Learned Approximate Query Processing: Make it Light, Accurate and Fast

Qingzhi Ma, Ali M. Shanghooshabad, Mehrdad Almasi, Meghdad Kurmanji, Peter Triantafillou
Department of Computer Science,
University of Warwick,
UK
{q.ma.2,ali.mohammadi-shanghooshabad,mehrdad.almasi,meghdad.kurmanji,p.triantafillou}@warwick.ac.uk

ABSTRACT

The advent of learning algorithms has revealed many opportunities for improving Data Systems' functionality and performance. Approximate Query Processing (AQP) is one such area where machine learning (ML) models have been used to improve query execution efficiency and accuracy, outperforming the traditional sampling-based approaches. Based on our group's experience in the ML-for-DBs area, [3–7, 29, 37–39], we contribute a novel AQP engine, coined *DBEst++*, which extends our previous effort (DBEst, [29]) and sets the state of the art in terms of accuracy and query execution efficiency. The *DBEst++* salient design objective is to derive lightweight ML models for the task, allowing a plethora of ML models to coexist, covering a very large fraction of the expected analytical query workload without requiring very large memory footprints. The *DBEst++* salient architectural feature rests on a novel blending of word embedding models with neural networks tasked with regression-based predictions for density estimation and aggregation-attribute values. We present design features and motivations/rationale behind *DBEst++* and discuss how all the ML models are brought together. We also present how *DBEst++* can deal with challenging scenarios, including how to deal with high-cardinality categorical attributes and how to ensure high accuracy under data updates. We provide a detailed experimental evaluation using the TPC-DS and Flights datasets against state of the art learned and sampling-based AQP engines, showcasing *DBEst++*'s gains in terms of accuracy, response-times, and memory space overheads.

ACM Reference Format:
Qingzhi Ma, Ali M. Shanghooshabad, Mehrdad Almasi, Meghdad Kurmanji, Peter Triantafillou. 2020. Learned Approximate Query Processing: Make it Light, Accurate and Fast. In *Proceedings of CIDR '21: 11th Annual Conference on Innovative Data Systems Research (CIDR '21).* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

Augmenting the functionalities of database systems with ML models is receiving great attention nowadays. Such ML models take various forms, including classical regression and density estimators (like XLeratorDB [12] for Microsoft SQL Server, MADLib [20] over PostgreSQL), or deep neural networks (like [21, 33], and for tasks such as deriving learned cost models [23, 40]), workload forecasting [28], database tuning [25, 44, 46], cardinality and selectivity estimation [18, 21, 45] and learned indexing [26] etc).

Approximate query processing has traditionally relied on sampling approaches. These are largely classified as online [22] (i.e., the sample is generated after the query arrives) or offline (i.e., samples are generated in advance, either for popular queries [2] or for all possible queries for the schema [34]). As very large sample sizes are typically required to achieve high accuracy - a fact that necessarily implies poor response times, the community started looking into alternative approaches. More recently, machine learning models were adapted for processing various aggregate queries instead of using (samples of) data. The first efforts by our group focused on using regression-based techniques to predict and/or explain approximate query answers [4–7, 30, 37, 39]. Learned AQP engines also emerged that would holistically process aggregate queries, promising to improve both accuracy and efficiency. The first such effort to our knowledge was DBEst [29], followed by DeepDB [21], and [43], etc. Learned AQP engines, like DBEst and DeepDB, adopt a data-driven perspective. Specifically, uniform samples are firstly generated, and models are trained based on samples. Subsequently only the models are used for query processing, For instance, DeepDB trains Relational Sum Product Networks (RSPNs) over the tables' columns, whereas DBEst trains Kernel Density Estimators (KDEs) and Regression Models (RMs) over column sets. Despite these developments and the improvements they introduced in terms of accuracy and efficiency, much more is left to be done, with respect to efficiency and accuracy and, more importantly, in terms of related memory overheads.

The remainder of this paper is organized as follows. Section 2 presents the rationale, motivations, and overall vision and contributions of *DBEst++*. Section 3 overviews the *DBEst++*internals. It explains its core ML models, how models are trained, and how models are used for inference. Section 4 demonstrates the performance of *DBEst++*for the TPC-DS and the Flights datasets and compares it against DeepDB and VerdictDB. Section 5 overviews related work, Section 6 introduces future work and Section 7 concludes.

2 DESIGN CHOICES, RATIONALE AND MOTIVATIONS

Given the good success thus far, and the large promises of ML for improving data systems internals, many a researcher are expected to continue to contribute more and more ML models for various data processing tasks. *Unfortunately in our view, this is done without much consideration to the aggregate requirements these models will place*

This article is published under a Creative Commons Attribution License(http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well as allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021. 11th Annual Conference on Innovative Data Systems Research (CIDR '21). January 10-13, 2021, Chaminade, USA.
CIDR '21, January 10–13, 2021, Chaminade, USA
©

holistically to the system which, after all, will be called to integrate all of these intelligent functionalities. Primarily we are concerned here with space/memory requirements of said ML models – hence, our emphasis on *light* models.

Following this rationale, a salient design feature of the proposed *light* engine, which goes against the grain in the current school of thought, is that it avoids the development of *universal* models: These models aim to be able to answer all queries involving any possible combination of attributes of a given schema. While the benefits of these approaches are highly touted, such universal models are typically very large and coarse-grained. As such, they waste all of the memory required to store a universal model to support all possible queries when typically only a very small subset (among all possible) queries will be executed (i.e., the queries involving popular combinations of columns).

Viewed from a different angle, as we move away from data accesses to model accesses, we view the ML models we propose as the counterpart to indexes (and other access structures) and data used traditionally for answering a query at hand. Except that the models should be dramatically smaller so that a large number of them would already be in memory and, if not, the time cost for their IO and (de)serialization would be also very small.

In the overall vision, the AQP engine would employ a query-to-models index, in order to map an incoming query to the model(s) it needs. Said models, as argued above, would likely be in memory already or would be fetched and deserialized from nonvolatile memory very fast.

Complementarily, a key issue is what are the appropriate models to leverage in order to develop these light AQP engines? Although this is an open problem, we will present our approach based on specific ML models, utilizing word embeddings and mixture density networks (MDNs) which, when appropriately combined, provide excellent space-accuracy-time performance.

