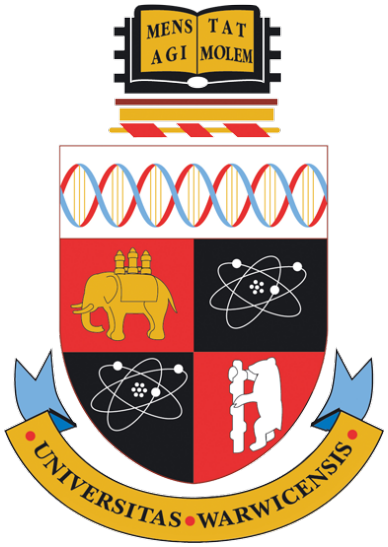
WARWICK
THE UNIVERSITY OF WARWICK



Approximate Query Processing Using Machine Learning

by

Qingzhi Ma

Thesis

Submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

June 2021

Qingzhi Ma

Q.Ma.2@warwick.ac.uk

ORCID ID:0000-0003-2418-090X

Doctoral Committee:

Professor Peter Triantafillou

Professor Hakan Ferhatosmanoglu

Assistant/Associate Professor Evangelia Kalyvianaki

.

Abstract

In the era of big data, the volume of collected data grows faster than the growth of computational power. And it becomes prohibitively expensive to compute the exact answers to analytical queries. This greatly increases the value of approaches that can compute efficiently approximate, but highly accurate, answers to analytical queries. Approximate query processing (AQP) aims to reduce the query latency and memory footprints at the cost of small quality losses. Previous efforts on AQP largely rely on samples or sketches, etc. However, trade-offs between query response time (or memory footprint) and accuracy are unavoidable. Specifically, to guarantee higher accuracy, a large sample is usually generated and maintained, which leads to increased query response time and space overheads.

In this thesis, we aim to overcome the drawbacks of current AQP solutions by applying machine learning models. Instead of accessing data (or samples of it), models are used to make predictions. Our model-based AQP solutions are developed and improved in three stages, and are described as follows:

1. We firstly investigate potential regression models for AQP and propose the query-centric regression, coined QReg. QReg is an ensemble method based on regression models. It achieves better accuracy than the state-of-the-art regression models and overcomes the generalization-overfit dilemma when employing machine learning models within DBMSs.

2. We introduce the first AQP engine DBEst based on classical machine learning models. Specifically, regression models and density estimators are trained over the data/samples, and are further combined to produce the final approximate answers.

3. We further improve DBEst by replacing classical machine learning models with deep learning networks and word embedding. This overcomes the drawbacks of queries with large groups, and query response time and space overheads are further reduced.

We conduct experiments against the state-of-the-art AQP engines over various datasets, and show that our method achieves better accuracy while offering orders of magnitude savings in space overheads and query response time.

Contents

List of Tables

List of Figures

Acknowledgments

I would first thank my Ph.D. supervisor Prof. Peter Triantafillou. I was so lucky to meet him during my master's study at Glasgow. Attracted by his personal charm and research interest, I moved to the University of Warwick and became a Ph.D. student under his guidance. Peter is one of the nicest persons I have ever met. He deeply cared about my Ph.D. study and daily life. He would always give me very good suggestions when I have difficulties, and kept encouraging and supporting me. Also, Peter is excellent at writing academic papers in a concise and professional way. I learned a lot of writing and presentation skills from him. I am so glad to become his first Ph.D. student at the University of Warwick. Now I finish my Ph.D. study, and I know the importance of having a good supervisor, and I am grateful to be supervised by Prof. Peter Triantafillou.

I am sincerely grateful to my thesis committee members. Both Hakan Ferhatosmanoglu and Evangelia Kalyvianaki are so nice to serve on my thesis committee, and they are supportive. I knew Hakan from the course Advanced Databases, and I got a better understanding of big data techniques from him.

I would also thank my colleagues in the LEADS team. Mehrdad Almasi and Meghdad Kurmanji helped a lot in conducting the experiments. Mohammadi Shanghooshabad shared some good advice to improve the performance of *DBEst++*.

My Ph.D. study was also greatly influenced by the data science theme members at the University of Warwick. Graham Cormode gave me valuable advice for improving QReg and DBEst. I also had enjoyable discussions with Michael Shekelyan, and we worked together to solve problems involving joins. I hope I could have more collaborations with the people at the data science

division.

Last, but most importantly, I deeply thank my parents and wife. I received endless care and support from them. During the Covid pandemic, life was not easy. I was so lucky to have my wife with me, and we managed to make it through. My wife is also a Ph.D. student, I wish she would graduate from Kings College London successfully.

Declarations

1 Publications

This thesis is submitted to the University of Warwick in support of my application and is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous applications for any degree. The work in this thesis has been undertaken by myself under the supervision of Prof. Peter Triantafillou. Parts of this thesis have been previously published by the author in the following:

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

[105] Qingzhi Ma and Peter Triantafillou. Query-centric regression for in-dbms analytics. In *22nd International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, pages 16–25, 2020

[106] Qingzhi Ma and Peter Triantafillou. Query-centric regression. *Information Systems*, page 101736, 2021

[107] Qingzhi Ma, Ali Mohhamedi Shanghooshabad, Meghdad Kurmanji, Mehrdad Almasi, and Peter Triantafillou. Learned approximate query processing: Make it light, accurate and fast. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021

Research was performed in collaboration during the development of this thesis, but does not form part of the thesis:

[141] Ali M Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. Pgmjoins: Random join sampling with graphical models. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021

[143] Michael Shekelyan, Graham Cormode, Qingzhi Ma, Ali M Shanghooshabad, and Peter Triantafillou. Weighted random sampling over joins. In *(submitted)*, 2021

2 Sponsorships and Grants

This research presented in this thesis was supported in part by the following benefactors and sources:

- The 'Tools, Practices and Systems' theme of the UK Research and Innovation (UKRI) Strategic Priorities Fund (EPSRC Grant EP/T001569/1)
- The Alan Turing Institute (EPSRC grant EP/N510129/1).

Acronyms

AF Aggregate Function.

AQP Approximate Query Processing.

CCPP Combined Cycle Power Plant.

CP Column Pair.

DBMS Database Management System.

DTR Decision Tree Regression.

FT Frequency Table.

GBoost Gradient Boosting.

IoT Internet of Thing.

KDE Kernel Density Estimation.

KPI Key Performance Indicator.

LR Linear Regression.

MB MegaByte.

MDN Mixture Density Network.

ML Machine Learning.

NNR Nearest Neighbours Regression.

NRMSE Normalized Root Mean Square Error.

OL Opportunity Loss.

OLAP Online Analytical Processing.

OLTP Online Transaction Processing.

PA Power Analysis.

PLR Piecewise Linear Regression.

PR Polynomial Regression.

RDBMS Relational Database Management System.

RM Regression Model.

ROL Relative Opportunity Loss.

RSPN Relational Sum-Product Networks.

SML Statistical Machine Learning.

SQL Structured Query Language.

SSD Solid-State Drive.

SVR SVM Regression.

UDF User-Defined Function.

WE Word Embedding.

Symbols

ϵ	estimation error
x	the attribute in the range predicate
y	the attribute to be aggregated
n	the size of a given table
w	the weight
T	Table T
S_T	A sample from table T
$\|T\|$	the size of Table T
$R(x)$	a regression model of x $y = R(x)$
$D(x)$	a density estimator over column x
μ	the mean in a normal distribution
σ^2	the variance in a normal distribution
$\mathcal{N}(\mu, \sigma^2)$	a normal distribution

Chapter 1

Introduction

1.1 Data Analytical Tasks and Machine Learning

In the era of big data, a huge amount of data is generated and collected. By the end of 2020, there are 2.8 billion active Facebook users, and more than 4 petabytes of data are generated per day [54]. Take YouTube as another example. The first video uploaded to YouTube was in 2005. By 2021, there are 2.3 billion users, and more than 500 hours of fresh video is uploaded per minute. Furthermore, the growth of data volume is speeding up as increased mobile devices, high-frequency sensors, and IoTs are being widely used.

As a consequence, the ability to answer queries over increased data is being challenged in various aspects: (a.) The query response time grows prohibitively. For instance, it takes several minutes to execute a query over the Conviva data (of size 7.5TB) by HIVE [6]. (b.) The throughput of queries is increasing continuously. Take Google as an example, 1.2 Trillion queries are being executed per day. (c.) More storage and memory resources are needed to answer queries over increased data.

Even with the state-of-the-art big data frameworks and infrastructures, the said problems still exist. Typically, data is stored in a distributed file system (like the Hadoop File System [145]), and each node is only responsible for a proportion of the final answer [45, 170]. By horizontal scaling, an analyst is able to reduce the query response time by a factor of 10 at the cost of at least

10 times the investment in computational resources. Although the reduction in query response time is linear to the number of computational resources, such a strategy is uneconomic/unrealistic as monetary investment typically grows slower than the growth of data [121]. In addition, it is not always necessary to get the exact query results. For many analytical tasks, people are willing to accept a fast approximate answer at the cost of minor quality loss. For KPI analytical tasks, an analyst is more interested in the distribution of a specific performance indicator, as long as the approximate distribution won't affect the final conclusion.

This motivates us that instead of providing exact answers at the cost of an unacceptably high response time, we could provide approximate answers by reducing/avoiding the amount of data processed during query execution. *Approximate Query Processing (AQP) aims to reduce the query response time by providing approximate answers.* Efforts in AQP largely fall into four categories: online aggregation [26, 37, 75, 120, 134], data sketches [40, 41], sample-based approaches [2, 3, 6, 29, 63, 65, 87, 122, 122, 123, 129] and model-based approaches [77, 104, 106, 107]. Prior to the research addressed in this thesis, there was no model-based AQP approach, and AQP was dominated by sample-based approaches.

Figure 1.1 shows the general system architecture of sample-based AQP engines. Samples are generated and maintained in the system. During query execution, instead of accessing tables in the back-end server, samples are used to answer the query. Typically, the sizes of the samples are much smaller than the tables. Thus, it takes less time to answer queries using samples.

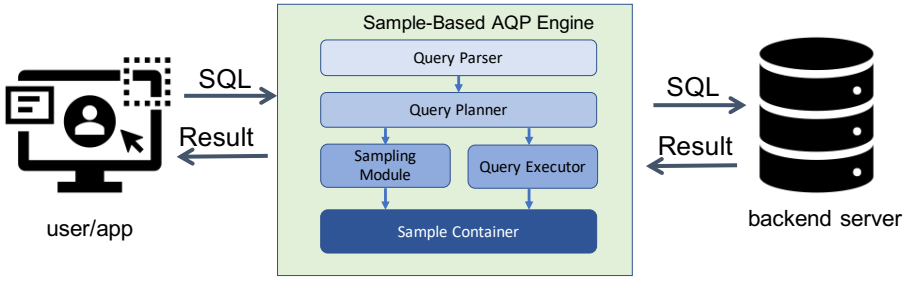


Figure 1.1: Architecture of Sample-Based AQP Systems.

However, the sample size has a direct impact on the quality of the approximate answer. A smaller sample leads to lower query response time, but the error tends to be higher. On the other hand, if we increase the sample size in favor of accuracy, query response time grows, and more space investments are needed. This means there is a trade-off between query accuracy and query response time/space overheads. For online aggregation, the user stops the query execution process as long as he is satisfied with the intermediate query result. For some AQP engines, like VerdictDB and BlinkDB, the sample size could be automatically allocated within a given threshold. However, the dilemma between query accuracy and query response time/space overheads still exists.

In this thesis, we show that we can provide high-quality approximate answers with much lower response times and much lower space overheads by applying machine learning (ML) models. Instead of using samples, ML models are used to provide approximate answers. Compared to samples, models take less time to respond, and the sizes of models are usually orders of magnitude smaller. In this way, model-based approaches enjoy benefits in all aspects.

We achieve this target in three steps. Firstly, we *exploit the existing classical machine learning models.* We examine and investigate potential models that could be applied for AQP tasks. Specifically, regression models and density estimators are carefully evaluated and analyzed. And we propose *QReg*, a query-centric regression, which fits the context of query answering in database systems. Secondly, we introduce *our first AQP engine, coined DBEst, which relies on regression models and density estimators.* Lastly, we *replace classical machine learning models with deep learning networks and other techniques, and create an improved AQP engine, called DBEst++.*

1.2 Thesis Outline

This thesis is divided into two parts. In the first part, we overview recent research on approximate query processing and machine learning models. In the second part, we present the three steps that we used to address the issues

for approximate query processing by machine learning models. The outline of this thesis is also summarized in Table 1.1.

Table 1.1: Thesis outline

Part	Introduction	Chapter
Exploiting Related Work	**ML Models**: We review potential machine learning models that could be used for AQP **AQP techniques**: State-of-the-art AQP techniques are listed and summarized.	2
AQP by ML	**Query-Centric Regression**: We firstly propose a query-centric regression that best fits the context of analytical tasks for databases.	3
	AQP engine DBEst: The first model-based AQP engine is built over classical machine learning models, including regressions and density estimators.	4
	AQP engine *DBEst++*: We further improve DBEst by applying Mixture Density Networks and word embedding.	5

Chapter 2 overviews the background and summarizes the related work. Specifically, the background is divided into two parts: (1.) overview of classical machine learning models, and (2) summary of related work on AQP efforts. Chapter 3 proposes a query-centric regression, coined $QReg$, which is an ensemble method based on regression models. $QReg$ achieves better accuracy than state-of-the-art regressions. Chapter 4 introduces the first model-based AQP engine, DBEst. Classical machine learning models, including regressions and density estimators are used to provide approximate answers for queries. Chapter 5 improves the work in Chapter 4 by applying Mixture Density Networks and word embedding. This overcomes the drawbacks of $DBEst$ for `GROUP BY` queries and further improves the performance. Finally, Chapter 6 introduces future work and concludes.

Note, the outcomes of this thesis have also been presented at Computer Science conferences and workshops, including ACM SIGMOD 2019 [104], 22nd International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data [105], the Conference on Innovative Data Systems Research [107] and Information Systems [106].

Here, we will give a detailed introduction to the problems addressed in each

chapter.

1.2.1 Introduction to query-centric regression

As data analytics is becoming increasingly important in the big data era, a great impetus is emerging for incorporating machine learning (ML) models within DBMS analytics engines. Within the data management community several efforts are underway to augment traditional DB functionality with data analytics tools and models [24, 35, 38, 76, 78, 80, 110, 110, 131, 165]. These efforts assume various forms. One prominent class pertains to connectors to back-end databases, which allow for statistical analyses and related queries on DB data, like MonetDB.R [113], SciDB-Py [64], and Psycopg [161]. Such connectors allow the analysis of data using known languages while avoiding costly data transfers from and to the database. Another class of efforts concerns learning from past answers to predict the answers to future analytical queries, e.g. for approximate query processing engines, which provide approximate answers to aggregate queries, using ML techniques [11–13, 122], or for tuning database systems [7], and for forecasting workloads [103]. Yet another class of efforts concerns model and query-prediction serving, like the Velox/Clipper systems [43, 44] managing ML models for predictive analytics. Finally, vision papers suggest the move towards *model selection management systems* [97], where a primary task is *model selection* whereby the system is able to select the best model to use for the task at hand.

