (100 hidden units total) down to three dimensions, which is the highest number of dimensions allowed for t-SNE. In the figure, there are four clusters, each representing different sets of joins:

- Cluster 1: customer,lineitem,nation,orders,partsupp,region
- Cluster 2: customer,lineitem,orders,part,partsupp,supplier
- Cluster 3: customer,lineitem,nation,orders,partsupp,supplier
- Cluster 4: customer,lineitem,nation,orders,part,partsupp

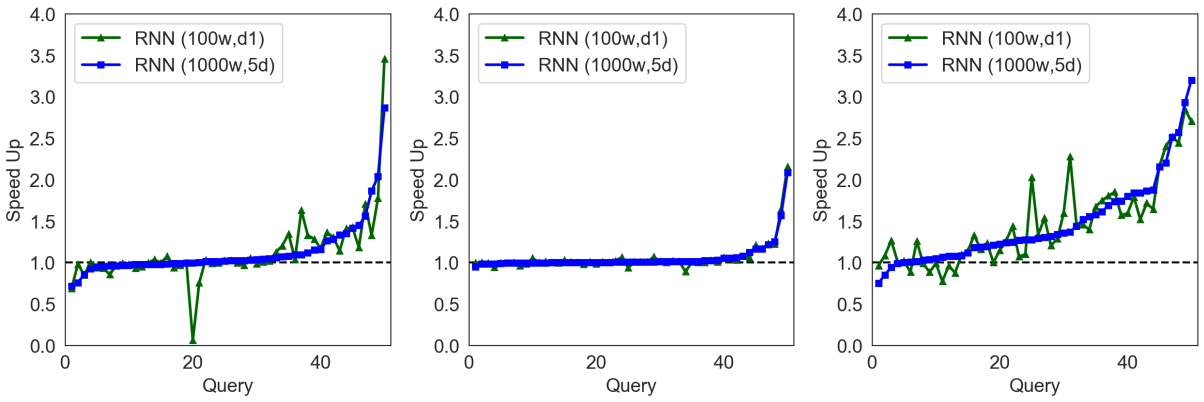
For t-SNE, the distance between clusters is irrelevant, the more important factor is the relevance among the points that are clustered together. For the DMV dataset and IMDB, the clusters do not represent combinations of relations, but we observe that queries that are near each other share similar selection predicate values.

For the RNN (100w, 1d) model, we find that clusters are determined based on the sequence of operations. Recall, during training, the RNN learns to predict cardinalities for different join sequences, as a result of observing many queries. We observe that the resulting clusters represent queries that end with similar operations. For example, one cluster contains combinations of relations *orders*, *lineitem*, *partsupp*, but always ends the sequence with joins on either the *supplier* and *part* relation or *customer* and *supplier*. We find that complex models (1000w, 5), also show a similar trend. This is actually a side-effect of RNNs, as more recent actions have a heavier influence on the content that exists in the hidden states. More specifically, it is difficult to learn long-term dependencies as the gradient is much smaller compared to short-term interactions [14].

As an additional experiment, we cluster the latents from queries that have not been included in the training. Ideally, although these queries have never been observed by the model, they should cluster with similar training queries. We focus on the TPC-H join removal scenario, originally shown in Figure 13b. When we cluster the latents from the (1000w, 5d) NN model, the queries that were not included in the training are clustered separately from the rest. This seems to imply that the NN does not learn the interactions between subqueries. This is not the case for the RNN, as queries that are left out of training are clustered together with queries that have similar subqueries. For example, a query that joins relations *lineitem*, *orders*, *partsupp*, *customer*, *part*, and *nation*, is clustered together with queries that contain relations *lineitem*, *orders*, *partsupp*, *customer*, *part*, and *supplier*.

5 PRACTICAL CONSIDERATIONS

In this section, we study two additional practical considerations. In Section 4, we evaluated the accuracy of cardinality estimates produced by the different models. In this section, we evaluate whether the cardinality estimate improvements lead to faster query execution plans. Additionally, in earlier sections, we showed the trade-offs between prediction error and time overhead due to model training. We did not consider the time that it takes to execute the training queries. To minimize this overhead, we consider using active learning as a way to reduce the time spent generating training sets.