Therefore, our key concern and contribution with this paper is the development of a learned AQP engine that:

- pushes the lower bounds of required space for its ML models (offering orders of magnitude space savings), while
- offering top-notch query execution times (especially being embarrassingly parallelizable, reducing query times at will with additional investment), and
- offering the highest accuracy (circa 2X better than state of the art learned approaches for more demanding datasets), and
- ensures its high accuracy, even in the presence of data updates, and
- can deal effectively with high-cardinality categorical attributes, which introduce challenging space/time vs accuracy dilemmas.

Extensive experimentation with real and benchmark datasets will showcase the gains introduced by *DBEst++* and analyze key sensitivities.

3 SYSTEM OVERVIEW

3.1 DBEst++ Query Processing Foundations

This paper shares the similar mathematical foundations as DBEst [29]. As an example, given regression model $y = R(x)$ and density estimator $D(x)$, *DBEst++*uses the following formula to produce approximate answers to SUM queries.

$$SUM(y) = N \cdot \int_{lb}^{ub} D(x)R(x)dx \tag{1}$$

where N is the scaling factor, and lb, ub are the lower bound and upper bound of the range selector. The formula for COUNT and AVG are addressed in DBEst [29]. They are omitted for space reasons.

Unlike DBEst which used KDEs and a Regressor (XGBoost), *DBEst++* employs MDNs for both the density estimation and the regression tasks. Using these neural networks in *DBEst++*avoids the need for having different models (say for different group and categorical values in the WHERE clause) with a single neural network handling all. Furthermore, MDNs help with updatability and also improve accuracy, especially when combined with embedding models.

Due to the simplicity of MDN models, *DBEst++*could easily be extended to also support other aggregates more efficiently. Take VARIANCE queries as an example. As mentioned, the output of MDN models is a mixture of Gaussians. Specifically, $p(x) = \sum_{i=1}^{m} w_i \cdot \mathcal{N}(\mu_i, \sigma_i)$. And VARIANCE is obtained by [13].

$$\begin{aligned} Var(x) &= E[(x-\mu)^2] \\ &= \sum_{i=1}^{m} w_i(\sigma_i^2 + \mu_i^2 - \mu^2) \end{aligned} \tag{2}$$

where $\mu = E[x] = \sum_{i=1}^{m} w_i\mu_i$.

3.2 System Architecture

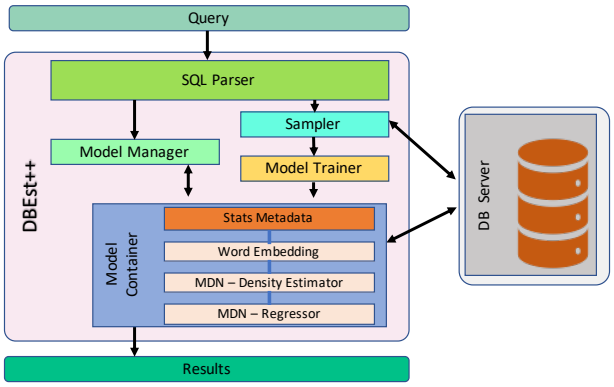

Figure 1: *DBEst++* System Architecture

Figure 1 shows the system architecture of *DBEst++*. *DBEst++* consists of several components: (i) The Parser parses incoming SQL queries, and checks whether the query is a SELECT query or a MODEL CREATION query; (ii) The Model Container maintains in-memory metadata and the models (i.e., MDNs and embeddings). MDN models are used to provide accurate predictions for probability densities of variables (attributes) and for values of dependent variables given independent variable values (such as those set by relational selection operators and/or group ids in GROUP BYs); (iii) The Model Manager selects the appropriate models from the Model Container to use per query and also selects representative data points from the

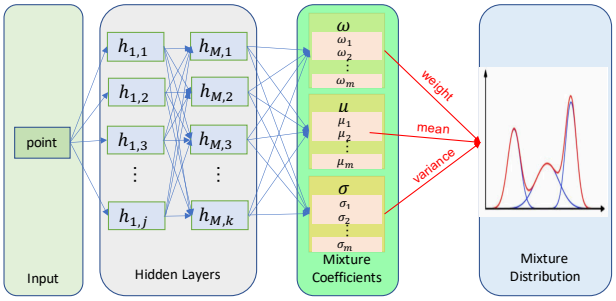

Figure 2: Structure of Mixture Density Networks

range of variable values specified in selection range predicates with which to call for MDN predictions. These representative values are used to (approximately) evaluate the integrals needed for the approximation (as shown above) and aggregates the predictions to provide the final approximate query answer; (iv) The Sampler interfaces with the DB in order to create samples of tables, based on which ML models will be build; (v) The Model Trainer module trains embedding and MDN models upon the drawn samples.

Model Creation Query. Suppose the user asks to create a model to answer SQL queries of the following format:

```
SELECT g, AF(y) FROM tbl
WHERE x BETWEEN low AND high
[AND city="London" AND ...]
GROUP BY g
```

(where aggregate function AF is typically COUNT, SUM or AVG.) The Sampler will by default use reservoir sampling to make random samples for table tbl. Afterwards, the Model Trainer will train one MDN model for density estimation of the independent variables (i.e., x given g, $P(x|g)$) and one MDN model for regression, yielding essentially $P(y|g,x)$.

SELECT Query. For a SELECT query, the Integral Module will select representative points between *low* and *high* for x, ask the corresponding MDN models to make predictions for these representative points, and aggregate to provide the final approximate results, as explained later.

3.3 Mixture Density Networks

There are two types of MDN models employed by *DBEst++*. The MDN-regressor is used for regression tasks and the MDN-density is used for density estimation. The structures of the networks are the same for MDN-regressor and MDN-density. However, the inputs to the two network types are slightly different.

MDNs are simple and straightforward - one of the main reasons we selected them. Combining a deep neural network and a mixture of distributions creates a MDN model. Many modern neural networks could be easily extended to support MDNs, including LSTMs, CovNets, etc. We chose to use MDNs as they are widely applied to solve real-world problems [16, 17] with high success. For instance, Apple uses MDNs for speech recognition [41].