In this realm, regression models, being a principal, powerful means for predictive analytics, are of particular interest to both analysts and data analytics platforms. An increasing number of major database vendors include in their products data mining and machine learning analytic tools. RMs are playing an increasingly important role within data systems. Examples of their extended use and significance include many modern DBs which provide support for regression, such as XLeratorDB [15] for Microsoft SQL Server, Oracle UTL_NLA [1, 168], IBM Intelligent Miner [156], which provide SQL interfaces for analysts to specify regression tasks. Academic efforts include MADLib (over Postgr-

eSQL) [76], MAD [35], and MauveDB [47], which integrates regression models into a RDBMS. [118, 119] uses User-Defined Functions (UDFs) to compute statistical machine learning models and data summarization. FunctionDB [152] builds regression models so that future queries can be evaluated using the models. Furthermore, [139] integrates and supports least squares regression models over training data sets defined by join queries on database tables. Finally, the DBEst and DBEst++ approximate query processing engines [104, 107], which will be introduced in this thesis, rely on RMs to provide highly accurate approximate answers for popular queries. Thus, in general, RMs are useful for query processing and data analytics tasks. In addition, RMs are helpful for many other key tasks: imputing missing values, testing hypotheses, data generation, fast visualization, etc.

But, given the large number of available RMs, how can one know which is the best regression model to use? Is there a model that outperforms the others for different data sets? Even within a single data set, is there a model that outperforms the others across all data subsets?

In this thesis, we aim to answer these questions. Chapter 3 tackles these issues using extensive experimental analyses, which show that different models win across different data sets and even across different subsets of the same data set. This is established to hold across many well-known base regression models and more sophisticated regression ensemble models and across many real-world datasets. Given this fact, Chapter 3 will also show that one can construct a new regression model which will comprise existing regression models. It is designed to always select the best constituent regression model $R_k(x)$ for a specific query for x_j to deploy, with respect to minimizing the estimation errors $\epsilon\ (i, x_j)$.

1.2.2 Introduction to AQP and Supported Queries

We live in the era of big data, whose timely, accurate, and inexpensive analysis bears great opportunities and benefits which permeate practically all facets of our lives. Analytical queries in this realm typically rely on two fundamental

components. Firstly, selection operators (such as range predicates, equality and IN conditions) help focus on specific data regions. Secondly, aggregation functions (such as AVG, SUM, COUNT, PERCENTILE, VARIANCE) are applied on the selected data regions to provide key insights. In SQL, a core component of a large class of analytical queries takes the form (and this is the type of queries the thesis aims to support):

```
SELECT  [g1, ...,] AF(y) FROM T
WHERE x1 BETWEEN lb AND ub
[AND x2='a1']
[AND x3 IN ('a2', 'a3', ...)]
[GROUP BY g1, ...]
```

where range predicates on attributes ($x1$) are used to define a data region within that of (a csv file or) table T, and an aggregation function AF is used on attribute y. A close look at many real-world data sets and analytical workloads reveals that certain types of data attributes play a key role. Obviously, AFs operate on numerical attributes. Additionally, selection operators often operate on numerical attributes as well, or equivalently on ordinal categorical attributes, such as dates, time, location, etc. Examples abound: Sensor and IoT datasets are a significant contributor to the big data phenomenon. Smart city analytical queries involve ranges on time, location, wind speed, air pressure e.g., to analyze pollution (e.g., PM2.5, CO2 levels, etc. [100]). Smart home analytics involve measurements (temperature, humidity, etc.) to analyze home power consumption. Power plants in operation, engineering plants, scientific applications (from astronomy to bio-medical applications) are awash with such data and analytics needs.

Such queries are fundamental to exploratory analytics, where primarily analysts wish to understand the datasets by exploring various data subspaces (defined using ranges) and deriving descriptive statistics information (using AFs) about said subspaces. Within data warehouses/databases, the above query type may be augmented with `GROUP BY` operators, whereby the AF is

performed separately for each value of the group attribute. Finally, queries may involve more than one table, requiring their join and performing the above analyses on the join-result table.

Unfortunately, the timely, accurate, and inexpensive analysis of big data presents formidable challenges. Traditional solutions do not scale well, suffer from long response times, and/or require large money investments to deploy them on top of big data analytics stacks (e.g., [45, 162, 171]). To address these challenges, AQP strives for approximate-but-accurate-enough answers which can be delivered swiftly. AQP has been studied for over two decades now, and significant progress has been made. Nonetheless, as data continue to grow in size, AQP engines struggle to keep up.

1.3 Thesis Contributions

Motivated and inspired by the problems mentioned in Section 1.2, this thesis focuses on approximate query processing by machine learning models. There are three major efforts in this thesis. Here, we only summarize the major contributions, and detailed contributions are presented in Chapter 3-5.

- We propose a query-centric regression, coined *QReg*. It is an ensemble method based on regression models. One of the major findings is that best practice, which suggests to an analyst to use a top-performing ensemble, is misleading and leads to significant errors for large numbers of queries. In several cases, despite the fact that different RMs had a very similar overall error (NRMSE), a significant fraction of queries face very large differences in error when using seemingly-similarly-performing RMs. Thus, both sophisticated and simpler RMs cannot cope well, in order to appease query-sensitive scenarios, where query distributions may target specific data subspaces.

- We introduce the first model-based AQP engine *DBEst*. The overriding guiding principle is to develop and study a model-driven solution, instead of a data-driven solution, where queries are answered by models of data

and not the data itself (or samples of it). By using regression and density estimators, *DBEst* requires small overheads and shorter response times, (even with just a single thread). Thus, *DBEst* renders analytics less costly and achieves much higher system throughput.

- We contribute another novel AQP engine, coined *DBEst++*, which extends our previous effort *DBEst*, 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.

Chapter 2

Background and Related Work

In this chapter, we briefly review modern techniques in approximate query processing and potential machine learning models that could be applied in AQP. As regression models are fundamental in machine learning, we firstly review regression models in Section 2.1. Classical regression models consist of simple regression models and ensemble methods. Ensemble methods are based on simple regression models, and are designed to reduce prediction variance and/or bias. A brief comparison of regression models is summarized here, while a more detailed comparison of regression models is made in Section 3.3.3. In the second section, we firstly present modern techniques and infrastructures in exact query processing. After that, we overview AQP methods and engines. Furthermore, as the majority of AQP solutions are based on samples, we summarize sampling techniques used for approximate query processing.

2.1 Machine Learning Models for AQP

For approximate query processing tasks, we first review modern regressions. Our study employs a set of representative and popular regression models (RMs), grouped into two categories: Simple and ensemble RMs.

2.1.1 Simple Regression Models

Simple RMs include linear regression (LR), polynomial regression (PR) [58], decision tree regression (DTR) [108], SVM Regression (SVR) [111], Nearest Neighbours Regression (NNR) [9]. An introduction to these simple regression models will be presented in the following sub sections. Table 2.1 summarizes the known asymptotic time complexity for training for key regression models. And more detailed comparisons are made and discussed in Section 3.3.3.

Table 2.1: Complexity of typical regression models

LR	PR	DTR	NNR	SVR	Gaussian Process
$\mathcal{O}(d^2n)$	$\mathcal{O}(d^4n)$	[$\mathcal{O}(dnlog(n))$, $\mathcal{O}(n^2d)$]	$\mathcal{O}(k.log(n))$ or $\mathcal{O}(ndk)$	$\mathcal{O}(vn^2)$	$\mathcal{O}(n^3)$

Note: d is the dimensionality of the data points, n is the number of points in the training data, k is the number of neighbors for KNN regression, v is the number of support vectors for SVM regression.

Linear and Polynomial Regression

Given a data set $\{\mathbf{x}_i, y_i\}_{i=1}^n$, the relationship between label y and feature $\mathbf{x}$ is assumed to be linear, taking the form $y_i = w_0 + w_1 x_{i1} + ... + w_p x_{ip} + \epsilon_i = \sum_{j=1}^p w_j x_{ij} + \epsilon_i$. Linear regression is simple, efficient to train and very popular.

Polynomial regression [58] is similar to linear regression. Uni-variate polynomial regression takes the form $y_i = w_0 + w_1 x_i + ... + w_p x_i^p + \epsilon_i = \sum_{j=1}^p w_j x_i^j + \epsilon_i$ Multi-variate polynomial regression further includes cross terms. Polynomial regression may be prone to overfitting and not feasible in high dimensional space due to its complexity.

Decision Tree Regression

Decision Trees [108] are a non-parametric supervised learning method used for classification and regression. DTR builds simple decision rules from data features. Internal nodes m define test functions (decisions) on features, with all data items satisfying all conditions on the path to m (denoted $\mathbb{D}_m$) will reach m in the decision process. Node m is typically associated with the mean (or median) of all such data items in $\mathbb{D}_m$. The decision process leads to leaves,

which store the predicted value. DTR is simple and takes less time to train for very large data sets.

SVM Regression

SVR [111] is also a supervised model based on SVMs, used for classification and regression. Labelled (training) data points are viewed as points in high-dimensional vector space and SVMs define a hyperplane that maximally separates the data points in that space using their labels. New data then are assigned to the space occupied by points with the same label. SVR uses the parameters of the hyperplane to derive predictions for new data points. SVR is very effective for high dimensional spaces, but inefficient as the number of training points increases.

SVR affords the flexibility to employ different kernel functions, e.g., linear and RBF kernel functions are popular with interesting time-accuracy trade-offs. SVR is effective for high dimensional spaces but less desirable for very large data sets.

Nearest Neighbours Regression

NNR [9] is based on defining appropriate distances between data points. For a given query, kNNR finds the k nearest points to the query, and the average of the k neighbors constitutes the predicted value. kNN regression is simple, robust to noise, and very effective for very large data sets.

2.1.2 Ensemble Methods

Ensemble learning is a machine learning paradigm where multiple models (or "weak learners") are trained to solve the same problem and combined to get better results. It is often observed that prediction accuracy is improved by combining the prediction results in some way (e.g., using weighted averaging of predictions from various base models) [20]. Ensemble learning is useful for scaling-up data mining and model prediction [146]. There have been many well-developed ensemble methods, including bagging (bootstrap aggregating)

[21], boosting [59, 60], stacking [167], mixture of experts [83], etc. Bagging tends to make predictions with less variance while boosting and stacking try to produce strong models with less bias.

Popular ensemble methods include AdaBoost, gradient boosting, XGBoost, etc. AdaBoost [59], short for"adaptive boosting", is a popular boosting algorithm. Unlike bootstrap aggregating whose models are trained in parallel, the prediction models in AdaBoost are trained in sequence. AdaBoost was firstly proposed to solve classification problems, and was applied to solve regression problems later on.

Gradient boosting (GBoost)'s objective is to minimize the loss function of the following form:

$$L(y_i, f(x_i)) = MSE = \sum (y_i - f(x_i))^2 \tag{2.1}$$

And the predictions are updated in the direction of gradient descent, which is

$$f(x_i)^{r+1} = f(x_i)^r + \alpha * \frac{\partial L(y_i, f(x_i)^r)}{\partial f(x_i)^r} \tag{2.2}$$

where r is the iteration number. GBoost usually uses only the first-order information; Chen et al. incorporate the second-order information in gradient boosting for conditional random fields, and improve its accuracy [31]. However, the base models are usually limited to a set of classification and regression trees (CART). Other regression models are not supported.

XGBoost [30] is a state-of-art boosting method, and is widely used for competitions due to its fast training time and high accuracy. The objective of XGBoost is

$$obj(\Theta) = L(\Theta) + \Omega(\Theta) \tag{2.3}$$

where $L(\Theta)$ is the loss function, and controls how close predictions are to the targets. $\Omega(\Theta)$ is the regularization term, which controls the complexity of the model. Over-fitting is avoided if the proper $\Omega(\Theta)$ is selected. The base models (booster) can be gbtree, gblinear or dart [163]. Gbtree and dart are tree models

while gblinear is linear.

2.1.3 Application of Regression Models and Key Insight

As mentioned, traditional ML best-practice, recognizing the plethora of RMs and the need for models to generalize to different distributions that the one in a training data set have led to the development of ensemble methods. This is best exemplified by the ensembles used in various competitions and challenges, such as Kaggle, Netflix, and KDDCup [18, 49, 150]. For example, the large majority of top-performers used XGBoost alone or is an ensemble (KDDCup2015). However, looking closely at related results, a key insight is that the top-k best performers in these competitions are shown to have almost identical performance. Nonetheless, the differences among the top 5 were in the 3rd digit after the decimal! The community wisdom, thus, leads to a "best practice" that employs any one of such top-performing RMs and uses it for in-DBMS predictive analytics.

We will reveal that such "best practice" hides significant losses of opportunity from a data management, query-centric perspective and we will analyze the space and the performance of possible solutions.

2.2 Approximate Query Processing Techniques and Engines

In this section, we will review the modern techniques and engines for query processing. Firstly, we summarize the big data infrastructures for exact query answering. After that, we review state-of-the-art AQP engines and methods. As sampling is vital in major AQP engines, we review popular sampling techniques.

2.2.1 Introduction to Exact Query Processing

This work is primarily interested in the efficient and accurate processing of analytical queries in big data environments. Big data infrastructures, (e.g., SPARK [171], Hadoop [45], and TEZ [135]) and query processing engines over

them, such as Hive [154], Spark SQL [16], Impala [94], Amazon Redshift [72], and Shark [53] have been a catalyst for analytical query processing. Efficient analytical QP engines with columnar data representations have also been developed (e.g., MonetDB [79] or, for streaming environments, Trill [26], which can also run in distributed .NET environments, (Orleans [23]). A related thread concerns applying data pre-fetching techniques [81], including semantic windows [86], or developing caches for analytical query results e.g., Data Canopy [166].

Such query processing infrastructures usually rely on more advanced techniques to speed up query processing, including indexing, caching, pre-computation, etc.

Indexing— Database index is a data structure to speed up the retrieval operations of tuples within a database. An index structure typically consists of key-location pairs. Thus, an index helps avoid the costly linear scan of large tables/partitions among the clusters by accessing the tuples of interest directly. There are intensive research and efforts in database indexing, and we list the most recent ones. Helios is a distributed, highly-scalable system used at Microsoft. It utilizes indexing in the cloud for large streams [128]. Hyperspace, as another example, is an indexing subsystem for Apache Spark [144]. It adopts the index-on-the-lake concept, which supports any data format, including text CSV, JSON, Parquet, ORC. As the amount of data to be scanned is reduced, indexes are particularly efficient in providing tremendous acceleration for certain workloads. However, for aggregate queries involving a large amount of data, indexes might not work well.

Caching— A database cache reduces the pressure on the server by caching the frequently accessed queries or data. Caches might be integrated into databases, like Result Cache for Oracle [8], and PostgreSQL Query Cache [71]. It offers 10x-100x improvement in query performance. Caches could also be stored on dedicated servers outside of databases. Typically, such caches are built upon key/value NoSQL stores such as Redis [25] and Memcached [56]. They act as another layer on top of database systems, and are able to

process up to millions of requests per second. Such caching techniques enjoy low latency and high throughput, and are widely adopted in the industry. Although caches bring orders of improvement in query processing, it is only effective for repeated/cached queries. The backend server still needs to process the query if it is not seen before.

Pre-computation— Pre-computation techniques, like data cubes, are also used to boost query processing in databases [70, 73]. Data cubes are multiple-dimensional (n-D) array of values. The data cube is used to represent data along some measure of interest. In this study, we are interested in aggregate queries involving `COUNT, SUM, AVG`, etc. If the measure of interest is pre-computed in each cell of the data cube, the final query result could be produced using the data cube instead of accessing the whole data. COSMOS [134] is such an effort by applying multi-dimensional cubes for online aggregation tasks. The results of past queries are stored in the cubes, and are re-used if they fall in the ranges of new queries. The boundary cases are read from the backend server. Such an effort reduces query latency and improves throughput. However, this approach is limited to low-dimensional data due to the exponential explosion in the number of possible cubes.

2.2.2 Approximate Query Processing Techniques

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 [77, 104, 152], workload forecasting [103], and database tuning [160, 172]. SageDB [96] aims to replace all components in RDBMSs by ML models. Models could also be used for exact query processing. For instance, the learned index [95] predicts approximate locations for tuples, and adjacent pages are also fetched during query processing.