(a) IMDB Impact **(b) DMV Impact** **(c) TPC-H Impact**

Figure 15: Query execution time speed-ups thanks to cardinality estimates from simple or complex RNN

5.1 Impact on Query Plans

We evaluate the impact of these models starting with a simple RNN model (100w, 1d) and going to a more complex one (1000w, 5d). We use the RNN, as query optimization requires evaluating cardinalities for several possible subqueries that could exist in the final plan. We evaluate the performance benefit for queries with 6 relations for each of the three datasets. As we collect the subquery cardinalities from the RNN, these estimates are then fed into a version of PostgreSQL modified to accept external cardinality estimates [5].

In Figure 15, we show the performance impact of these improved cardinalities compared to the default cardinality estimates from PostgreSQL. First, for the IMDB dataset, we show the performance improvement for 50 queries in Figure 15a. The runtimes for these queries range from <1sec up to 200sec. The simple RNN model improves the performance of 54% of the queries, while the complex model improves 60% of the queries. For the simpler model, query 22 is an outlier where the model's estimates actually slows down the query considerably (from 2 seconds up to 39 seconds). In contrast, there is no significant slow down on any query for the complex model.

For the DMV dataset, both the simple model and complex model improve the performance for 76% of the queries and there is no significant slow down for any query. We should note, however, that a majority of the query runtimes in this dataset range from 1 to 3 seconds. Finally, for the TPC-H dataset, the complex model improves 90% of the queries. The simpler model also makes a significant improvement, speeding up 84% of the queries. The query execution times for this dataset range from 20 to 120 seconds.

5.2 Reducing the Training Time

Building a model can be time consuming depending on the size of the model, the training time, and the amount of time that it takes to collect the training samples. To train the models shown in Section 4, we needed to run a large set of random queries to collect their ground-truth cardinalities, the output Y, for the models. Depending on the complexity of these queries, running them and collecting these labels can be time consuming. This process can be parallelized, but it comes with a resource cost.

Models can be trained in several ways. One approach to reducing the time to collect training samples, is to train the model in an online fashion. That is, as the user executes queries while using the system, the model can train on only those queries. The learning happens in an incremental fashion, and updates the model after observing a batch of samples. This approach can work well if the user executes similar queries. Online learning can also be fast and memory efficient, but the learning may experience a drift [13], where the model's decision boundary changes largely depends on the latest samples it observes.

Alternatively, instead of relying on a user to provide query samples, we can use a technique known as *active learning*. Active learning selects the best sample of candidates to improve a model's objective and to train as effectively as possible [20]. It is ideal in settings where labeled examples are expensive to obtain [6].

Active learning works through a series of iterations. In each iteration, it determines unlabeled points to add to the training sample to improve the model. Given a large pool of unlabeled samples, active learning will select the unlabeled sample that should be annotated to improve the model's predictions. In our context, given a large pool of unlabeled queries, active learning should help narrow down which queries to execute next.

There are various existing active learning methods. Common techniques include using uncertainty sampling, query-by-committee (QBC), and expected model change [20]. In this work, we focus on using QBC [36]. After each active learning iteration, QBC first builds a committee of learners from the existing training dataset via bootstrapping [43]. Each learner in the committee makes a prediction for all the samples in the unlabeled pool. The sample with the highest disagreement is labeled and added to the training pool. For regression tasks, this disagreement can be measured by the variance in the predictions across the learners [33].

Traditionally, active learning only adds a single informative sample in each data sampling iteration [4]. More recently, *batch-model* AL (BMAL), where multiple samples are labeled in each iteration has become more prevalent, as labeling in bulk or in parallel has been more accessible in recent years [7]. As shown in work by Wu et.