Figure 2 shows the structure of typical mixture density networks. The cost function is the average negative log-likelihood (NLL), and gradient descent is used to minimize the cost function. Assuming the input features are $\mathbf{x}$, and labels are y, NLL takes the format of

$$\underset{\theta}{\operatorname{argmin}}\, l(\Theta) = -\frac{1}{|\mathbb{D}|} \sum_{(\mathbf{x},y)\in\mathbb{D}} \log p(y|\mathbf{x}) \tag{3}$$

As said, the input features and labels are different for the regression and density estimation tasks. Consider a simple query of the following format

```
SELECT g, AF(y) FROM tbl
WHERE x BETWEEN low AND high
GROUP BY g
```

For the density estimation tasks, the input features are the word embedding format of the g values, coined $WE(g)$ (which will be introduced in Section 3.4), and the corresponding labels are $\mathbf{x}$. The MDN density estimator aims to predict the distribution of $\mathbf{x}$ for all groups. For regression tasks, the input features are $[WE(g), \mathbf{x}]$, and the corresponding labels are y. The task of MDN regression is to predict the average value of y for a given group g and $\mathbf{x}$. Figure 3 summarizes the input features and labels for training MDNs. In general, the output of MDN models is a mixture of Gaussians which can model the distribution of values of the dependent variable(s) (e.g., y) given the values of independent variables (e.g. x and g). The central hyper-parameter in MDNs is the number of Gaussians used. And grid search is used to find the optimal number of Gaussians for each query template.

(a) MDN density estimator **(b) MDN regression**

Figure 3: Input features and labels for training MDNs

3.4 Word Embeddings

In ML, dealing with categorical (nominal) variables is always challenging. Some of the ML algorithms can easily deal with categorical variables such as Decision Tree algorithms [36] and Association Rule Mining methods [1], but many modern ML methods based on NNs can only operate on numerical/continuous data. This means those variables must be converted to such a form. One-hot, binary, dummy variable or integer(ordinal) encoding methods are usually used for this aim. These methods map categorical values into arrays of 0 and 1s, and since the output is numerical, they can be used in all ML methods. Nonetheless, none of the mentioned encoding approaches can assign a meaning into output values. For example, a binary encoding for "red" and "blue" cannot give us information about how similar these colors are. To capture the meaningful encoding for categorical values, mostly, embedding approaches are used. Once a meaningful vector representation for each single

categorical value was learned, it could significantly improve the accuracy of the models.

There are many embedding approaches like [9], [27] and [14], but Skip_Gram [31] have been highly successful because of the deep linguistic theory behind it. In this paper also, the Skip_Gram model is used to transform group-attribute values and other categorical attributes into a real valued vector representation. As categorical attributes have no meaningful distance between successive values (ie city="Toronto" vs city="New York"), learning a relationship between a categorical independent variable and a dependent one is very difficult. Using word embedding introduces such a meaningful distance between different independent categorical-attribute values (e.g., g, x) so that learning $P(y|g, x)$ becomes easier and more accurate.

For instance, Figure 4(a) shows the salary information of employees in different cities. Toronto and New York share more common salaries (40-45k in this example) than Toronto and a small town. Therefore the embedding vectors for Toronto and New York will be similar, whereas for Toronto and small towns their distance in the embedded space will be much larger.

Figure 4: Data Pre-processing for Word Embeddings.

To use word embedding approaches in our solution, we need to prepare the training data in the same way NLP methods work. For our task, instead of dealing with sentences in a document, we have rows in a table. When preparing the training data, instead of creating a dataset of pairs of words that come together in the sentences, we create the dataset with the pairs of categorical-attribute values that come together in a row of the table. For example, for a row like ("London","red","Laptop"), to learn the embedding for the first attribute (City names), we create two training data pairs like ("London","red") and ("London","Laptop"). These pairs are given to the Skip Gram model which tries to find similar vector representations for cities that have common pairs. To create the training pairs, we only use the attributes that are involved in the query. If the involved attributes are not categorical, we discretize them first. Furthermore, it is possible that a distinct value exists in two different attributes, so to avoid pushing wrong information to the Skip Gram model, we add a prefix for each distinct value in the attributes. For example, if the name of the attribute is "City" and the value is "London", we instead use "$City\_London$".

The key hyper-parameter for word embeddings is the size of the embedding vector. Grid search is used to find the optimal vector size with respect to accuracy.

3.5 Updatability

DBEst++ supports data updates - here we discuss insertions of new data that was not seen when building the model. This is challenging because in the end *DBEst++* must maintain all info it has gleaned from 'old' data, while also learning to incorporate the new data in its knowledge.

We sketch and compare two naive approaches: (a) Only frequency tables (FTs) are updated; (b) both frequency tables and the MDN models are updated. The general scheme of the experiment is described as follows: Firstly, a base model is trained, as discussed in the previous sections. Subsequently (batches of) new data items arrive. When new batches of data arrive, we aim to handle these updates with minor changes to the model.

In the first method, we only update the frequency tables and predictions are made based on the previously learned MDNs. In the second approach, we also update the MDNs by re-using the weights from a previous state (e.g., the old original model) and retrain the model using the new batch of data. Our first approach basically evaluates the generalization of the MDN model to unseen data. After updating the FTs, the model relies on the generalization of MDNs to produce approximate answers for queries now involving unseen data. Although this method is fast and easy to implement, it disregards the major part of information that are introduced in the new data.

The second approach, on the other hand, avoids missing any new knowledge by retraining the MDN model on the new data batches. This is achieved as follows. The weights of the old MDN are retained and copied into a new MDN structure. Then the new data items in the batch are feed-forwarded to the new MDN, which adjusts its weights accordingly using back propagation.

However, this approach may suffer from the problem of "catastrophic forgetting". In other words, while we fine-tune the MDN network on the new data, the new MDN fits the new distribution and forgets the knowledge that has been acquired previously. Most previous works [15, 24, 35] that address this problem require major efforts that sometimes also include the deformation of the architecture.

For our task we have achieved highly promising results with a rather simple idea: while updating the MDN model on new data batches, the learning rate used for learning from each new batch is kept smaller. The insight behind this idea is that the smaller learning rate, create finer changes in the model's weights. Therefore, the model does not drastically forget the previous knowledge. At the same time, it learns from the new batches of data. The detailed results are provided in Section 4.

4 PERFORMANCE EVALUATION

All code, datasets, and query workloads used in the following experiments can be found at: https://github.com/qingzma/DBEst_MDN.