As exact answers for analytical queries can still require very long response times AQP engines become desirable.With respect to AQP research, the existing solution space is quite complex. Some approaches based on data sketches have received considerable attention [39–41]. Others focus on progressive/online

aggregation [26, 37, 75, 120, 134]. Nonetheless, AQP research has been largely dominated by sampling-based approaches [2, 3, 6, 29, 63, 65, 87, 122, 122, 123, 129]. A different perspective is to think of forgetting data. DBs with 'amnesia'[90] could be viewed as equivalent to sampling approaches in that forgotten items correspond to non-sampled items. It would be interesting to see how such an approach compares with state-of-the-art AQP engines.

Table 2.2: Online versus offline sampling

	Offline	Online
Assumption:	(Partially) Known Workload	No Assumptions
Speedup:	High	Low
Representatives:	BlinkDB [6], VerdictDB [123] DeepDB [77], *DBEst++*[104, 107]	QuickR [87]

State-of-the-art sampling-based AQP approaches are broadly divided into two categories and in general no single approach is a 'silver bullet'[149]. Techniques that rely on online sampling, create samples on the fly and use them to approximate answers. But, even the best such efforts (e.g., [87]) only deliver a ca. 2x speedup. The second category can bring much bigger speedups [2, 6, 29, 122, 123] exploiting the fact that often query workloads are (at least partially) predictable: one can know beforehand popular query templates, including the joined tables and join keys, attributes for range predicates and grouping, etc. STRAT [29] creates a stratified sample over the unions of columns that occur in the `GROUP BY` or `HAVING` clauses. It considers all combinations of the column pairs. BlinkDB [6] showed that such templates can be identified from a small prefix of a workload. VerdictDB [123] depends on users providing this information. Samples are created for predictable/popular tables and column sets offline and kept in memory and queries are processed over the samples reducing drastically execution times. Table 2.2 summarizes typical AQP engines and their sampling strategy.

Works on predictable queries with prebuilt, prior samples are closest to our methods. BlinkDB [6] relies on uniform and stratified sampling and can trade-off performance vs accuracy, while it supports the `COUNT, SUM, AVG`

AFs. DBL [122], builds a 'learning' layer on top of AQPs (like BlinkDB) in an effort to learn how to reduce errors. VerdictDB [123] develops uniform, hashed, and stratified samples and supports currently `COUNT, SUM, AVG`. Samples are at least 10m-tuples each. It contributes fast error approximation techniques, providing error guarantees with low costs. Our work was inspired by such efforts. We extend the state of the art in this domain by uniquely combining machine learning models, which can generalize and provide high accuracy, even when built over very small samples. These models are very compact guaranteeing large speedups in query times. However, unlike sampling-based AQP research, currently our method does not provide a priori error guarantees.

More recently model-based/learned approaches for AQP emerged. Such efforts include DeepDB [77], deep generative models [153], etc. and they can achieve lower query response times and higher accuracy. For instance, DeepDB introduces Relational Sum-Product Networks (RSPNs), which is used to learn a representation of tables and use the RSPNs to provide an approximate answer. [142] developed clustering techniques to derive low-error density estimators (DEs) and showed how to use them for COUNT/SUM/AVG. It does not use regression models (RMs) and pits DEs vs sampling. FunctionDB, [152] builds Piecewise Linear (PLR) functions over complete datasets and query these functions instead: Queries define data regions, R, using ranges. AFs are computed integrating PLR over sampled data points in R. No DEs are employed and sampling online is expensive, while PLR often suffers from high errors. With respect to space overheads, models tend to be smaller than samples. More recent research on applying ML models include DBL [140] and [11, 13, 14]. Most of these works make assumptions of the expected workload. Recent work on learning to forecast workloads [103] may help overcome some of these assumptions.

Our method builds SML models and queries are answered without any sample, or base data accesses and our method is "first class citizens", unlike DBL, being the only way to answer queries. Unlike [11, 13, 14] it does not depend on a large number of prior queries to learn while it handles many, not

just one aggregate function.

2.3 Sampling Techniques

As stated in the last section, sampling plays an important role in approximate query processing. Many modern AQP engines rely on sampling techniques to make samples [6, 77, 123]. Typical sampling techniques include random sampling, stratified sampling and hash sampling, etc.

2.3.1 Sampling With/Without Replacement

A simple random sample without replacement is a subset of the members of a population, for which each member of the population has the same probability to be included in the sample. No duplicates are allowed. *A simple random sample with replacement* is generated by putting back the selected random element into the population during sampling generation. A simple random sample with replacement is usually generated by iterative, or batch sampling algorithms [117]. In databases, samples are usually generated with replacement, especially for joins where the tuples from the first table may be joined with more than one tuple in the second table. Generating samples without replacement will break the probability that each join path is fetched. In addition, for a simple random sample with replacement, the distribution is a binomial distribution. For a simple random sample without replacement, the distribution is a hypergeometric distribution [159].

Samples are also classified as *dependent* and *independent* samples. *Sample dependency* are measurements made on two samples from the same population. Two samples are called independent if the members chosen from one sample do not determine which members are chosen for the second sample [32]. While for dependent samples, the members in one sample provide information and affect the members in the other sample. Usually, two samples are independent if they are generated with replacement. And they are dependent if they are generated without replacement [125]. Mathematically, the covariance between the two

samples is zero if the two samples are independent. When we sample without replacement, and get a non-zero covariance, the covariance depends on the population size. For cases where the population is very large, this covariance is very close to zero and there is no much difference between the two methods. Sometimes people describe sampling with replacement as sampling from an infinite population and sampling without replacement as sampling from a finite population.

2.3.2 Random Sampling

A simple (uniform) random sample is a subset of a statistical population in which each member of the subset has an equal probability of being chosen. A simple random sample is meant to be an unbiased representation of a group. For some sampling algorithms, only the sampling ratio is provided, and the size of the resulting sample is not fixed. While for other sampling techniques, like the reservoir sampling [27, 42, 164], the size of the sample is fixed. Generating a simple random sample usually requires one scan of the table. However, some sampling techniques like [99] discards some items before the next item is processed and/or accepted, which reduces the time cost to produce the sample significantly. The key observation is that this number follows a geometric distribution and can therefore be computed in constant time.

Weighted random sampling [50, 51] is useful for cases where the sampling probability is provided as the weights associated with each item. Simple random sampling is a special case of weighted random sampling where the weights for all tuples are the same. Weighted random sampling is useful for making samples over the join queries without generating the actual join results.

A random sample usually works well for AQP queries that do not involve filters and `GROUP BYs`. However, for some rare groups in a `GROUP BY` query, a random sample will produce 100% error when no samples are generated for this particular group! A standard approach to solve this problem is stratified sampling.

2.3.3 Stratified Sampling

In stratified sampling [29, 157], the population is divided into homogeneous sub-populations (coined strata). The strata are formed based on specific characteristics or attributes such as gender, income, education, etc. After that, each stratum is sampled by other sampling techniques like simple random sampling. As each stratum properly represents a specific characteristic/attribute, stratified sampling is helpful for a population with diverse characteristics.

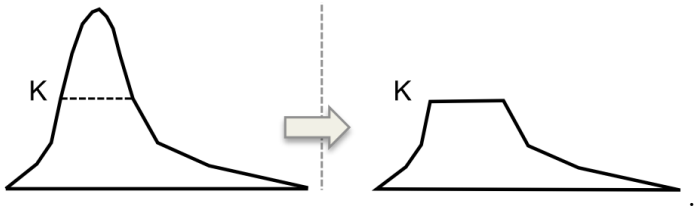


Figure 2.1: Presentation of Stratified Sampling in BlinkDB.

Figure 2.1 demonstrates a derivative of stratified sampling adopted in the AQP engine BlinkDB [6]. Here, the size of each stratum is capped with a fixed size K. For strata with a size less than K, the sample includes all the rows, and there will be no sampling error for that group. In this way, stratified sampling has 100% accuracy for rare groups while maintaining high accuracy for popular groups.

Typically, stratified sampling produces a smaller error in estimation and greater precision than the simple random sampling. In addition, to guarantee the same level of accuracy as simple random sampling, stratified sampling needs a smaller sample, which reduces the storage overheads and query response time during query processing. However, several conditions must be met before stratified sampling could work properly. For instance, stratified sampling could not be applied for scenarios where every member of the population could not be confidently classified into a group. Overlapping [82] is another issue for cases where tuples fall into multiple subgroups. Tuples that are in multiple subgroups are more likely to be selected when random sampling is used, which produces a misrepresentation of the population.

2.3.4 Hash Sampling

Hash sampling is used to produce a uniform sample by applying a uniform hash function to the attributes of interest. Mathematically, given a column set C, a hashed sample S_T over column set C in table T is defined as

$$S_T = \{t \in T | h(t.C) < \tau\} \tag{2.4}$$

Where $h(\cdot)$ is a uniform hash function that maps every value of C in to a real value in $[0, 1]$. The sampling probability $|S_T|/|T|$ is set to τ. For instance, $\tau = 0.01$ produces a 1% sample of the population.

Note, once the hash function and the sampling ratio are selected, hash sampling will always generate the same sample for a given table. Thus, hash sampling generates a *uniform* but *dependent* sample. This is especially useful for cases when we want to join samples instead of tables. It is widely acknowledged that joining random samples won't produce a random sample of the join results (specifically, $sample(R) \bowtie sample(S) \neq sample(R \bowtie S)$) [28, 173]. By using hash sampling, joining hashed samples creates a uniform but dependent sample of the actual join result.

2.4 Challenges of Approximate Query Processing

Prior to the methods proposed in this thesis, approximate query processing was dominated by sample-based approaches. They offer orders of magnitude reduction in query response time and space overheads. Although AQP attracts huge attention since the last several decades, there remain many challenges and opportunities stated as follows:

1. Limited aggregate functions are supported. Currently, most AQP engines only support `COUNT, SUM` and/or `AVG`, while other aggregate functions like `MIN, MAX, VARIANCE, PERCENTILE` are not supported.

2. The support for joins (especially for multi-way joins) remains open.

3. The space overheads are still high. For sample-based approaches, the samples can be very large to guarantee high accuracy. For model-based AQP approaches like DeepDB, the size of the model built over all columns is similar to the size of the actual table [77, 107].

4. For queries with a large number of groups, the error from current AQP solutions is high.

Chapter 3

QReg: Query-Centric Regression

3.1 Motivations

Given the increasing interest in bridging ML and RMs with DBs, as stated in Chapter 1, we focus on how seamless this process can be. ML models (and RMs in particular) are trained to generalize and optimize a loss function (invariably concerning *overall expected error*). We refer to this as a **workload-centric** view, as the aim is to minimize expected error among all (possible) future queries in an *expected* workload. The probability distribution of data items within the data set and the need for generalization drive this process. The focus thus is on ensuring accuracy across all possible future queries. However, data systems research has long recognized that query workloads follow patterns generally different to data distributions [10, 93]. Therefore, from a data analyst's point of view, she is less interested in a model that ensures accuracy over the whole set of possible queries, but on a model that will ensure as high accuracy as possible for each of her specific queries, which target specific data subspaces (e.g., using range or equality selection predicates on various attributes).

If an ML model is weak for a particular query (part of the dataset), it will suffer from high errors for this query, even though over the whole expected workload it can achieve high accuracy. And, if such queries are popular, the

majority of queries will experience undesirable accuracy. This gives rise to the need for a "query-centric perspective": We define **query-centric regression** as a model which strives to ensure both high average accuracy (across all queries in a workload) as well as high *per-query* accuracy, whereby each query is ensured to enjoy accuracy close to that of the best possible model. Specifically, given a large number of point queries, the target is not to find a single best model to serve all the queries, but to dynamically select a best model per query.

The ML community's general answer to such problems is to turn to ensemble methods, in order to lower variance and generalize better (e.g., to different distributions). Nevertheless, the above discussion reveals a potentially serious type of an *impedance mismatch* in the sense that the general ML approach to train models to generalize needs to be used in an environment where overfitting is also important.

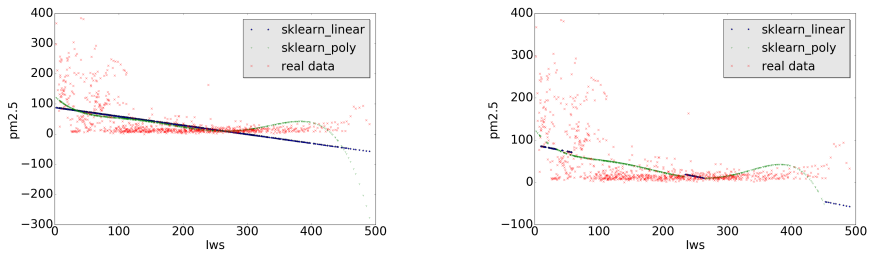
The above also bears strong practical consequences. Consider a data analyst using python or R, linked with an ML library (like Apache Spark MLLib, Scikit-Learn, etc.), or using a DB connector like MonetDB.R or SciDB-Py, or a prediction serving system like Clipper. The analyst uses a predicate to define a data subspace of interest and calls a specific RM method: It would be great if she knew beforehand which RMs to use for which data subspaces. Alternatively, it would be highly desirable if the system could select the best RM automatically for the analyst's query at hand.

We wish to shed light into this possible impedance mismatch problem, see if it holds for simpler and even for state-of-the-art ensemble RMs, and discover valuable experience for the seamless use of RMs for in-DBMS analytics. Specifically, we wish to (i) quantify the phenomenon: in Section 3.5 , we shall use several real data sets (from the UCI ML repository and TPC-DS data) and a wide variety of popular RMs and new metrics that reveal workload-centric and query-centric performance differences. We will reveal the need for different RMs for different datasets, and even for different sub-space of the dataset and (ii) see if the problem can be addressed by adopting a query-centric perspective,

(using a new ensemble method named QReg) whereby instead of selecting a single model for all queries, a best model is selected per query. We wish to push the lower bound of prediction errors from regression models by combining the strength of base regression models. A detailed analysis will be carried out in Section 3.6.

3.2 Exemplifying the Problem

In general, the data space of interest is denoted $\mathbb{D}(\mathbf{x}, y) \in \mathbb{R}^{d+1}$, where $\mathbf{x} = [x_1, ..., x_d]$ are the features (independent variables) and y is the label (dependent variable). Let us consider the Beijing PM2.5 pollution problem [100] as an example. The aim is to predict PM2.5 concentrations based on several variables. This is a real-life dataset and there exists a good relationship between PM2.5 and other variables. To simplify the problem, only Cumulated Wind Speed (m/s) **Iws** is used as the feature. Thus, the data of interest is $\mathbb{D}(\mathbf{Iws}, PM2.5) \in \mathbb{R}^2$, and the feature $\mathbf{Iws} \in \mathbb{R}^1$.



(a) Linear and polynomial curves (b) target regression curve

Figure 3.1: Beijing PM2.5 problem of 2-dimensional space

Figure 3.1(a) shows the problem space $\mathbb{D}(\mathbf{Iws}, PM2.5)$. Two base models, linear regression and polynomial regression (from the Scikit-Learn package [126]), are used to fit the data. Linear regression fits better than polynomial regression if Iws lies between [450,500], while polynomial regression has higher accuracy between [250, 450]. This, exemplifies that a regression model can only be good (accurate) at some unknown specific sub-spaces in the dataset. What we are after is to combine the strengths of the base regression models in order to create a new regression method, whose performance will equal that of the

curve shown in Figure 3.1(b), where obviously the overall accuracy is much better than any of the base models. (For simplicity, hereafter we consider the input to F to be one-dimensional, so we will write $F(x)$ instead of $F(\mathbf{x})$.)