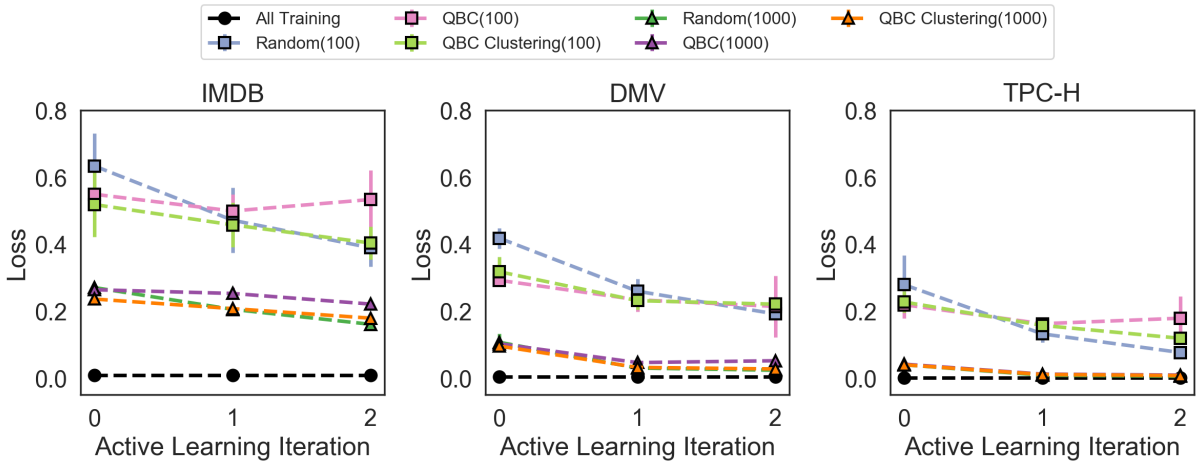


Figure 16: Active Learning

al. [43] careful attention must be placed in picking out diverse points with BMAL, as models might disagree on a batch that contains very similar points, leading to suboptimal results.

We use BMAL in the following experiment and run three different methods to help select the unlabeled points for each iteration:

(1) **QBC**: after each iteration, we train an ensemble of models and select the top K points with the highest disagreement
(2) **QBC+Clustering**: we train an ensemble of models, but pick out the top K *diverse* set of points through clustering, which is based on the technique from [43] for linear regression
(3) **Random**: we select a random sample of points from the unlabeled pool

For each dataset, we use training samples from the $2Join$, $4Join$, and $6Join$ set along with all their subqueries, for a total of 600K samples for the model. We run two experiments. In the first experiment, we start with a small number of training samples (100) and set K=100. For the second experiment, we start with a larger sample (1000) and set K=1000. As the number of training samples is small, we include regularization to prevent overfitting.

In Figure 16, we show the loss of each technique for three active learning iterations on each dataset. We show the results for both experiments (K=100 and K=1000). Each point represents the average loss for three separate runs. For each graph, we also include the loss for the case where all samples are labeled (labeled as "all training").

In general, we find that with small training sets, QBC and QBC+Clustering result in a lower loss, particularly at the end of the first iteration. For subsequent iterations, the random technique performs just as well and in some cases even better, as in the TPC-H dataset for example. QBC is competitive, but it often overfits as shown by the cases where the loss increases (IMDB and TPC-H). This is expected, as BMAL techniques are known to select a distinct set of points to improve the loss more effectively.

When the training set is larger (K=1000), all techniques perform similarly, negating the immediate benefit of active learning. Nevertheless, adding fewer points rather than the entire training set can still reach a loss that is approximately an order of magnitude away from the loss that includes all the training data.

6 RELATED WORK

Learning Optimizers Leo [39], was one of the first approaches to automatically adjust an optimizer's estimates based on past mistakes. This requires successive runs of similar queries to make adjustments.

Similarly, in the effort of using a self-correcting loop, others have proposed a "black-box" approach to cardinality estimation by grouping queries into syntactic families [27]. Machine learning techniques are then used to learn the cardinality distributions of these queries based on features describing the query attributes, constants, operators and aggregates. They specifically focus on applications that have fixed workloads do not require fine-grained, sub-plan estimates.

Work by Marcus *et al.* [28] uses a deep reinforcement learning technique to find optimal join orders to improve query latency on a fixed database. They use cost estimates from PostgreSQL to bootstrap the learning and continuously improve the accuracy of the model's rewards during training. Related work by Sanjay *et al.* [23], also uses deep reinforcement learning to improve query plans, but they assume perfect cardinality predictions for base relations.

Neural Networks and Cardinality Estimation Liu *et al.* [26] use neural networks to solve the cardinality estimation problem, but focus on selection queries only. Hasan *et al.* [15] also only focus on selectivity estimation, but show that deep learning models are particularly successful at predicting query cardinalities with a large number of selection predicates.

Work by Kipf *et al.* [19] proposes a new deep learning approach to cardinality estimation by using a multi-set convolutional network. Cardinality estimation does improve, but they do not show improvement of query plans. In addition, our work explores the space, time, accuracy of these models across a variety of datasets.