4.1 Experimental Setup

We have evaluated *DBEst++* using column sets from queries in TPC-DS dataset [32]. We use scaling factors 10, 100 and 1000 to produce three versions of the dataset, with sizes of 10GB, 100GB and 1TB, respectively. Firstly, comparisons are made between *DBEst++* and universal approaches: Namely, a learned approach, DeepDB, and a sampling-based one, VerdictDB. We compare space overheads, accuracy (relative error), and query response times. As the DeepDB code did not support TPC-DS, we had several interactions with the DeepDB authors and used their suggestions to properly

tune DeepDB for this setting. To evaluate how well the *DBEst++* models work as lightweight models, we also compare against a "compact" version of DeepDB, whereby it is trained only over the same columns as *DBEst++*. We further demonstrate the embarrassingly parallelizable nature of *DBEst++*, using parallel inference. We also evaluate *DBEst++* with a real-world dataset, the Flights dataset, [1], which was also used in the DeepDB paper. IDEBench [11] is used to scale up this dataset to contain 1 billion tuples.

In addition, we report on our experiments and results regarding the following key issues with respect to *DBEst++* (and, actually, any machine-learning-based method for AQP): Namely, (i) the performance/sensitivity of the models with respect high-cardinality categorical attributes, (ii) the impact to accuracy that the embedding model within *DBEst++* have on accuracy, (iii) the accuracy performance of the approach that *DBEst++* adopts for updatability, and (iv) the performance of the parallel version of *DBEst++* on query response times.

Hyper-parameter tuning for the *DBEst++* models was as follows. For the MDNs, we used values between 5 and 20 for the number of Gaussians. For embedddings we used Word2Vec (from the gensim package [2]) to train Skip_Gram models with vector size values varying between 15 and 35.

4.2 TPC-DS Dataset

For fairness, *DBEst++* and DeepDB use the same samples (from the original dataset) to build their models. Then, 30 queries are randomly generated, covering `COUNT`, `SUM` and `AVG` in equal portions and containing `GROUP BY` and different selections operators.

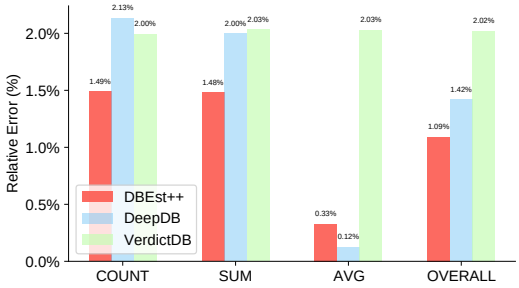

Figure 5: Relative Error for SUM / COUNT / AVERAGE Queries over the TPC-DS Dataset (SF=10)

4.2.1 Universal Models. Figure 5 summarizes the average relative errors of *DBEst++*, universal DeepDB and VerdictDB for `COUNT`, `SUM` and `AVG` queries. The relative error of VerdictDB is around 2% for all aggregate queries, and is the highest among all. DeepDB has similar accuracy as VerdictDB for `COUNT` and `SUM` queries. For `AVG` queries, DeepDB achieves the least error (0.12%). The relative error of *DBEst++* is much smaller than that obtained by DeepDB or VerdictDB for `COUNT` and `SUM` queries. And the overall relative error by *DBEst++*is only 1.09%!

The same experiment is repeated for TPC-DS with scaling factors (SFs) equal to 100 and 1000. Relative errors for `COUNT` and `SUM` are shown in Figures 6 and 7. Again, *DBEst++* achieves smaller errors across all SFs.

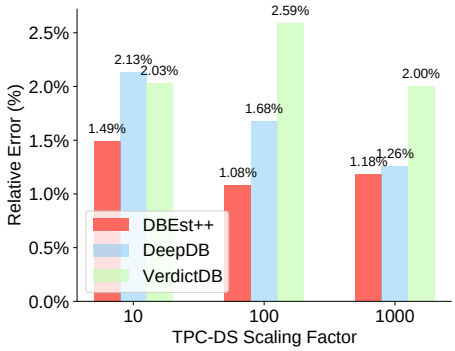

Figure 6: Scalability for COUNT Queries Varying SF

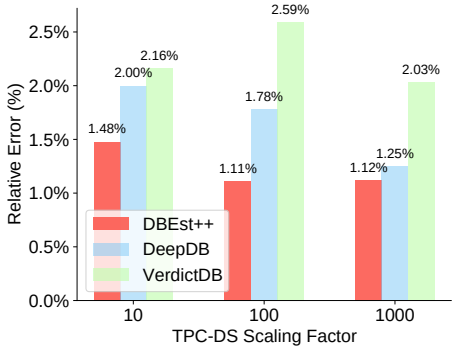

Figure 7: Scalability for SUM Queries Varying SF

Figure 8 shows the corresponding query response times. VerdictDB requires 1 order of magnitude longer time than model-based AQP engines. This shows the strength of models for fast query processing. Query response times of *DBEst++* are slightly lower than those for DeepDB. Both *DBEst++* and DeepDB require less than ca. 400ms (even for SF=1000) to respond to all queries. *DBEst++* outperforms DeepDB by a factor of ca. 25% to 40%.

As *DBEst++* is put forth also as a "light" learned-AQP approach, we now turn our attention to required memory space for the various approaches. So, space-wise, as shown in Figure 9, *DBEst++* achieves ca. 3 orders of magnitude savings compared to DeepDB or VerdictDB! Interestingly, universal DeepDB requires even higher space overheads than VerdictDB.

4.2.2 Compact Models. Figure 10 corresponds to Figure 5, showing the relative errors for `COUNT`, `SUM` and `AVG` queries over the TPC-DS dataset with SF=10. Figure 11 corresponds to Figures 6 and 7, demonstrating the overall accuracy of *DBEst++* and DeepDB for the TPC-DS dataset, and as the dataset scales up. Clearly, *DBEst++* achieves higher overall accuracy than DeepDB. It is interesting to note that when switching from universal DeepDB to compact

[1] https://www.kaggle.com/usdot/flight-delays
[2] https://radimrehurek.com/gensim/models/word2vec.html

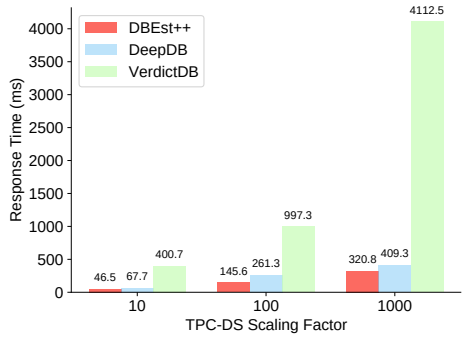

Figure 8: Comparison of Query Response Times for Queries over the TPC-DS Dataset

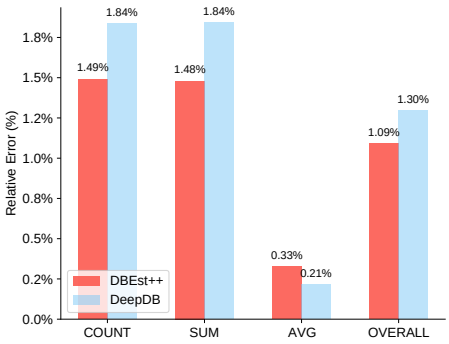

Figure 10: Accuracy comparison for Compact Models for Queries over the TPC-DS Dataset (SF=10)