More generally, there may be n base regression models $\{R_i\}_{i=1}^{n}$ available, with the corresponding prediction functions $R_i(x) = f_i(x)$. Creating a prediction function $F(x)$ based on these base models $\{f_i\}_{i=1}^{n}$, $F = g(f_1, ..., f_N)$, which ensures that the best prediction model is selected for each query x_q is obviously desirable.

$$F(x_q) = g(f_1(x_q), ..., f_n(x_q)) = f_k(x_q),\ k \in \{1, 2, ..., n\} \tag{3.1}$$

For each query, x_q, function $F(x)$ finds the best model ($f_k(x_q)$) and uses it. This is a model selection problem to find the best regression model for each query. This can be treated as an $n - way$ classification problem, whose output labels identify one (the best) of the n base regression models. Thus, for a given query x_q, this classifier assigns it to the best prediction model, and this model will be used to answer this specific prediction query.

Our analysis of the problem will test to see whether indeed the above is a real-world problem. It will formulate the basic problems and will test them across a large number of real-world data sets and across a large number of base regression models and ensemble regression models. Subsequently, it will shed light into how one decides which should be the constituent regression models to use. And, on how important is the decision on which classifier to use. And, on how much improvement in accuracy can one anticipate. And, at what costs.

3.3 Our Systemic Setup and Design Choices

3.3.1 System Architecture

Assume that the data system maintains m regression models. When a query arrives, the system needs to identify the model with the least prediction error for this query. We treat this model selection problem as a classification problem.

Figure 3.2 shows the architecture of this classified regression *QReg* [1].

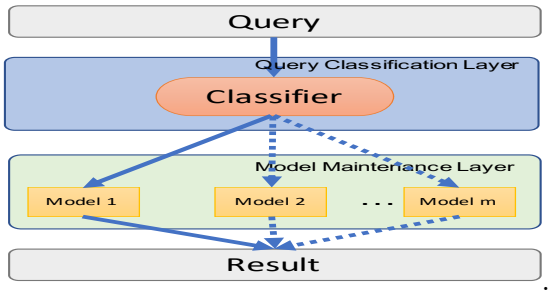


Figure 3.2: QReg Architecture.

There are two layers in the system. (i) The **model maintenance layer**, deploys and maintains the base regression models. (ii) The **query classification layer** implements the core of *QReg*. A query is first passed to the pre-trained classifier. Because the classifier "knows" the prediction ability of each model in the various queried data spaces, the query will be assigned to the model that performs best locally for *this* query's data space.

Two configurations are studied for *QReg*: *Simple QReg* uses simple regressions, including LR, PR, and DTR. *Advanced QReg* uses GBoost and XGBoost as its base models. Although the selection of base regression models depends on the user's choice, we find that having RMs with the same level of accuracy achieves better performance for QReg. And a detailed discussion about the selection of base RMs is made in Section 3.3.3.

	Simple QReg	Advanced QReg
Base Models	LR, PR, DTR	GBoost, XGBoost

Table 3.1: QReg Configurations

3.3.2 Model Training Strategy

Figure 3.3 shows the model training strategy of *QReg*. In Step 1, the data set is split into three partitions: (i) RM Partition is used to train the base regression models; (ii) QReg Partition is used to train the classifier in *QReg*; (iii) Testing Partition is used to evaluate the accuracy of *QReg*.

[1]The source code for *QReg* is available at `https://github.com/qingzma/qreg`

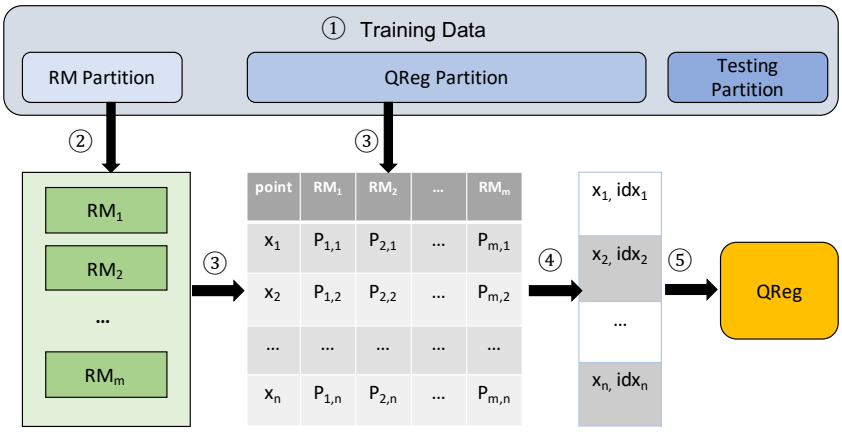


Figure 3.3: Model Training Strategy of *QReg*.

In Step 2, the RM Partition is used to train m regression models. The selection of base models is in principle open and depends on the users' choice (taking into account the above issues). For Simple QReg, the base models include linear regression, polynomial regression and decision tree regression. After that, Each base model RM_i makes predictions $p_{i,j}$ of each data point x_j in QReg Partition, as shown in Step 3. In Step 4, a comparison is made between the predicted $p_{i,j}$ and the real label y_j to find the best prediction model for each query x_j. Having the individual predictions and associated errors, a new data set is generated by combining the data point x_j and the index idx_j of the best model for this query, depicted $[x_j, idx_j]$. This data set is then used to build the classifier reflecting the prediction ability of base models in the query space. The classifier is the core of *QReg*, and a well-designed classifier could potentially grasp the prediction ability of each model in the query space correctly. Thus, the prediction accuracy can be significantly improved compared to individual prediction models.

Note that the original data set is partitioned into 3 subsets instead of 2. This is done in order to ensure that different training data sets are used to train the base models and the classifier. In addition, models are fine-tuned via cross-validation using *GridSearchCV()* in the scikit-learn package.

3.3.3 Candidate Base Models

When deciding which models to include in QReg the following key criteria are considered.

(a.) Model training time. This should be as low as possible and should exhibit good scalability as the number of the training data points increases.

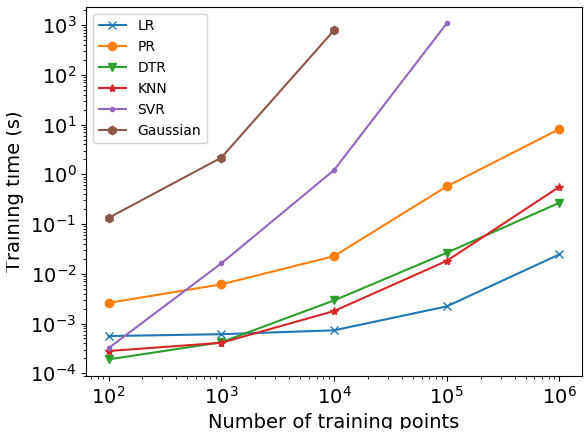


Figure 3.4: Training Time of Typical Regression Models.

Given the asymptotic complexity, as summarised in Section 2.1.1, a number of experiments were conducted to quantify the training times for various regression models. Figure 3.4 is a representative result for data set 4 using 4 dimensions (listed in Table 3.2). It shows how model training time (for six regression models) is impacted as the number of training instances increases.

Model training time is shown to behave acceptably with respect to the number of instances in the training data set for LR, PR, DTR, and kNN regression. The training time of Support Vector Regression – Radial Basis Function tends to increase much more aggressively as the number of training points increases. The experiment was repeated for all data sets. The above conclusion holds across all experiments and are omitted for space reasons.

(b.) Query response time: During the classifier's training process in QReg, predictions are made from each base prediction model. To reduce the overall training time as well as the query response time, the models should have as low response time as possible. Figure 3.5 shows the typical query response

time of various regression models for data set 4 using 4 dimensions. And the size of the data set is set to 10,000. Clearly, LR, PR, DTR and SVR takes less than 100us to make a prediction, while kNN takes ca. 250us to respond.

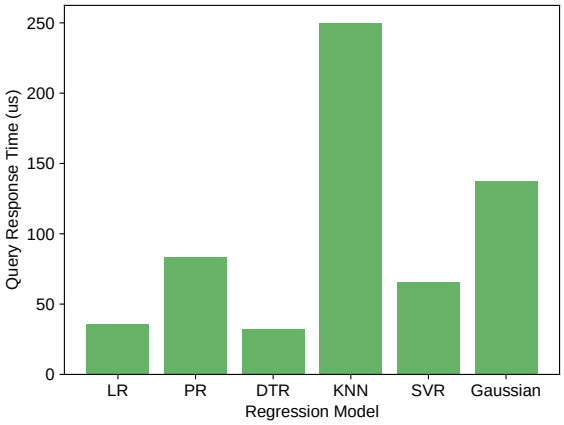


Figure 3.5: Query Response Time of Typical Regression Models.

(c.) Prediction accuracy: Figure 3.6 compares the average absolute error. For this specific data set, LR, PR and DTR achieve the least error. While the error of kNN, SVR and Gaussian is much higher. If only LR, PR and DTR are used as the base models, the average absolute error obtained by QReg is 0.0389. However, if kNN is added as another base model (LR, PR, DTR and kNN are used as the base models), the average absolute error increases to 0.0489.

This is an interesting issue arising from using different regression models together, as in *QReg*. If the base regression models have large differences in accuracy levels, then this may result in *QReg* having poorer accuracy as *QReg* might be distracted by worst-performing RMs. This is a direct result of errors introduced during the separate classification process. Therefore, care must be taken so to ensure that base models enjoy similar and good accuracy levels.

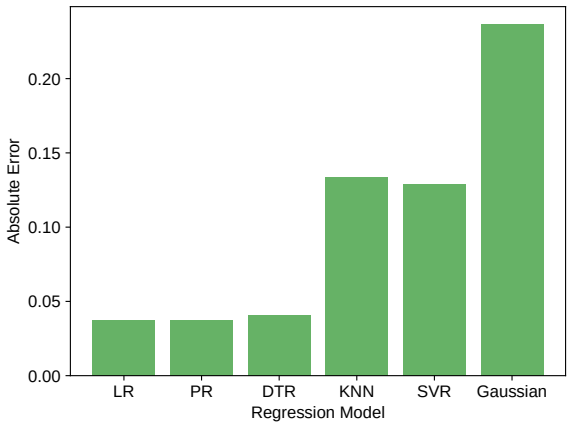


Figure 3.6: Absolute Error of Typical Regression Models for Data Set 4.

3.4 Experimental Setup

All experiments run on an Ubuntu system, with Intel Core i5-7500 CPU @ 3.40GHz × 4 processors and 32GB memory.

3.4.1 Data Sets and Dimensionality

Eight real-world data sets with different characteristics from the UCI machine learning repository [101] are used, varying the dimensionality from 2 to 5, as well as a large fact table from the TPC-DS benchmark [115]. In Table 3.2, we only show the features and labels used in the 5-d experiments.

ID	Data set	# of Records	Features & Label (5d)
1	Online Video Time [46]	168,286	duration, width, i_size, utime , *umem*
2	Physicochemical Properties	45,730	F2, F3, F4, F5 , *RMSD*
3	Beijing PM2.5 Data [100]	43,824	DEWP, TEMP, PRES, Iws, *energy_output*
4	Online News Popularity [55]	39,797	NTT, NTC, NNSUT, NH, *NUT*
5	Combined Cycle Power Plant [158]	9,568	T, EV, AP, RH, *EO*
6	YearPredictionMSD [19]	515,345	c1, c2, c3, c4, *year*
7	Gas sensor [22]	4,178,504	Methane_conc_(ppm), c1, c3, c4, *c2*
8	electric power consumption	2,075,259	GAP, GI, GRP, V, *energy*
9	store_sales [115]	2.6billion	ss_sales_price, etc, *ss_quantity*

Table 3.2: Characteristics of data sets used in experiments.

Data set 1 is a collection of YouTube videos showing input and output video characteristics along with the transcoding time and memory requirements. Data set 2 contains Physicochemical properties of Protein Tertiary Structure. The task is to predict the Size of the residue (RMSD) based on several properties. There are 45730 decoys and size varies from 0 to 21 armstrong. Data set 3 is an hourly data set containing the PM2.5 gas concentration data in Beijing.

The task is to predict PM2.5 concentration (ug/m^3), and the independent variables include pressure (PRES), Cumulated wind speed (Iws), etc. Data set 4 is an online news popularity data set and tasks include predicting the number of shares in social networks (popularity). There are totally 39797 records in this data set. In our experiments, the feature adopted is the number of unique tokens, and the labels include the number of token content, number of none-stop unique tokens, etc. Data set 5 contains 9568 data points collected from a Combined Cycle Power Plant over 6 years (2006-2011), and the task is to predict the net hourly electrical energy output (EP) of the plant. Data set 6 is the YearPredictionMSD data set used to predict the release year of a song from audio features. Most of the songs are commercial tracks from 1922 to 2011. Data set 7 contains the recordings of 16 chemical sensors exposed to two dynamic gas mixtures at varying concentrations. The goal with this data set is to predict the recording of one specific chemical sensor based on other sensors and the gas mixtures. This is a time-series data set containing more than 4 million records in total. Data set 8 records the individual household electric power consumption in one household for more than four years, and there are two million records. Features include the voltage, household global minute-averaged active power (in kilowatt), etc.

We further employ table `store_ sales` from the popular TPC-DS benchmark [115]. Typical columns used for the experiments include `ss_list_price`, `ss_wholesale_cost`, `ss_sales_price` and `ss_ext_sales_price`.

3.4.2 Experimenting with QReg for AQP Engines

We further applied QReg to DBEst, a newly model-based approximate query processing (AQP) engine [104] (which will be introduced in Chapter 4), to demonstrate QReg's performance. DBEst adopts classical machine learning models (regressors and density estimators) to provide approximate answers to SQL queries. We replace the default regression model in DBEst (XGBoost) with *Advanced QReg*, and compare the accuracy with DBEst using other ensemble methods, including XGBoost and GBoost. The well-known TPC-DS dataset

is scaled up with scaling factor of 1000, which contains $\sim$ 2.6 billion tuples (1TB). 96 synthetic SQL queries covering 13 column pairs (like , [ss_quantity, ss_ext_sales_price], [ss_wholesale_cost, ss_list_price], etc) are randomly generated for `COUNT`, `SUM` and `AVG` aggregate functions. For instance, for column pair [ss_quantity, ss_sales_price], the SQL queries are of the following format:

```
SELECT SUM(ss_quantity) FROM store_sales
WHERE ss_sales_price BETWEEN lower  AND upper
```

where *lower* and *upper* are randomly generated within the space domain. The DBEst engine is trained over a sample of 100k rows from the complete dataset. The complete experiments and discussions are introduced in Section 3.7.6.

3.4.3 Evaluation Metrics

Accuracy is measured using the Normalized Root Mean Square Error (**NRMSE**) metric, defined as:

$$NRMSE = \frac{\sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t - y_t)^2}{n}}}{y_{max} - y_{min}} \tag{3.2}$$

NRMSE shows overall deviations between predicted and measured values; it is built upon root mean square error (RMSE), and is scaled to the range of the measured values. It provides a universal measure of prediction accuracy between different regression models.

The NRMSE ratio r, compares the prediction accuracy of one RM_i against that of any other RM_j, and is defined as: $r = \frac{NRMSE_i}{NRMSE_j}$. If $NRMSE_j \leq NRMSE_i$, this ratio shows how worse RM_i is compared to RM_j.

The above are standard metrics used for comparing accuracy. However, our study calls for additional metrics. Inherent in our study is the need to reflect the differences in accuracy observed by a query as they depend on the model used. For this we define the concept of *Opportunity Loss* as a natural way to reflect how much the query loses in accuracy by using a sub-optimal model.

Assuming RM_{opt} is the RM with the lowest NRMSE, we define *Opportunity*

Loss OL_i as

$$OL_i = \frac{NRMSE_i}{NRMSE_{opt}} - 1, \tag{3.3}$$

which quantifies as a % the error (the opportunity loss) due to not using the best model RM_{opt} and using RM_i instead.