Work by Kraska *et al.* [22] uses a mixture of neural networks to learn the distribution of an attribute with a focus on building fast indexes. In SageDB [21], this work is extended towards building a new system that learns the underlying structure of the data to provide optimal query plans. In their work, they state that one key aspect

in successfully improving these query plans is through cardinality estimation. They are currently working on a hybrid model-based approach to cardinality estimation, where they balance between looking for a model that can learn the distribution of the data and a model that can capture the extreme outliers and anomalies of the data.

Wu *et al.* [42] learn several models to predict the cardinalities for a variety of template subgraphs in a dataset instead of building one large model. Input features include filters and parameters for the subgraph, but they do not featurize information about the dataset (i.e. the relations). Thus, their models cannot make predictions for unobserved subgraph templates.

7 CONCLUSION

We show the promise of using deep learning models to predict query cardinalities. In our study we found that even simple models can improve the runtimes of several queries across a variety of datasets. Although these is a large training overhead, we can use techniques such as active learning to reduce the loss quickly without having to run a large set of queries.

Acknowledgements This project was supported in part by the Graduate Opportunities and Minority Achievement Program (GO-MAP) fellowship, NSF grant IIS-1524535 and Teradata.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Amazon AWS. http://aws.amazon.com/.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] W. Cai, M. Zhang, and Y. Zhang. Batch mode active learning for regression with expected model change. *IEEE Transactions on Neural Networks and Learning Systems*, 28(7):1668–1681, July 2017.

[5] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD 2019*, 2019.

[6] Wenbin Cai, Ya Zhang, and Jun Zhou. Maximizing expected model change for active learning in regression. *2013 IEEE 13th International Conference on Data Mining*, pages 51–60, 2013.

[7] S. Chakraborty, V. Balasubramanian, and S. Panchanathan. Adaptive batch mode active learning. *IEEE Transactions on Neural Networks and Learning Systems*, 26(8):1747–1760, Aug 2015.

[8] Surajit Chaudhuri and Vivek Narasayya. Program for TPC-H data generation with skew.

[9] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[10] Antonio Criminisi and Jamie and Shotton. *Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning*, volume 7, pages 81–227. NOW Publishers, foundations and trends in computer graphics and vision: vol. 7: no 2-3, pp 81-227 edition, January 2012.

[11] Misha Denil, David Matheson, and Nando De Freitas. Narrowing the gap: Random forests in theory and in practice. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 665–673, Bejing, China, 22–24 Jun 2014. PMLR.

[12] Todd Eavis and Alex Lopez. Rk-hist: An r-tree based histogram for multi-dimensional selectivity estimation. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 475–484, New York, NY, USA, 2007. ACM.

[13] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, March 2014.

[14] Ian Goodfellow et al. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[15] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Das Gautam. Multi-attribute selectivity estimation using deep learning.

[16] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[17] Minsuk Kahng, Pierre Y. Andrews, Aditya Kalro, and Duen Horng Chau. Activis: Visual exploration of industry-scale deep neural network models. *IEEE Transactions on Visualization and Computer Graphics*, 24:88–97, 2018.

[18] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *PVLDB*, 10(13):2085–2096, 2017.

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[20] Ksenia Konyushkova, Raphael Sznitman, and Pascal Fua. Learning active learning from data. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4225–4235. Curran Associates, Inc., 2017.

[21] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[22] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[23] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning.

[24] Viktor Leis et al. Cardinality estimation done right: Index-based join sampling. In *CIDR 2017*, 2017.

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[26] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[27] Tanu Malik, Randal C. Burns, and Nitesh V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR 2007*, pages 56–67, 2007.

[28] Ryan Marcus and Olga Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[29] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18, pages 4:1–4:4, New York, NY, USA, 2018. ACM.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[31] PostgreSQL. https://www.postgresql.org/.

[32] J. Ross Quinlan. Learning with continuous classes, 1992.

[33] Jakub Repicky. Active learning in regression tasks, 2017.

[34] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, 1979.

[35] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, 2017.

[36] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 287–294, New York, NY, USA, 1992. ACM.

[37] Sandro Skansi. *Introduction to Deep Learning - From Logical Calculus to Artificial Intelligence*. Undergraduate Topics in Computer Science. Springer, 2018.

[38] Statistics collection recommendations - Teradata. http://knowledge.teradata.com/KCS/id/KCS015023.

[39] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[40] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne, 2008.

[41] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016.

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[43] Dongrui Wu. Pool-based sequential active learning for regression. *IEEE transactions on neural networks and learning systems*, 2018.

[44] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1721–1736, New York, NY, USA, 2016. ACM.

[45] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.