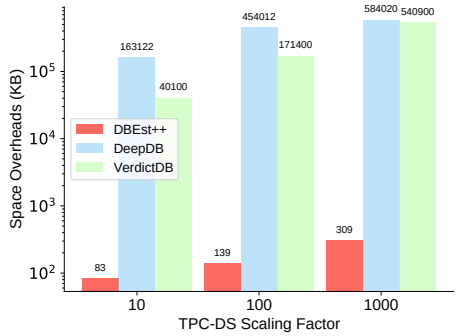

Figure 9: Comparison of Space Overhead for Queries over the TPC-DS Dataset

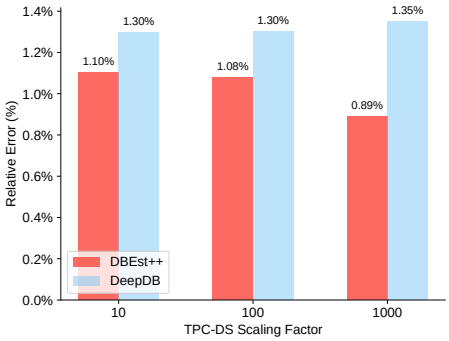

Figure 11: Comparison of Overall Relative Error for Queries over the TPC-DS Dataset

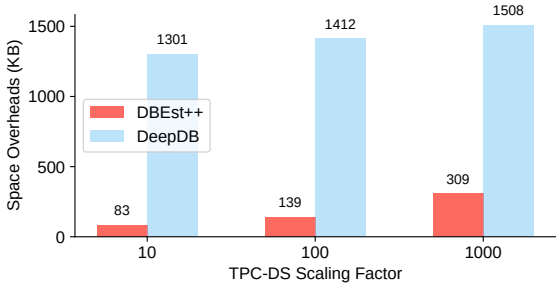

Figure 12: Space Overheads for Queries over the TPC-DS Dataset.

DeepDB, we see a reduction in relative error. Take TPC-DS with SF=10 as an example: the relative error for COUNT obtained by universal DeepDB is 2.13% (see Figure 6). The corresponding relative error obtained by compact DeepDB is 1.84% (see Figure 10). This shows that building a universal model for all types of queries may lead to higher errors.

Figure 12 compares the space overheads between *DBEst++* and compact DeepDB. As compact DeepDB is trained over relevant columns only, the size of RSPNs reduces significantly (compared against that of universal DeepDB). Despite this reduction, *DBEst++* still outperforms with respect to space overheads by about 1 order of magnitude. This testifies that the *DBEst++* approach which integrates embeddings plus the two MDNs truly delivers in all fronts.

4.3 Flights Dataset

We now further evaluate the performance of the above approaches using the Flights dataset, which was also used in the DeepDB paper. 9 queries covering COUNT, SUM and AVG are taken from the DeepDB paper and are used here. Samples of size 1m and 5m are used to train models for *DBEst++* and universal DeepDB.

Figure 13 shows the relative errors for the Flights dataset. This dataset is much simpler and easier to achieve great performance than TPC-DS. We include it here only in order to have a common test dataset to compare against DeepDB. Both *DBEst++* and DeepDB enjoy extremely high accuracy. For instance, if the 5m sample is used, the overall relative error is below 0.5% for *DBEst++* and DeepDB. As the sample size increases from 1 million to 5 million, we see a reduction in relative errors for *DBEst++* and DeepDB.

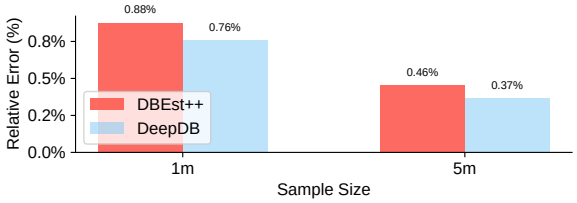

Figure 13: Accuracy Comparison for Queries over the Flights Dataset

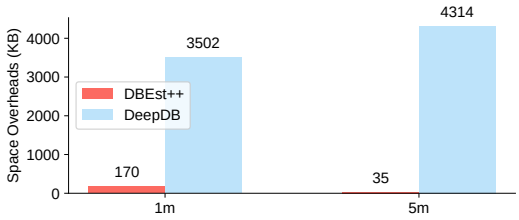

Figure 14: Space Overheads for Queries over the Flights Dataset

Figure 14 compares the space overheads between *DBEst++* and DeepDB. Again, we see a big difference in space overheads between *DBEst++* and DeepDB. For instance, if the 5m sample is used, *DBEst++* requires 35 kilobytes, while DeepDB needs 4.3 megabytes. This represents more than 2 orders of magnitude savings in memory footprint.

4.4 Impact of Word Embedding

Section 3.4 introduced word embeddings within *DBEst++* for categorical attributes, and explained the rationale and intuition underpinning its utilization and expected improvements over other the traditional techniques, such as one-hot encoding and binary encoding for inputing data to neural networks. We expected that word embeddings would group together "similar" items in the embedded space. Similarity here refers to the values of attributes among different rows. (Therefore, the model's accuracy is improved.) Using only a one-hot or binary encoding would fail to capture such latent relationships between items as they would be transformed into an orthogonal representation in another dimension.

We conduct the same experiments as in Section 4.2. Instead of using word embeddings, one-hot encoding or binary encoding is used to input categorical attributes into the MDNs. This would reveal the gains due to embeddings.

Figures 15 and 16 summarize the relative error of *DBEst++* for queries over the TPC-DS dataset using one-hot and binary encoding, and word embedding. Take `COUNT` queries as an example, as shown in Figure 15. For scaling factor equal to 10, the relative error is 3.26% (3.73%) if binary (one-hot) encoding is used. We see a significant decrease in relative error if word embedding is used – the corresponding relative error is only 1.49%. The same conclusion holds for the TPC-DS dataset with various scaling factors and `SUM`, `AVG` queries. Also, it is worthwhile to note that we do not have statistics of one-hot encoding for scaling factor=1000. One-hot encoding requires much larger memory and is not ideal for large groups.