Furthermore, we define

$$ROL_{i,k} = \frac{OL_i}{OL_k}, \tag{3.4}$$

as *Relative Opportunity Loss*, which quantifies how much better RM_k does vs RM_i in improving on the opportunity loss.

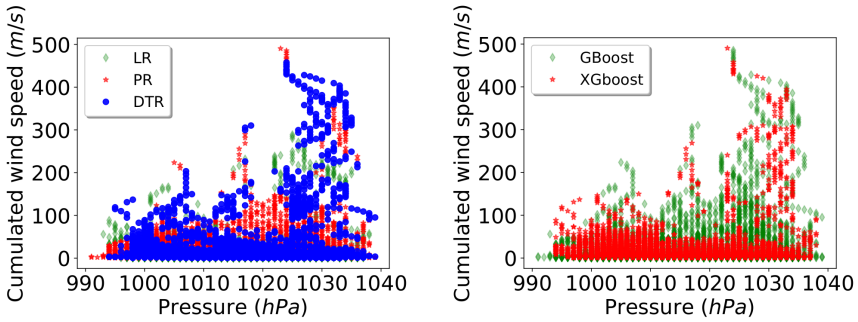
Intuitively, our aim (which will be discussed in Section 3.5) is to show that, despite which single model is used, some queries will always be processed by sub-optimal models. So we wish to quantify this opportunity loss. Furthermore, our aim (which will be studied in Section 3.6) is to show that a new ensemble model can help significantly alleviate this problem. The $ROL_{i,j}$ metric will help quantify how much a model (*QReg*) improves on this opportunity loss for queries.

3.5 Query Space Exploration

As introduced in Section 3.1, this section will analyze the performance of various RMs for different datasets, or even for different sub-regions of the data space. Consider data set 3, the Beijing PM2.5 data set [100], using Cumulated Wind Speed (IWS) and Pressure (PRES) as the features, yielding a 3-dimensional regression problem.

Figure 3.7.(a) shows the distribution of the model with the least error for all data points. LR, PR, and DTR are used as the base RMs (*Simple QReg*). Figure 3.7.(b) shows the distribution of best models when ensemble models GBoost and XGBoost are used (*Advanced QReg*).

Take *QReg* using simple models as an example. LR dominates in the upper-central region. PR dominates at the lower central regions. DTR performs best



(a) Simple models (b) Ensemble models

Figure 3.7: Distribution of best models for Beijing PM2.5.

in the rest of the space. The NRMSEs for LR, PR and DTR are 8.48%, 8.84%, and 8.32%, respectively. However, if for each point the best model can be selected to make the prediction (as shown in Figure 3.7), the corresponding ("optimal") NRMSE drops to 7.19%. This would be a large improvement in accuracy.

Figures like Figure 3.7 can help analysts decide on which models to use when querying this space. Similar figures exist for all data sets studied in this work and are omitted due to space reasons.

Table 3.3: Win-counts of simple RMs.

Data set ID	Count of LR	Count of PR	Count of DTR
1	6468	6919	5366
2	3606	4133	6217
3	1077	1580	1472
4	3618	4826	4155
5	980	885	1307
6	62749	51953	57007
7	9646	4029	4953
8	35702	28800	40311

Table 3.3 adopts a different perspective. It shows the number of most accurate predictions ("wins") made by each simple RM, per data set. First, note that all models have a good number of wins. Second, no model has the highest number of wins across all data sets. So, there is no single winner. DTR enjoys the most wins for data sets 2, 5, and 8; PR makes the most accurate predictions for data sets 1, 3, and 4; LR wins for data sets 6 and 7. Note, the original data set is much larger than the numbers shown in Table 3.3. For the

2-d problem, we remove the duplicates.

Table 3.5 zooms in, augmenting Table 3.3 by showing the NRMSEs when different simple RMs win. For example, for data set 1, we know from Table 3.3 that LR wins 6468 times. For these, the LR's NRMSE was 11.08%, as indicated by Table 3.5, whereas for PR was enormous and for DTR was 18.28% – see the 3 numbers in cell [1,3] in Table 3.5. Similarly, for the 6919 queries where PR won, LR's NRMSE was 11.82%, as indicated by Table 3.5 whereas for PR was 9.35% and for DTR was 11.68% – see the 3 numbers in cell [1,4].

Table 3.4: Win counts of all models.

ID	LR	PR	DTR	GBoost	XGBoost
1	5193	5113	1692	3581	3174
2	2964	3050	3912	1974	2056
3	858	1179	1144	722	226
4	2870	3088	2792	1475	2374
5	670	490	855	754	403
6	57083	34105	44908	14899	20714
7	7088	1253	1556	6089	2642
8	29839	20213	19876	22189	12696

Consider data set 4. When LR wins, its error is markedly lower (almost half) that of PR and DTR–unlike their overall NRMSEs which show LR to be the worst model.

To further facilitate a query-centric perspective, we delve into the performance of the queries for which each RM reached a top-20% performance. For data set 1, for example, this includes the best 20% of the 6468 queries for which LR wins, the best 20% of the 6919 queries for PR wins, and the best 20% of the 5366 queries for DTR wins. Hence, the NRMSE of interest does not come from all the queries, but from the top 20% queries for which the least error was achieved by a simple RM. Table 3.6 shows these results, along with the overall NRMSE of each simple RM for the whole set of queries. Again, note that the overall NRMSEs are quite close. However, individual differences are very large. For data set 1, for instance, the top 20% of queries when LR wins enjoy an NRMSE that is about half of the NRMSE of the others. Interestingly, the same holds for PR and DTR! Similar conclusions hold for the other data sets.

Table 3.5: NRMSE values when different RMs win.

Data set ID	NRMSE By	NRMSE where LR wins	NRMSE where PR wins	NRMSE where DTR wins	Overall NRMSE
1	LR	**11.08%**	11.82%	14.23%	12.32%
	PR	≫1000%	**9.35%**	≫1000%	≫1000%
	DTR	18.28%	11.68%	**10.60%**	14.06%
2	LR	**27.68%**	23.78%	28.61%	27.02%
	PR	30.96%	**20.20%**	27.85%	26.72%
	DTR	34.24%	26.18%	**21.81%**	26.79%
3	LR	**8.47%**	8.00%	8.99%	8.48%
	PR	9.91%	**6.60%**	10.04%	8.84%
	DTR	10.54%	7.99%	**6.65%**	8.32%
4	LR	**2.46%**	4.96%	5.55%	4.62%
	PR	3.82%	**2.62%**	4.49%	3.67%
	DTR	4.07%	3.54%	**2.51%**	3.41%
5	LR	**16.69%**	15.97%	19.91%	17.90%
	PR	18.75%	**13.94%**	18.79%	17.56%
	DTR	21.07%	16.99%	**15.30%**	17.72%
6	LR	**11.06%**	12.43%	13.39%	12.29%
	PR	12.15%	**11.47%**	12.77%	12.15%
	DTR	12.30%	12.04%	**12.15%**	12.17%
7	LR	**80.48%**	113.75%	106.84%	95.85%
	PR	210.45%	**83.39%**	104.22%	165.31%
	DTR	118.60%	111.17%	**76.58%**	107.31%
8	LR	**8.81%**	10.02%	10.49%	9.82%
	PR	10.36%	**8.05%**	10.04%	9.66%
	DTR	11.13%	9.99%	**8.29%**	9.80%

Each [i,j] cell shows 3 values for data set i, for the cases where model j wins. $j = 3$ (4, or 5) represents LR, (PR, or DTR) respectively. The top number in each cell shows the NRMSE for LR, the middle shows the NRMSE of PR, and the bottom the NRMSE for DTR.

We conducted the same experiment using more sophisticated ensemble regression models. Similarly to Table 3.3 and Table 3.5 which refer to simple RMs, Table 3.7 and Table 3.8 show the corresponding results for the ensemble RMs.

Interestingly, a close examination of these tables substantiates the same conclusions as above, but for ensemble methods. Again, we see from Table 3.7 that both RMs enjoy a large number of wins. From Table 3.8 we see that NRMSE differences are very high between RMs for queries when GBoost or XGBoost win.

For completeness, Table 3.4 lists the win counts when all models (LR, PR,

Table 3.6: NRMSEs for the top 20% queries per simple RM.

Data Set ID	NRMSE By	NRMSE where LR wins	NRMSE where PR wins	NRMSE where DTR wins	Overall NRMSE
1	LR	**3.61%**	5.36%	6.67%	5.36%
	PR	7.06%	**2.92%**	6.33%	5.72%
	DTR	7.96%	5.78%	**3.42%**	6.01%
2	LR	**18.35%**	16.14%	16.10%	16.89%
	PR	21.26%	**12.39%**	12.88%	16.04%
	DTR	24.14%	17.09%	**7.97%**	17.69%
3	LR	**3.03%**	3.64%	4.83%	3.902%
	PR	4.89%	**1.44%**	4.07%	3.76%
	DTR	5.35%	2.78%	**2.12%**	3.69%
4	LR	**0.96%**	2.33%	2.74%	2.15%
	PR	2.33%	**0.63%**	1.56%	1.66%
	DTR	2.54%	1.40%	**0.78%**	1.74%
5	LR	**7.56%**	9.77%	9.44%	8.98%
	PR	10.00%	**7.75%**	8.11%	8.68%
	DTR	12.11%	10.55%	**4.85%**	9.69%
6	LR	**4.11%**	5.26%	5.500%	5.00%
	PR	5.22%	**4.12%**	4.82%	4.74%
	DTR	5.42%	4.76%	**4.13%**	4.80%
7	LR	**42.14%**	109.35%	95.42%	87.25%
	PR	106.72%	**80.88%**	82.48%	90.80%
	DTR	79.49%	106.48%	**65.26%**	85.47%
8	LR	**3.07%**	5.57%	3.65%	4.23%
	PR	4.37%	**3.77%**	3.67%	3.95%
	DTR	5.17%	5.92%	**1.60%**	4.63%

DTR, GBoost and XGBoost) compete. Each model has a considerable win count, substantiating the need for a query-centric perspective. Noticeably, although ensemble models are designed to have higher prediction accuracy than simpler models, the win counts of simpler models are often shown to be higher than those of ensemble models.

To summarize this section, we extensively evaluate the performance of various RMs over different data sets, or different regions of the queried data spaces. Experimental results show that different RMs might exhibit higher accuracy for different regions of the queried data spaces, and a top-performing ensemble method might not work well for all data sets. In addition, although said RMs might enjoy similar overall accuracy, the prediction difference can be large for different regions. In the following section, we will propose an

Table 3.7: Win counts of ensemble RMs.

Data set ID	Count of GBoost	Count of XGBoost
1	10334	8419
2	7037	6919
3	2653	1476
4	2142	10457
5	1596	1576
6	92916	78793
7	6991	11637
8	52949	51864

Table 3.8: NRMSEs when different ensembles win.

Data set ID	NRMSE By	NRMSE where GBoost wins	NRMSE where XGBoost wins	Overall NRMSE
1	GBoost	**14.22%**	16.34%	15.21%
	XGBoost	14.55%	**16.02%**	15.23%
2	GBoost	**25.43%**	26.68%	26.06%
	XGBoost	26.15%	**25.59%**	26.05%
3	GBoost	**9.24%**	7.17%	8.13%
	XGBoost	10.53%	**6.20%**	8.35%
4	GBoost	**2.67%**	12.97%	11.87%
	XGBoost	4.62%	**2.85%**	3.22%
5	GBoost	**16.21%**	18.66%	17.47%
	XGBoost	17.38%	**17.42%**	17.40%
6	GBoost	**10.91%**	13.45%	12.15%
	XGBoost	11.07%	**13.30%**	12.15%
7	GBoost	**103.56%**	97.12%	99.58%
	XGBoost	121.14%	**87.67%**	97.58%
8	GBoost	**9.04%**	10.38%	9.72%
	XGBoost	10.18%	**9.35%**	9.77%

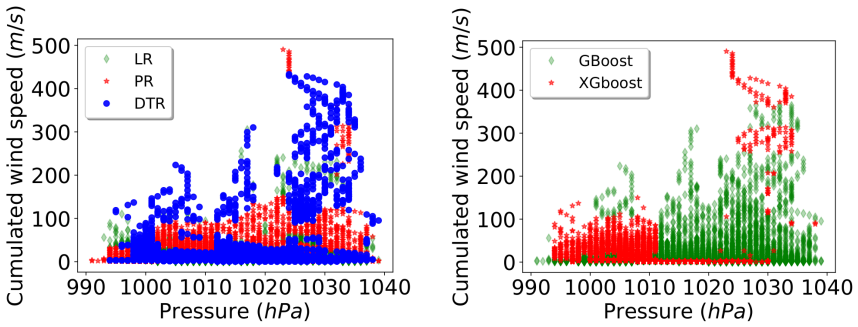
ensemble method, coined *QReg*, based on the key findings from this section. It is designed to reduce the prediction error based on the error distribution of various RMs in the queried space.

3.6 QReg Evaluation

This section aims to substantiate whether it is possible to develop a method that can learn from the key findings of the previous section and leverage them, automating the decision as to which RM to use, relieving the DB user/analyst of the conundrum, towards a query-centric RM. Specifically, we study if and

at what costs a method can: (i) select the best regression model for any query at hand and (ii) achieve better overall accuracy than any single (simple or ensemble) method.

It is natural to treat this problem as a model selection problem, using a classifier for the method selection. We show a new ensemble method, *QReg*, which materializes a query-centric perspective achieving the above two aims.[2] We have considered various classifiers for *QReg*, including SVM-linear classifiers, SVM classifiers using the RBF kernel, and the XGBoost classifier. In Section 3.6.5, we conducted a comprehensive comparison between various classifiers. Unless explicitly stated otherwise, results for the XGBoost classifier are shown, due to its overall prediction accuracy and scalability performance.



(a) Simple *QReg* (b) Advanced *QReg*

Figure 3.8: *QReg* distribution of base models.

Figure 3.8 shows the RMs chosen by *Simple and Advanced QReg* for each corresponding query, giving a feeling of the overall RM distribution suggested by *QReg*. The model distribution shown resembles the distribution of the truly optimal models across the queried data space, as shown in Figure 3.7. Thus, *QReg* does a good job in selecting (near-) optimal RMs per query. As per Scenario 1, presenting such visualisations can be of real value to analysts.

3.6.1 Workload-centric Perspective: Simple QReg

A workload-centric perspective assumes that the query distribution is identical to the data distribution, as described in Section 3.1. Simple *QReg* uses simple

[2]NB: the aim here is *not* to find the best method to achieve this. By presenting *QReg* we showcase that (i) this is achievable and that (ii) significant gains can be achieved using easy-to-deploy methods.

regression models, including LR, PR, and DTR. Figure 3.9 shows the NRMSE ratio r as defined in Equation (3.2) for all data sets in 3-d space. An NRMSE ratio r larger than 1 means *QReg* has less prediction error than the other base model. *QReg* is shown to outperform or be as good as any of its base models. Specifically, for data sets 2, 3, 5, 6, and 8, *QReg* performs slightly better than other regression models, whereas for data sets 1, 4, and 7, we can see *QReg* being significantly superior versus LR, or PR, or DTR. This shows that even the base RMs perform badly, *QReg* has the ability/potential to obtain much better accuracy.