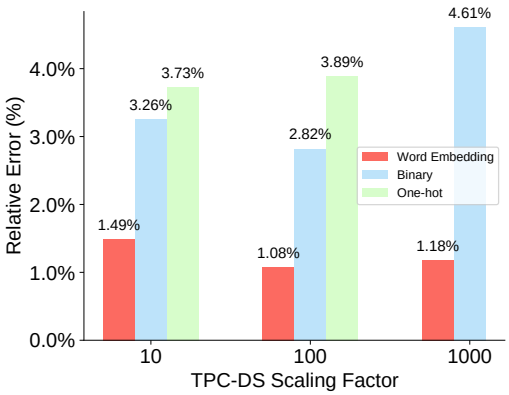

Figure 15: Comparison of Relative Error Between Word Embedding, One-hot and Binary Encoding for COUNT Queries.

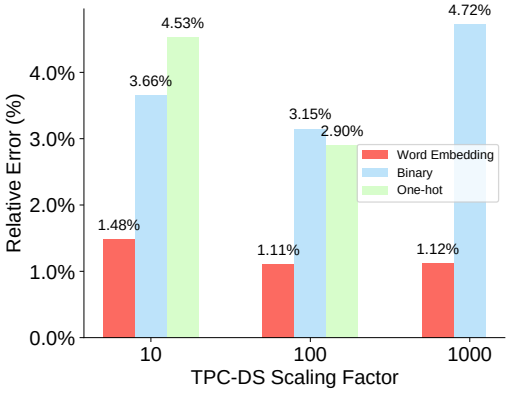

Figure 16: Comparison of Relative Error Between Word Embedding, One-hot and Binary Encoding for SUM Queries.

4.5 Sensitivity to Attribute Cardinality

It is known that sample-based AQP solutions (like VerdictDB) are challenged for `GROUP BY` queries with a large number of groups. There must be enough representative points per group to guarantee high accuracy. As a consequence, the sample size must be greatly increased to achieve good accuracy. Expert readers will also know that even ML-based approaches struggle with high-cardinality categorical attributes. Here, we compare the performance of *DBEst++*, DeepDB and VerdictDB for 30 `GROUP BY` queries with the following query template:

```
SELECT ss_store_sk, ss_quantity, AF(ss_sales_price)
FROM store_sales
WHERE ss_sold_date_sk BETWEEN low AND high
GROUP BY ss_store_sk, ss_quantity
```

where the aggregate function (AF) is COUNT, SUM or AVG, and the range predicate is randomly generated within the space domain. Here, the TPC_DS dataset is scaled up with SF=1000, resulting in 2.8 billion tuples and the grouping attribute has more than 50,000 distinct values (groups). The sample size ranges from 2.5m to 30m.

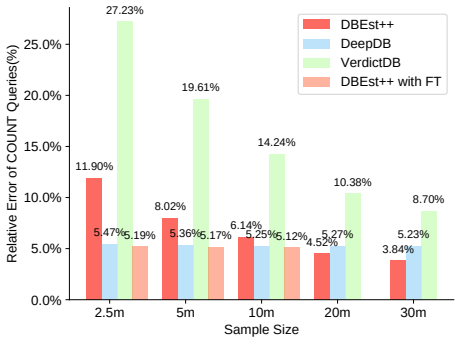

Figure 17: Comparison of Sensitivity on Large Groups for COUNT Queries.

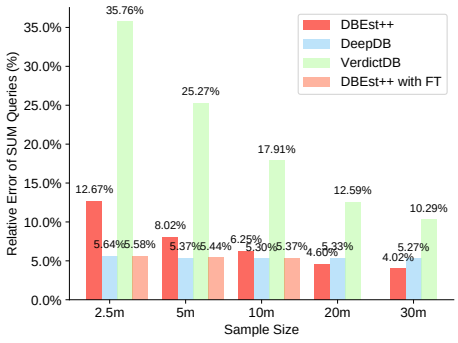

Figure 18: Comparison of Sensitivity on Large Groups for SUM Queries.

Figures 17 and 18 show the effect of sample size on relative error for *DBEst++*, DeepDB and VerdictDB. Take COUNT queries as an example, as shown in Figure 17. With a sample of size 2.5m, VerdictDB's relative error (27.23%) is unacceptably high. The relative error for *DBEst++* and DeepDB is 11.90% and 5.47%, respectively. This substantiates the intuition that the sample-based AQP engine VerdictDB performs worse for large groups. However, note that also the accuracy of *DBEst++* is poor. On the other hand, the error of DeepDB is better for small sample sizes. However, note that this error, albeit better for small sample sizes, it is still very high (approximately 5%). And, unfortunately, it is stable even for increasing sample sizes. As the sample size increases, the error of *DBEst++* and VerdictDB decreases.

As the sample size increases to more than 20m, *DBEst++* overtakes DeepDB and becomes the most accurate AQP solution for large groups. The same conclusion holds for SUM queries.

By default, *DBEst++* uses the frequency table (FT) *obtained from the samples* to scale up the predictions (see eq. (1)). When sample sizes are smaller, as shown in Figures 17 and 18, *DBEst++* has a higher error, largely caused by the inaccuracy of estimating the frequency table from the samples. So we set out to see the effect of this scaling up error, computing the exact frequency table statistics - which only requires one COUNT/Group BY query beforehand to compute. Use the exact frequency table to scale up the results reported by the *DBEst++* AQP engine, (marked as *DBEst++* with FT) are shown in Figures 17 and 18. Clearly, even for the small sample of size 2.5m, *DBEst++* with FT achieves smaller relative error (5.19%) than DeepDB(5.47%), or that of *DBEst++* with estimated frequency tables (11.90%) for COUNT queries. This is even better than DeepDB trained over a 30m sample. As the sample size increases, the relative error of *DBEst++* with FT decreases slightly.

Overall, sample-based AQP solutions like VerdictDB do not deal with large groups accurately. DeepDB achieves much better accuracy than VerdictDB. However, increasing the sample size does not decrease the overall error. *DBEst++*, as a model-based AQP approach, provides the most accurate answer with its error improved with larger samples. And also provide the smallest error even with small samples with exact frequency statistics.