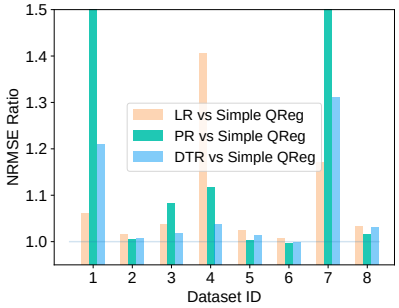
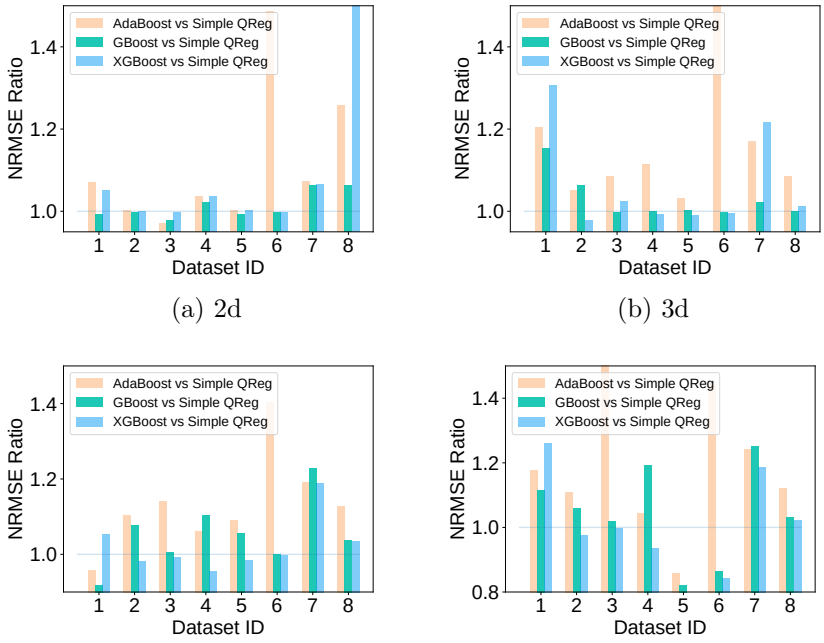


Figure 3.9: Accuracy of *Simple QReg* vs LR, PR, DTR.



(a) 2d (b) 3d

(c) 4d (d) 5d

Figure 3.10: Accuracy of *QReg* vs ensemble RMs

Figure 3.10 compares the prediction error between *Simple QReg* against the more sophisticated ensemble methods, including AdaBoost, GBoost, and XGBoost. Figure 3.10 shows that the even *Simple QReg* often achieves better prediction accuracy than any of the sophisticated ensemble methods. For example, up to 25% reduction in NRMSE is achieved by *Simple QReg* for data set 1 in 3-d space. Note, for the 5-d cases, *Simple QReg* perform worse than other ensemble method for data set 5 and 6. And the corresponding NRMSE ratio is ca. 0.85. This urges us to use more advanced ensembles as the base models for such data sets.

There are two sources of error for *QReg*: one comes from the base regression models, and the other from the classifier. With Table 3.9 we show the proportion of classification error of the total *QReg* error $NRMSE_{QReg}$. For most cases, the error caused by improper classification is less than 20% of the total *QReg* error. However, we see that for datasets with higher dimensionality, the classification error can become increasingly important. A detailed discussion on the selection of classifiers is provided in Section 3.6.5.

Table 3.9: Classification error % of *Simple QReg*

Data Set ID	2D	3D	4D	5D
1	6.85%	9.96%	32.79%	18.42%
2	4.47%	14.76%	16.19%	17.50%
3	8.01%	11.91%	11.94%	16.52%
4	9.74%	25.13%	28.11%	30.38%
5	6.93%	12.38%	19.82%	39.04%
6	3.91%	5.89%	3.29%	24.19%
7	0.00%	2.65%	3.16%	4.35%
8	9.98%	12.53%	13.41%	14.71%

To summarize this section, we evaluate the performance of simple *QReg* against its base RMs and the state-of-the-art ensembles. *QReg* achieves much better accuracy than its base RMs, and even outperforms other ensembles over various datasets. In the next section, we will repeat the same experiment, but for advanced *QReg*.

3.6.2 Workload-centric Perspective: Advanced QReg

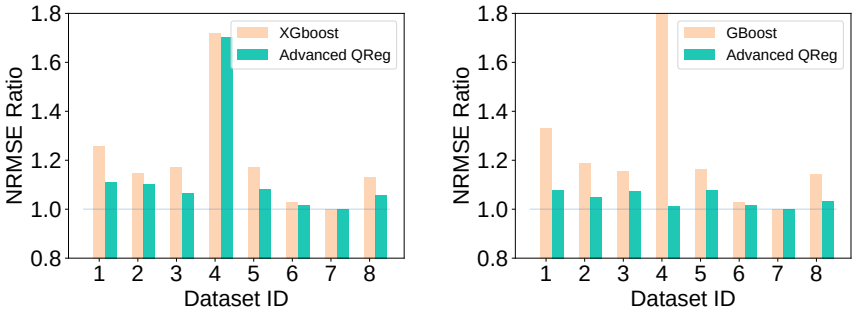
For the majority of the cases, *Simple QReg* is shown to outperform simpler RMs and occasionally more complex ensemble models. For the remainder we concentrate on *Advanced QReg* constructed using GBoost and XGBoost.

Delving deeper, we now show the NRMSE ratios between GBoost (or XGBoost) and *QReg*, broken down to sub-collections of points in the data space, specifically, for the sub-collection of points where XGBoost regression (or GBoost regression) has the best prediction accuracy.

Figure 3.11 shows bands of 2 bars each. Each band of bars shows the NRMSE ratio between other RM and the best RM for the collection of points. Take the 4-d data set 1 as an example, for the collection of points where XGBoost has the best prediction accuracy. The NRMSE ratio between GBoost (second best RM) and XGBoost (collection-best RM) is 1.3288 (orange bar in the figure), while the corresponding NRMSE ratio between *QReg* and XGBoost is 1.0780 (green bar in the figure). This shows that for this collection of points where XGBoost has the best prediction accuracy, GBoost suffers from a 32.88% error relative to the optimal, while using *QReg* reduces this to 7.80%.

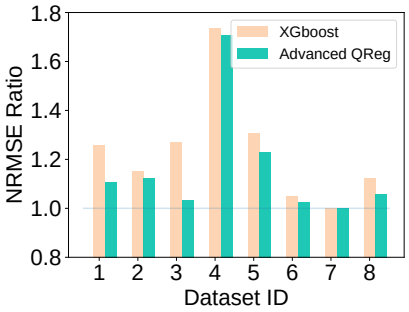
As another example, consider the collection of points where XGBoost has the best prediction accuracy in the 5-d data set 8. The NRMSE ratio between GBoost (second best RM) and XGBoost is 1.1458, while the NRMSE ratio between *QReg* and XGBoost is 1.0316. Thus, the relative opportunity loss is $0.1458/0.0316 = 4.61$, which means the error caused by using GBoost (relative to the best model) is 4.61 times the error caused by *QReg* for the collection of points where XGBoost has the best accuracy. The relative opportunity loss is much larger for data sets 4 and 5. For data set 7, GBoost and XGBoost have almost identical performance. And the corresponding NRMSE ratio is 1.0.

Figure 3.12 shows the ratio r of NRMSE between the base (ensemble) methods and *Advanced QReg* for the whole data sets. Improvement in 2-d space is typically small, but from d=4, we start seeing larger improvements brought about by *QReg*. For 2-d case, the accuracy of GBoost and XGBoost

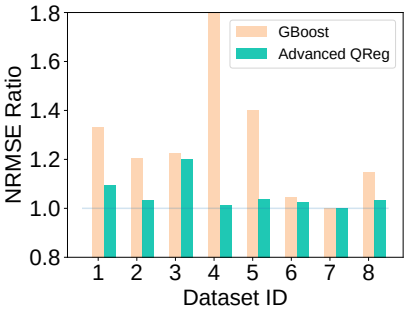


(a) 4-d GBoost win collection

(b) 4-d XGBoost win collection



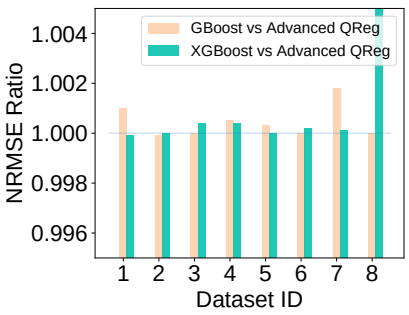
(c) 5-d GBoost win collection



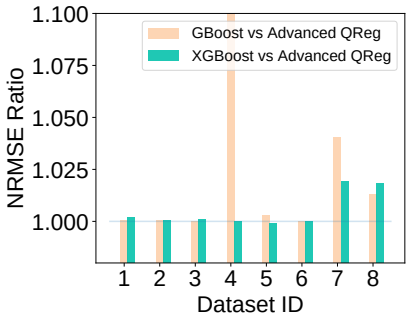
(d) 5-d XGBoost win collection

Figure 3.11: Workload-centric collection-level NRMSE ratio

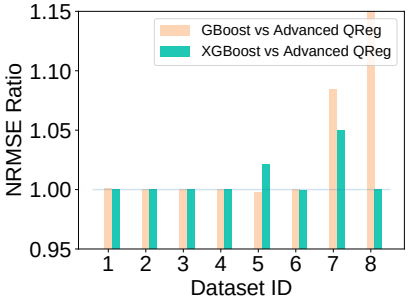
regressor are high and almost equal, which explains why *QReg* cannot improve things further. For the 3-, 4-, and 5-d cases, almost all ratios are above the



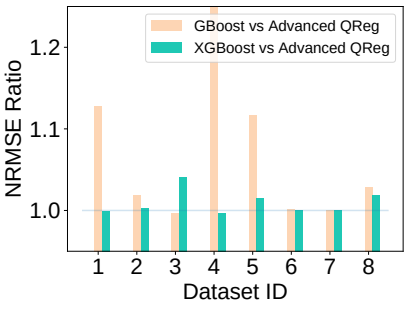
(a) 2-d



(b) 3-d



(c) 4-d



(d) 5-d

Figure 3.12: r between *Advanced QReg* and base ensemble models

horizontal line $r = 1$. It shows that models do very well for most data sets. However, there are some cases where *QReg* is significantly better than other ensemble methods, for example against GBoost for the 5-d data set 4 and against XGBoost for the 4-d data set 8. We see that *QReg* can improve accuracy across data sets and dimensionalities. As explained in Section 3.5, *QReg* is trained based on the distribution of the best model in the query space, thus QReg always selects the best model to make the prediciton for a given query.

Comparing Figure 3.12 with Figure 3.11, we see that even though the overall NRMSE of various RMs is similar, different RMs give different accuracy in different subspaces of the data. Interestingly, this figure also shows that for different data sets different ensemble methods win (as we have seen previously), showcasing the need for a method like *QReg*.

In this section, we demonstrate the performance of *QReg* from a workload-centric perspective, where we assume that the query space has the same distribution as the data space. For sub-regions where a base RM outperforms other RMs, *QReg* is shown to perform better than other RMs. This shows that *QReg* will always (or try to) select the best RM to make a prediction for each query.

3.6.3 Query-centric Perspective: Simple *QReg*

Recall the results regarding RM winning counts and associated errors in the evaluation part in Section 3.5. These results revealed that, regardless of which RM is chosen, unavoidably, for a significant fraction of queries, a sub-optimal method will be used. Adopting a query-centric perspective, our aim is to show that *QReg* does very well for the cases revealed above where unavoidably a non-optimal performer may be used. So we will be looking at cases where LR, PR, and DTR win (and later in cases where XGBoost and GBoost win). In fact we will be looking at the top 20% of queries where each RM wins, to delve into the cases where the minimum error is achieved.

Given that LR (or PR or DTR) wins, our hope for *QReg*'s performance

is to be much closer to the best performer than PR and DTR (or LR and DTR when PR wins, or LR and PR when DTR wins). In other words, we aim to minimize the "opportunity loss" for queries: that is, offer a performance that is much better than the performance if we used *QReg* than if we used any other non-optimal method. The ROL metric was created exactly for this reason. Table 3.10 captures this information.

Table 3.10: ROL w.r.t. *Simple QReg* where different simple RMs win for top 20% of queries.

Data set	where LR wins	where PR wins	where DTR wins
	23.78	261.00	9.10
1	23.92	129.00	12.00
	0.53	3.20	5.26
2	1.2	4.77	3.76
	1.26	7.78	6.25
3	1.31	4.75	3.82
	1.02	19.25	1.93
4	1.67	18.65	1.11
	1.08	1.35	3.95
5	2.02	1.87	2.81
	2.05	2.85	3.04
6	2.73	2.02	1.80
	2944.50	15.90	23.68
7	1931.00	16.12	19.36
	1.18	1.33	7.32
8	1.90	1.59	2.39

Table 3.10 focuses on the accuracy performance of *Simple QReg* vis-a-vis that of the simple RMs. We use the ROL metric, in an attempt to show how much of the opportunity loss *QReg* gains back. Each cell [i,j] ($j \in \{2, 3, 4\}$) in Table 3.10 represents performance for data set i, for the cases where RM_j is the winner, focusing only on the queries for which a top-20% error was achieved. For instance, column 2 ($j = 2$) refers to the (top 20% of) queries, among all queries where LR was the winner (among LR, PR, and DTR). The top number in cell [1,2] denotes that if PR was used, instead of (the optimal) LR, this would incur an NRMSE higher by 23.78 times, compared to the case where *QReg* was used. The bottom number in [1,2] denotes that if DTR was used instead of (the optimal) LR, this would incur an NRMSE higher by 23.92 times

compared to the case where *QReg* was used. This much better performance is reached, as *QReg* often selects the best performer (LR in this case).

Similarly, the top number in cell [4,3] shows that if LR was used instead of PR (which is the best RM for column 3), this would incur an NRMSE higher by 19.25 times compared to the case where *QReg* was used, for data set 4. The bottom number in [4,3] shows that if DTR was used instead of (the best RM) PR, this would incur an NRMSE higher by 18.65 times, compared to the case where *QReg* was used, for data set 4.

Finally, the top number in cell [3,4] denotes that if LR was used instead of (the best RM for column 4 which is) DTR, this would incur an NRMSE higher by 6.25 times, compared to the case where *QReg* was used , for data set 3. The bottom number in [3,4] denotes that if PR was used instead of (the best RM for column 4 which is) DTR, this would incur an NRMSE higher by 3.82 times, compared to the case where *QReg* was used, for data set 3.

Table 3.10 shows that indeed for 20% of the queries, for which the top error was achieved using either LR, PR, or DTR, using *QReg* would significantly reduce the opportunity loss. Notably, there is only one entry that is less than 1, showing that in all other cases, using *QReg* instead of the any other 2nd-best performer would bring about significant improvements in error for even the most vulnerable of the queried spaces.

In this section, we show the performance of Simple *QReg* from a query-centric perspective, where the workload distribution differs from the data distribution. We use relative opportunity loss (ROL) to measure the improvement made by *QReg* for not using the 2nd-best performer. Experimental result shows that using *QReg* instead of the 2nd-best performer will bring much improvement in prediction accuracy. In the next section, we will conduct the same experiments for Advanced *QReg* from a query-centric perspective.

3.6.4 Query-centric Perspective: Advanced QReg

As discussed in Section 3.6.2, we calculate the NRMSE error of *advanced QReg* for the full collection of points where a single ensemble model wins. To zoom

into the context of query-centric prediction serving, we now focus only in the top 20% of queries with the least error, as done previously. To summarize relative performance, the relative opportunity loss between RMs and *QReg* is shown in Table 3.11.