4.6 Updatability

In this subsection, we conduct experiments to study how well *DBEst++* performs when unseen-previously data is inserted into the database. We use a 100-million-row store_sales table from the TPC-DS dataset. The setup we use is similar to that used by DeepDB, which showed how well it handles such updates (only for COUNT queries).

Our experiments are conducted as follows: We split the 100-million-row table into two partitions, P1 and P2: P1 has 90% of the table (90m rows) as the original data from which the *DBEst++* model will be created. P2 has 10% of the original table (10m rows). P2 will be used as a pool from which to derive the new previously-unseen data items to be inserted into the DB. As before, the *DBEst++* models will be trained on a small sample (5m rows) of the original data (90m rows). And updates/insertions will be 'streaming'into the system in batches. We generate 19 such batches. Each batch contains 50k rows sampled (without-replacement) from partition P2. We track the accuracy of *DBEst++* estimations after each batch.

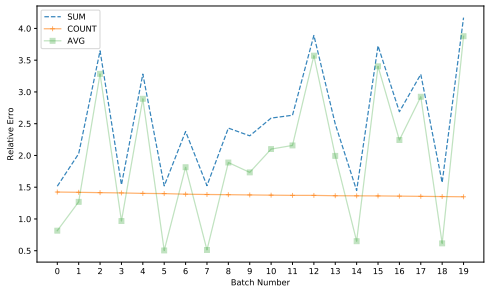

Figure 19: Relative Error When FTs Are Updated Only.

Figure 19 illustrates the mean relative error if *only the FTs are updated*. At batch 0, we create the *DBEst++* model from a 5m sample from P1. At batch 1, a new 50k batch arrives (a 50k sample without replacement from P2). At this point we update the frequency tables of *DBEst++* and then re-calculate the error of the same queries. This repeats for every new batch until batch 19. For `COUNT` queries, the curve is a constant. This implies that the MDN density estimator generalizes nicely and can predict cardinalities without performance degradation. Also, despite the unpredictable behavior on new batches, for many batches $(1, 3, 5, 6, 7, \dots)$ the model can make accurate predictions for `SUM` and `AVG` queries with small performance degradation. With the first experiment, the only thing that we capture from new data is the change of cardinality. Due to the generalization of the model, *DBEst++* enjoys good accuracy.

In the second experiment, we *update the frequency tables and the MDN models*. Figure 20 shows the errors for this case. At batch 0 we train *DBEst++*. At batch 1, a new batch of data arrives and *we update FTs and also update the MDN models*. Updating the MDN models in this case means that we maintain the weights of the previously-built models at batch 0 (copying them to a new MDN network) and feed-forward each of the new items in the 50k batch, which updates the overall MDN weights to account for the new items. This repeats for every new batch of data until batch 19. As shown in the figure, the relative error gradually converges to another state (after 12 updates). This phenomenon is due to a well known in incremental learning, known as "catastrophic forgetting".

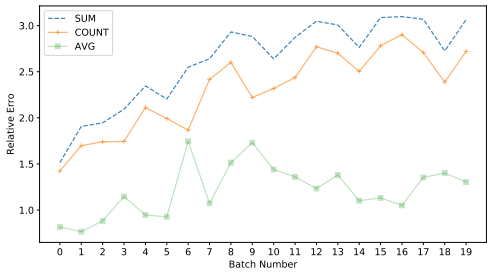

Figure 20: Relative Error When FTs and MDNs Are Updated.

To rectify this increased error we turn to using a smaller learning rate when we feed-forward the new items through the MDNs. Figure 21 shows the results for this approach. Specifically, here we set *LR_new* to *LR_base*/100 where *LR_base* is the learning rate we used to train the original model. (We chose to decrease it by a factor of 100 as each batch size is 1/100 of the sample size with which we trained the original models.) This figure illustrates the ability of *DBEst++* to deal with such data updates. It even shows that accuracy improves with time. This can be intuitively explained as the model sees increasingly more of the data after each batch.

In the following table, Table 1 we depict the time costs associated with updating the *DBEst++* models for the above experiments. As we can see, *DBEst++* can maintain high accuracy even when faced with fairly large batches of new data insertions, while the overhead to maintain such high accuracy is very small.

Table 1: Training time for updating the models to account for a new batch of 50k unseen tuples

	Method 1	Method 2	Method 2: Smaller lr
Training time (s)	2	44	46

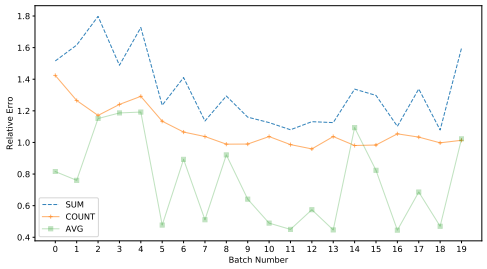

Figure 21: Fine-tuning Models With a Smaller Learning Rate

4.7 Parallel Inference

As mentioned, *DBEst++* is embarrassingly parallelizable. This is achieved by dividing the search space (e.g., the number of values of a categorical/group attribute) into the number of available threads. Currently, DeepDB does not support parallel inferencing and it is not clear how to do this. We run 10 `GROUP BY` queries with more than 50,000 groups, each from the TPC-DS data.

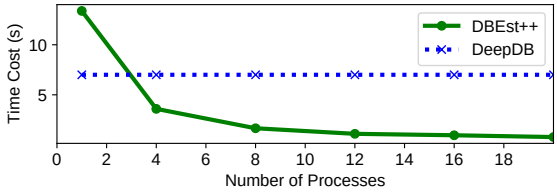

Figure 22: Query Response Time Reduction with Varying Degrees of Parallelism

Figure 22 shows that query response time decreases linearly as the degree of parallelism increases. For instance, it takes 13.38s for *DBEst++* to process such a query. If 20 cores are used, query response time drops to 0.78s.

5 RELATED WORK

Many research projects are ongoing to enhance the functionalities of, or replace, RDBMSs by ML models. ML models are widely used for approximate query processing [21, 29, 42], workload forecasting [28], and database tuning [44, 46]. SageDB [25] aims to replace all components in RDBMSs by ML models. Models could also be used for exact query processing. For instance, the learned index [26] predicts approximate locations for tuples, and adjacent pages are also fetched during query processing.