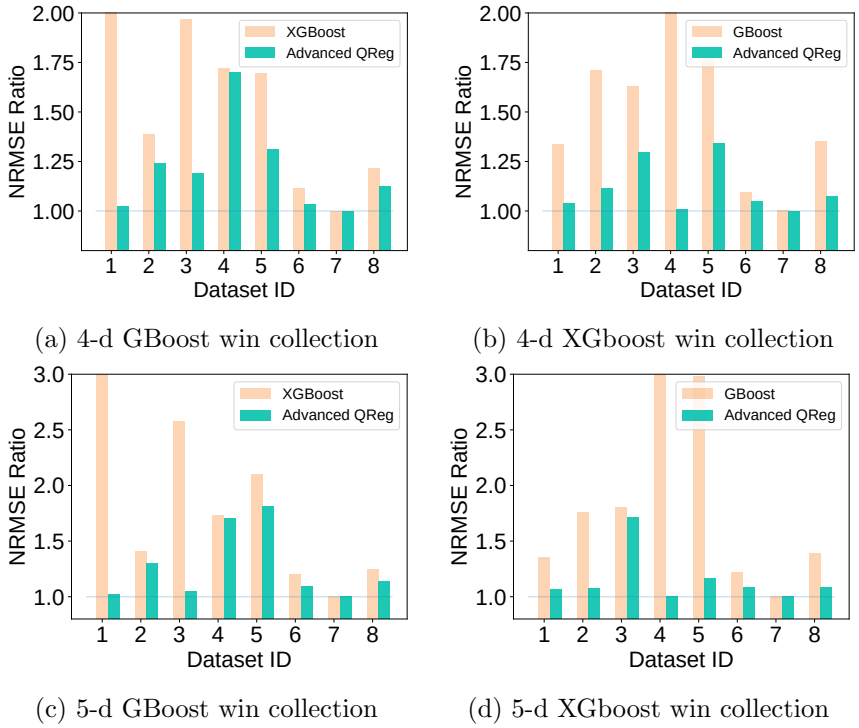
Table 3.11: ROL w.r.t. *QReg* when different ensemble RMs win for their top 20% queries.

Data set ID	where GBoost wins	where XGBoost wins
1	3.00	2.39
2	2.77	2.68
3	75.50	2.00
4	0.99	>1000
5	2.77	1.87
6	5.67	1.77
7	7.13	3.44
8	2.64	5.29

The values shown exactly are the ROL of using as a second-best RM GBoost (XGBoost) vs *QReg* when XGBoost (GBoost) wins. For example, cell [1,2]=3.00, says that if XGBoost was used (instead of the optimal in this case GBoost) it would result in an error that is 3 times higher than if *QReg* was used. In other words, previous results have shown that, regardless of which RM is chosen, this RM will be suboptimal for certain queries. So these ROL values show how *QReg* can minimize this cost when being suboptimal.

Similar to Figure 3.11, Figure 3.13 shows the NRMSE ratio for the 4-5 dimensional space, but in the query-centric (the top 20% of queries) perspective. Focus on the collection of points where XGBoost has the best prediction accuracy in the 5-d data set 8 as an example. The NRMSE ratio between GBoost (second best RM) and XGBoost is 1.3842, while the NRMSE ratio between *QReg* and XGBoost is 1.0872. Thus, the relative opportunity loss is $0.3842/0.0872 = 4.4$, which means the error caused by using GBoost is 4.4 times as the error caused by *QReg* for the collection of points where XGBoost has the best prediction accuracy.

It is noticeable that the NRMSE ratio from *QReg* is always less than that from the second best model, and is very close to 1. Thus, for the most



(a) 4-d GBoost win collection

(b) 4-d XGboost win collection

(c) 5-d GBoost win collection

(d) 5-d XGboost win collection

Figure 3.13: Query-centric collection-level NRMSE ratio

vulnerable queried spaces (where a single ensemble model wins by far), *QReg* can achieve almost the same accuracy, reconciling the otherwise unavoidable loss.

Again, even from a query-centric perspective, Advanced *QReg* achieves much improvement in prediction accuracy for not using the second-best RM. By using *QReg*, a user avoids the risks of using ensembles for datasets where they will perform badly.

3.6.5 Analysis of the QReg Classifier

Classification accuracy is defined as the proportion of correct decisions the classifier made with respect to chosing the best of the base models.

For the 3-d Bejing PM2.5 problem, Table 3.12 summarizes the NRMSEs of base models and *Simple QReg*, and the SVM RBF classifier is used. The $NRMSE_{QReg}$ achieved using SVM-RBF is 8.12%. This is better than that of any base model. This is a result to extend Section 3.6 in that the prediction accuracy of *QReg* is not bounded by that of the single best model.

Table 3.12: Comparison of NRMSEs.

LR	**PR**	**DTR**	**optimal**	***QReg***
8.49%	8.84%	8.23%	7.19%	8.12%

When each data point is predicted by the local best model, the optimal NRMSE (7.19%) is achieved. Notably, there exists a marked difference between the optimal NRMSE and the $NRMSE_{QReg}$. However, first note that this is achieved with not a state of the art classifier, SVM-RBF, which was correct in only 57% of classifications. So, even with a classifier with fairly low accuracy, *QReg* improves accuracy overall. If the accuracy of the classifier can be increased, then overall accuracy will increase further.

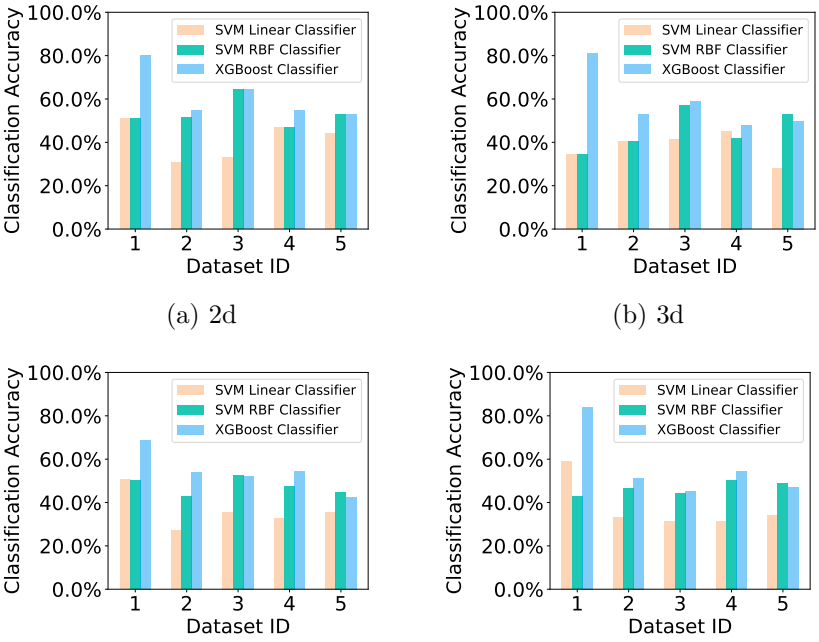
Classification Accuracy

Given the above, we now study three different popular classifiers, namely, SVM-linear, SVM-RBF and a state of the art ensemble classifier, XGBoost and observe their affect on overall *QReg* accuracy. This will shed light as to how much of an impact the classifier selection can have. We focus on the Beijing PM2.5 data set.

Figure 3.14 shows the dimensionality influence, as it varies from 2 to 5 , and the classification accuracy of the three classifiers. The XGBoost classifier always has the highest classification accuracy for data sets 1,2,3 and 4. For data set 5, SVM-RBF achieves (slightly) better classification accuracy than that of XGBoost. SVM linear classifier has in general the lowest classification accuracy. Although there is not a single classifier that wins across all data sets, XGBoost and SVM-RBF classifiers perform distinctively better than SVM-linear classifier and XGBoost is the desirable classifier, across dimensionality values.

Classifier Training Time

Nonetheless, classification accuracy does not tell the whole story. Figure 3.15(a) shows training time vs number of points for the SVM-linear, SVM-RBF, and

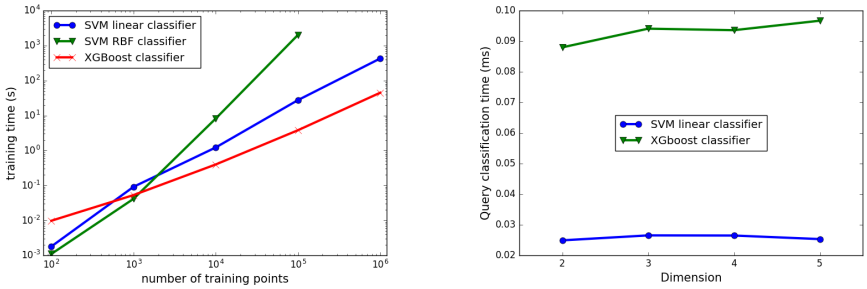


(a) 2d (b) 3d

(c) 4d (d) 5d

Figure 3.14: Classification accuracy for various dimensions

XGBoost classifiers. Times are shown to be approximately linear to the number of training points for SVM-linear and XGBoost. The training time for SVM-RBF is much higher than the other two classifiers and it grows much faster. With ca. 100k points, it takes about 34 minutes to train SVM-RBF. In contrast, when the number of training points increases to a million, the training time of SVM-linear is 427s, while for XGBoost is only 45s. Therefore, although SVM-RBF enjoys high accuracy, its training time becomes prohibitive with larger data sets.



(a) Training time of classifiers (b) Influence of dimension on classification

Figure 3.15: Comparison of typical classifiers

Classification Time

Query classification time refers to the time spent on the classifier for each query to be assigned to a regression model. Figure 3.15(b) shows that the influence of dimensionality on query classification time is negligible. XGBoost takes a longer time (0.093ms) to classify each query than SVM-linear (0.026ms). Given that the overall query execution time is ca. 0.42ms, the classification times represent ca. 23% and ca. 6% of the overall query execution time.

To summarize this section, we analyze the performance of various classifiers in three aspects: classification accuracy, model training time, and model inference time. In terms of classification accuracy, the XGBoost classifier and SVM-RBF enjoy better accuracy. In addition, as the XGBoost classifier requires less training time for large datasets, we will hereafter study *QReg* using the XGBoost classifier.

3.7 QReg Scalability

3.7.1 QReg Training Time

Figure 3.16 shows the results of our study focusing on the scalability of *QReg*, seeing the performance overheads that need be paid for *QReg*'s accuracy improvement. There exists an approximately linear relationship between the

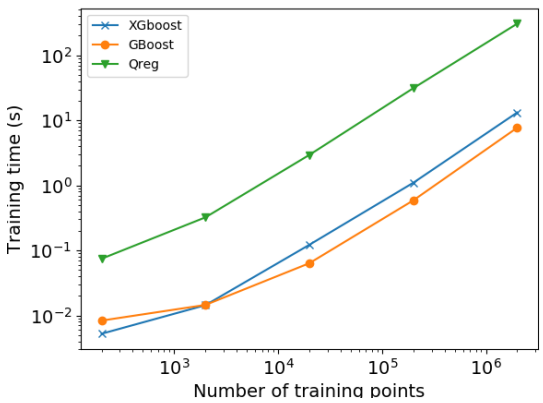


Figure 3.16: Comparison of model training time

model training time and number of training points. Even for a relatively

high number of training points, (e.g., hundreds of thousands), the training time for *QReg* is shown to be a few dozen of seconds. Although this is an order of magnitude worse than XGBoost in absolute value it is acceptable for medium-sized data sets. Also, about 90% of the training time is spent for getting predictions from the individual base models. In the current version of the code, predictions are received sequentially from base models; doing this in parallel, would reduce the total training time. In contract, in boosting methods, the weak learners (models) are trained sequentially in a very adaptive way, and it is not possible to train them in parallel. For bagging methods, the weak learners are trained independently from each other in parallel and are further combined following some kind of deterministic averaging process. So bagging methods are easily parallelizable.

3.7.2 Query Response Time

We now study the query response time overhead inherent in *QReg*. Figure 3.17 summarizes the *QReg* query time for eight data sets. Results are shown for the 4-d case and are indicative of all cases.

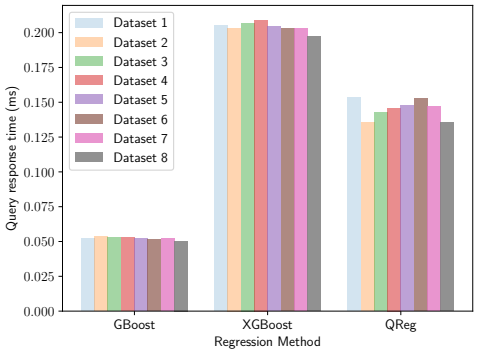


Figure 3.17: Comparison of query response time

GBoost takes the least time, ca. 0.05 ms. XGBoost takes up to 0.20ms. *QReg* falls in between these two: ca. 0.14ms. The query response time of QReg consists of two parts: (a) classification time from the classifier and (b) prediction time from one of its base models. As a consequence, the query response time of QReg typically lies in between the query response time of its

base models.

3.7.3 Sample Size Planning

As discussed in section 3.7.1, the total training time of *QReg* increases approximately linearly as the data size increases. This limits its application to very large data sets. For instance, for large data sets consiting of billions of data points, building a single regression model will be prohibitively expensive. An approach for addressing this issue is to build samples from the data and train *QReg* on the samples. We study the implications of this approach on *QReg*'s performance and observe also whether our findings hold for this case as well.

One major question is how big the sample size should be? A smaller sample requires less training time, but might lead to poor accuracy. According to the tasks, various strategies could be used to determine the sample size. For general purposes, Cochran's formula [33] is usually used to determine the sample size for a population.

$$n_0 = \frac{z^2 p(1-p)}{e^2} \tag{3.5}$$

where n_0 is the sample size, z is the selected critical value of desired confidence level, p is the degree of variability and e is the desired level of precision. For instance, we need to determine the sample size of a large population whose degree of variability is unknown. $p = 0.5$ indicates maximum variability, and produces a conservative sample size. Assume we need 95% confidence interval with 1% precision, the corresponding sample size $n_0 = 9604$. For datasets with a finite size, the sample size is slightly smaller than the value obtained in eq. (3.5).

For regression-specific tasks, sample size planning techniques include power analysis (PA) [34], accuracy in parameter estimation (AIPE) [89], etc. The sample sizes obtained from both methods are different, and the magnitude is usually hundreds or thousands. [84] proposes a method to combine these methods with a specified probability, while [109] recommends that the largest sample size should be used.

For classification-specific tasks, [48] finds that many prediction problems do not require a large training set for classifier training. [17] uses learning curves to analyze the classification performance as a function of the training sample size, and concludes that 5-25 independent samples per class are enough to train classification models with acceptable performance. Also, 75-100 samples will be adequate for testing purposes.

In this study, the sample size varies from 10k, 100k to 1m, which are conservative compared to the values obtained by the PA and AIPE methods for regression tasks, or the size for classification tasks.

3.7.4 Workload-centric Perspective

We show results for data sets 6, 7, 8 and Table `store_sales` from the TPC-DS data set. Data sets 6, 7, 8 contain 2-4 million records, and Table `store_sales` is scaled-up to 2.6 billion records. We use reservoir sampling to generate uniform random samples for these data sets. Experiments are done using *Advanced QReg*.