With respect to AQP, there exist many well-established approaches, including online aggregation [19], data sketches [10] and sample-based approaches [2, 34].The state-of-the-art AQP was up to very

fairly recently dominated by sampling-based approaches. STRAT [8] creates stratified samples to answer GROUP BY and HAVING queries. BlinkDB [2] maintains uniform and stratified samples, and a sample of specific size is automatically selected for fast query processing with bounded errors. To create samples, STRAT considers combinations of all column pairs, BlinkDB makes samples over popular column sets, while VerdictDB [34] relies on the user to provide such information. To guarantee high accuracy, large samples are usually generated and maintained. This potentially increases the space overheads, and the corresponding query response time is still high.

More recently learned approaches for AQP emerged. Such efforts include DBEst [29], DeepDB [21], deep generative models [43], etc. and can achieve lower query response times and higher accuracy. With respect to space overheads, models tend to be smaller than samples. However, it is not always the case. For instance, DBEst maintains a density estimator and a regressor for each group of a grouping attribute. If the number of groups is very large, the size of models can explode, limiting its appeal for many queries.

6 FUTURE WORK

Our current plans for future work include the following. First, support for updating ML models for AQP need further investigation. Currently, only insertion operations has been supported and tested. Support deletion operations (in addition to insertions) and without forgetting the past DB state creates interesting and difficult challenges, as neural networks have not been designed for such contexts.

Additionally, it is worth pursuing more elaborate approaches for avoiding catastrophic forgetting, while enabling learning from new data. This is expected to be a formidable undertaking when new data comes from significantly different underlying distributions. Methods for concept drift detection may come in handy for this task. Similarly for methods for fusing old and new models.

An interesting challenge comes from investigate what error guarantees can be provided for learned AQP engines. As *DBEst++* rests on regression as its underpinning driver, prediction intervals as they have been employed within various regression models may prove beneficial.

Finally, our experience is that many learned approaches struggle with large-cardinality categorical attributes. The latter tend to adversely impact the sizes of models and/or their accuracy and training times. Hence, more research is warranted to address these challenges.

7 CONCLUSION

In this paper, we have argued for light learned AQP engines and introduced the *DBEst++* AQP engine. At its essence, *DBEst++* is underpinned by a regression-inspired approach to estimating answers to analytical queries, continuing in this respect the insights of the original DBEst engine [29]. The new engine, *DBEst++* puts forth a novel learned AQP architecture comprised of models for (i) word embedding, (ii) density estimation using MDNs, and (iii) regression-based prediction for aggregation function/variable values using again MDNs. Experimental results show that *DBEst++* achieves higher accuracy and shorter response time performance than the current state of the art sampling-based and learned approaches. At the same time it comes with dramatically reduced memory footprints. Our results also show that *DBEst++* maintains high accuracy even under settings where the underlying datasets are changing with time. Likewise, *DBEst++* shows the best performance for challenging cases, such as high-cardinality categorical attributes. We hope this contribution will spark a discussion in our community and a trend towards light AQP learned DB functionality and their integration into DB engines.

REFERENCES

[1] Jean-Marc Adamo. 2001. *Data mining for association rules and sequential patterns: sequential and parallel algorithms.* Springer Science & Business Media.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[3] Christos Anagnostopoulos and Peter Triantafillou. 2014. Scaling out Big Data Missing Value Imputations: Pythia vs. Godzilla. , 651–660 pages.

[4] Christos Anagnostopoulos and Peter Triantafillou. 2015. Learning Set Cardinality in Distance Nearest Neighbours. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM) (ICDM '15)*. 691–696.

[5] Christos Anagnostopoulos and Peter Triantafillou. 2015. Learning to Accurately COUNT with Query-Driven Predictive Analytics.

[6] C. Anagnostopoulos and P. Triantafillou. 2017. Efficient Scalable Accurate Regression Queries in In-DBMS Analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 559–570.

[7] Christos Anagnostopoulos and Peter Triantafillou. 2017. Query-Driven Learning for Predictive Analytics of Data Subspace Cardinality. *ACM Trans. Knowl. Discov. Data* 11, 4 (2017), 47:1–47:46.

[8] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 9–es.

[9] Ronan Collobert. 2014. Word embeddings through hellinger pca. In *in Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*. Citeseer.

[10] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[11] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1555–1569.

[12] Charles Flock and Joe Stampf. 2009. XLeratorDB. https://westclintech.com/

[13] Sylvia Frühwirth-Schnatter. 2006. *Finite mixture and Markov switching models*. Springer Science & Business Media.

[14] Amir Globerson, Gal Chechik, Fernando Pereira, and Naftali Tishby. 2007. Euclidean embedding of co-occurrence data. *Journal of Machine Learning Research* 8, Oct (2007), 2265–2295.

[15] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. 2013. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211* (2013).

[16] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

[17] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476.

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

[19] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 171–182.

[20] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711.

[21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[22] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016*

International Conference on Management of Data. 631–646.

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[24] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[25] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. (2019).

[26] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[27] Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*. 2177–2185.

[28] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[29] Qingzhi Ma and Peter Triantafillou. 2019. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*. 1553–1570.

[30] Qingzhi Ma and Peter Triantafillou. 2020. Query-Centric Regression for In-DBMS Analytics.. In *DOLAP*. 16–25.

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[32] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS.. In *VLDB*, Vol. 6. 1049–1058.

[33] Aniruddh Nath and Pedro M Domingos. 2015. Learning relational sum-product networks. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[34] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[35] Anthony Robins. 1995. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science* 7, 2 (1995), 123–146.

[36] Lior Rokach and Oded Z Maimon. 2008. *Data mining with decision trees: theory and applications*. Vol. 69. World scientific.

[37] Fotis Savva, Christos Anagnostopoulos, and Peter Triantafillou. 2019. Explaining Aggregates for Exploratory Analytics. , 478–487 pages.

[38] F. Savva, C. Anagnostopoulos, and P. Triantafillou. 2020. SuRF: Identification of Interesting Data Regions with Surrogate Models. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1321–1332.

[39] Fotis Savva, Christos Anagnostopoulos, Peter Triantafillou, and Kostas Kolomvatsos. 2020. Large-scale Data Exploration using Explanatory Regression Functions. *ACM Transactions on Knowledge Discovery from Data* (08 2020).

[40] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560* (2019).

[41] Siri Team. 2017. Deep learning for Siri's voice: on-device deep mixture density networks for hybrid unit selection synthesis. *Apple Machine Learning J* 1, 4 (2017).

[42] Arvind Thiagarajan and Samuel Madden. 2008. Querying continuous functions in a database system. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 791–804.

[43] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1309–1320.

[44] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1009–1024.

[45] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[46] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.