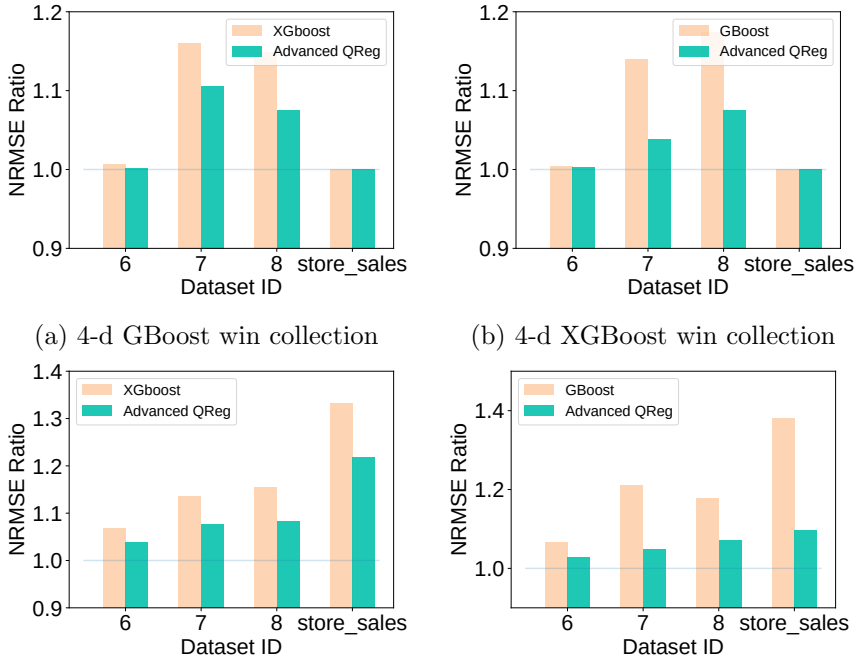
Table 3.13: Win counts of ensemble RMs.

Data set ID	Count of GBoost	Count of XGBoost
6	16209	17124
7	15854	17479
8	13757	19576
store_sales	13415	17003

Table 3.13 shows the occurrences of best predictions (wins) made by each model, for the samples of size 100k. Similarly to Table 3.3 in section 3.5, each base model is shown to win for a substantial percentage of queries (or, equivalently for a considerable part of the data set). This supports Section 3.5 that there is not a single regression model capable of dealing with various data sets, and each regression model is only good at sub-spaces of the data sets.

Similar to Section 3.6.2, this section focuses on the workload-centric evaluation but for sample-based *QReg*. We show the NRMSE ratio r between XGBoost (or GBoost) and *QReg*, broken down to subcollections of points in

the data space, specifically, for the subcollection of points where GBoost (or XGBoost regression) has the best accuracy.



(a) 4-d GBoost win collection

(b) 4-d XGBoost win collection

(c) 5-d GBoost win collection

(d) 5-d XGBoost win collection

Figure 3.18: Workload-centric collection-level NRMSE ratio

Consider the collection of points where XGBoost has the best prediction accuracy in the 5-d data set 7. The NRMSE ratio r between GBoost regression (second best RM) and XGBoost regression (best RM) is 1.2107, while the NRMSE ratio between $QReg$ and XGBoost regression is 1.0499. Thus, the corresponding ROL between GBoost and $QReg$ is 0.2107 /0.0499 = 4.22, which means for this collection of points, GBoost induces 4.22 times higher error than $QReg$.

The same conclusion holds for the query-centric perspective, and is omitted for space reasons.

3.7.5 Model Training Time

The training time of sample-based $QReg$ consists of two parts: (a) **Sampling time** to generate samples from the base tables; (b) **Training time** to train $QReg$ over the samples. Figure 3.19 shows the training time of $QReg$ for the

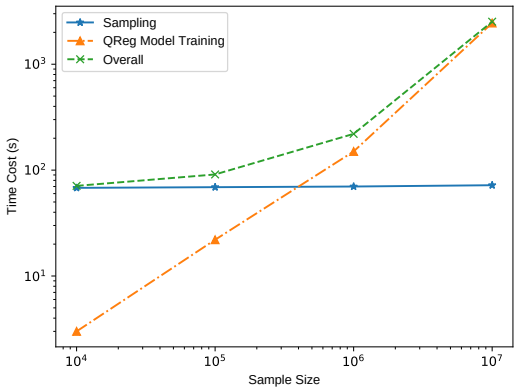


Figure 3.19: Sample Size vs Training Time for store_sales

100m `store_sales` table, while sample sizes vary {10k, 100k, 1m}. It takes ca. 68-72s to generate the samples. For 10k (100k, 1m) samples, it takes less than 3s (22s, 150s) to train *QReg*. We also trained QReg on a 10m sample, and it takes ca. 24 minutes to train the model. With 100k samples, *QReg* performs excellently. So, in conclusion, sample-based *QReg* is scalable and enjoys high accuracy and lower training time as samples could be used to train the model.

3.7.6 Application to AQP engines.

The previous experiments demonstrate the strength and potential of *QReg*. In this section, *QReg* is applied to a real-world, state of the art approximate query processing engine, DBEst. (which will be introduced in Chapter 4). DBEst uses regression-based models to approximate answers to aggregation queries[104]. We replace the default regression model in DBEst (XGBoost) with *Advanced QReg*, and compare the accuracy with DBEst using other ensemble methods, including XGBoost and GBoost. The detailed description of the experimental setup is introduced in Section 3.4.2.

Figure 3.20 shows the relative error achieved by DBEst using various regression models. For `SUM`, the relative errors using XGboost or GBoost are 8.35% and 8.10%. However, if *Advanced QReg* is used, the relative error drops to 7.77%. Although *Advanced QReg* is build upon XGBoost and GBoost, the relative error of DBEst using *Advanced QReg* is better than DBEst using

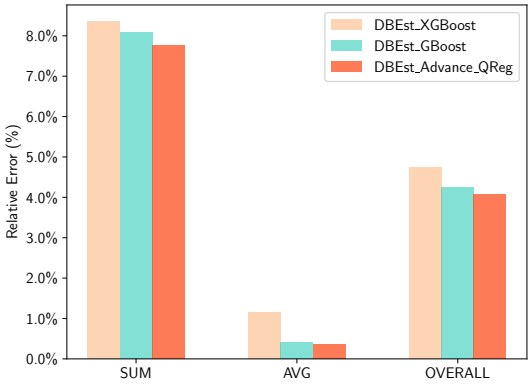


Figure 3.20: Application of QReg to DBEst for TPC-DS dataset

XGBoost or GBoost only. For further comparison, if the linear regression is used in DBEst, the relative error becomes 21.20%, which is much higher than DBEst using *Advanced QReg*.

We further examine the performance of *Advanced QReg* in DBEst for data set 3 (the Beijing PM2.5 data set). 48 `SUM` and `AVG` queries are randomly generated for three column pairs: [TEMP, pm25], [DEWP, pm25] and [PRES, pm25]. The sample size is set to 100k.

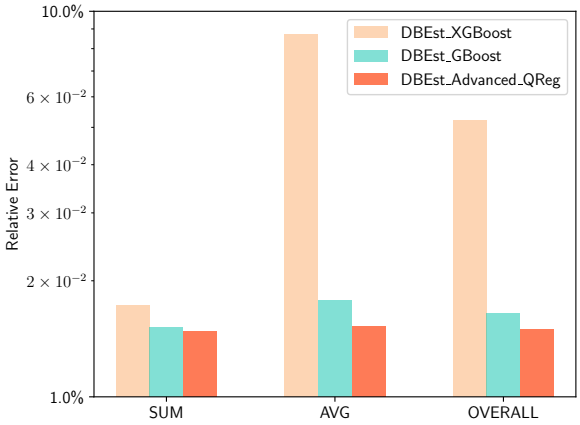


Figure 3.21: Application of QReg to DBEst for Beijing PM2.5 data set

Figure 3.21 compares the relative error obtained by DBEst using XGBoost, GBoost and *Advanced QReg*. For `AVG` queries, the relative error is 8.73% (1.78%) if XGBoost (GBoost) is used in DBEst. If *Advanced QReg* is applied, the relative error drops to 1.52%, which is a significant reduction in error compared to other models. For `SUM` queries, *Advanced QReg* also achieves the

least error.

In this section, we apply *QReg* to DBEst, a model-based AQP engine, which will be introduced in Chapter 4. Experimental results show that replacing XGBoost regression with *QReg* reduces the prediction error significantly, at the cost of slightly increased inference time.

3.8 Major Lessons Learned

The key lessons learned by this study are:

- Different RMs are better-performing for different data sets and, more interestingly, for different data subspaces within them. This holds for simpler models and, perhaps surprisingly, for advanced ensemble RMs, which are designed to generalize better.
- Each examined RM is best-performing (a winner) for a significant percentage of all queries. Necessarily, this implies that, for a significant percentage of queries, regardless of which (simple or ensemble) RM is chosen by a DB user/analyst, a suboptimal RM will be used.
- When said suboptimal RMs are used, significant additional errors emerge for a large percentage of queries.
- Best practice, which suggests to an analyst to use a top-performing ensemble, is misleading and leads to significant errors for large numbers of queries. In several cases, despite the fact that different RMs had a very similar overall error (NRMSE), a significant fraction of queries face very large differences in error when using seemingly-similarly-performing RMs. Thus, both sophisticated and simpler RMs cannot cope well, in order to appease query-sensitive scenarios, where query distributions may target specific data subspaces.
- A query-centric perspective, as manifested with *QReg*, can offer higher accuracy across data sets and dimensionalities. This applies to overall

NRMSEs. More importantly, it applies to query-centric evaluations. The study revealed that when *QReg* is used, there are significant accuracy gains, compared to using any other non-optimal RM (which as mentioned is unavoidable).

- Accuracy improvements are achieved with small overheads, even with very large data sizes, using sampling.

3.9 Summary

This chapter studied issues pertaining to the seamless integration of DBMSs and regression models. The analysis revealed the complexity of the problem of choosing an appropriate regression model: Different models, despite having overall very similar accuracy, are shown to offer largely-varying accuracy for different data sets and for different subsets of the same data set. Given this, the analysis sheds light on solutions to the problem. It showed and studied in detail the performance of *QReg*, which can achieve both high accuracy over the whole data set and top-notch accuracy, per query targeting specific data subsets. The analysis also showed the impact of key decisions en route to *QReg*, such as selecting different constituent base regression models. In addition, it studied issues pertaining to scalability, showing that even with large data sets, the same issues hold and the same model solution can be used to achieve per-query and overall high accuracy. In general, the proposed *QReg* offers a promising approach for taming the generalization-overfit dilemma when employing ML models within DBMSs. Based on the key findings from this chapter, we will propose a model-based approximate query processing engine, and it will be introduced in the next chapter.

Chapter 4

DBEst: A Model-Based AQP Engine

In the previous chapter, we investigate the performance of typical regression models for various datasets and even for different regions of the data space. We find that it is misleading to always select the best performer for all datasets. After that, we propose an ensemble method, coined $QReg$, which enjoys good accuracy, even for different regions of the data space. In this chapter, we will introduce a model-based approximate query processing (AQP) engine. It is based on regression models and density estimators to provide approximate answers for SQL queries.

4.1 Motivations

The state of the art in AQP research has been dominated by sampling-based approaches, broadly divided into two categories. First, techniques that rely on online sampling, create samples on the fly during query execution and use them to approximate answers. The second category of research, exploits the fact that often query workloads are (at least partially) predictable, in the sense that one can know beforehand the popular query templates, including for example the attributes for range predicates, the joined tables and join keys, the grouping attributes, used together. Given this knowledge, these works

create offline samples for selected tables and column sets, kept in memory, and process queries over said samples. But, across the spectrum, the state of the art still suffers from several shortcomings.

Major Challenges

Typically, the performance of AQP engines is evaluated from the following three technical aspects: 1.) query response time, 2.) space overheads, 3.) prediction accuracy. In addition, the state building time and the types of supported queries are also considered. Take the Conviva [6] data with the size of 7.5 TB as an example. It takes HIVE more than 2000 seconds to produce the query results, while a cluster of 100 machines is used. If an AQP engine, say BlinkDB, is used with a 1% error bound, the corresponding query response time reduces to 10s. A large sample is usually generated and maintained to guarantee high accuracy. Typically, the size of a sample is at least 1% of the actual data. And a 10% sample means 10% more investment in storage. In terms of query response time, we wish to reduce the it to less than 1 second without the usage of a big cluster.

This chapter is driven by these questions, the answers to which currently leave a lot to be desired. We revisit the problem and solution space. **The overriding guiding principle is to develop and study a model-driven solution, instead of a data-driven solution, where queries are answered by models of data and not the data itself (or samples of it)**. The principal challenges and goals are to develop and experimentally substantiate such a model-based AQP engine that is much more efficient, ensures high accuracy, and investigate its benefits and limitations. **The key insights of this work is to exploit the ability of models to generalize. This affords the luxury of building the models from very small samples**. Combined, these facts ensure small overheads, with shorter response times, even with just a single-thread, thus rendering analytics less costly and achieving much higher system throughput.

This chapter will present the design and implementation of DBEst, an AQP

engine supporting the aforementioned analytical needs, using prebuilt, a priori state (i.e., models over samples of datasets). Specifically, this chapter will:

- show how to develop and train appropriate models and how to use them to answer analytical queries.
- analyze the sensitivity of and stress DBEst's performance on key parameters, such as sample sizes used to derive the models, the selectivity of selection operators, the effect of groups in Group By, the effect of joins, etc.
- perform a comprehensive performance evaluation of DBEst, comparing it against state of the art big data AQP engines, (BlinkDB and VerdictDB), using queries based on the TPC-DS queries and its schema/data and synthetic queries over real-life UCI-ML repository datasets.
- evaluate single-threaded and multi-core DBEst performance and show that even the sequential DBEst can often achieve large (>10x) query processing speedups against a 12-core state of art AQP engine and that this can help achieve 10x to 30x speedups in system throughput.

To our knowledge, this is the first AQP engine based on combining sampling, density estimators, and regression models. DBEst can offer big gains across all metrics of interest. The chapter also takes a qualitative step forward: its models can be employed for various other analytical needs: (i) imputing missing attribute values; (ii) estimating the value of a dependent variable when values for the independent variables are missing or hypothesized, (iii) estimating the value of aggregation functions over the dependent variable, when independent-variable values are missing or hypothesized, (iv) quickly discovering relationships between attributes, (v) quickly visualizing descriptive statistics for the dependent attribute in data subspaces, etc. In general, DBEst provides support for predictive analytics, hypotheses testing, etc.

4.2 The DBEst AQP Engine

4.2.1 System Overview

Figure 4.1 shows the architecture of DBEst. DBEst is independent of the underlying storage layer; it can be just a local FS, an RDBMS, or a distributed FS, and/or a NoSQL DB.

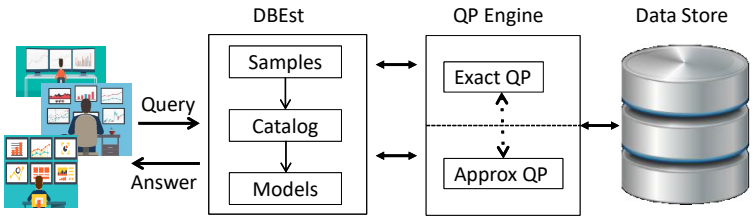


Figure 4.1: DBEst architecture.

There are three major components: (1) The sampling module interacts with the storage layer to build samples; (2) The models module stores density estimators and regression models, which are built from the samples. (3) The model catalog stores information for the available models and their correspondence to the column sets and tables of the base data they model.

To build the model for a specific query template, DBEst will firstly make samples over the tables (stored locally or distributed in the backend server). And the reservoir sampling is applied to build uniform samples efficiently. After that, DBEst builds the corresponding density estimator and regression model for this column pair. And the model catalog will be notified of the building of new models. To serve a query, DBEst reads the model catalog to check for models that could answer it. If there exists such a model for this query, the particular models will be fetched from the model catalog to process the query. If not, the query will be sent to the backend server in the level below it, as shown in the architecture. This can be another AQP engine (e.g., one with online sampling, QuickR [87]) or it can go directly to an exact answer QP engine. Sometimes, even though there exists a model to serve the query, the user might not be satisfied with the prediction accuracy. For such cases, DBEst allows the user to interact with the backend server directly.

4.2.2 Supported Queries

DBEst belongs in the class of AQP for predictable/popular query templates. In this class, as mentioned, DBEst supports analytical queries, involving range predicates and aggregate functions, including `COUNT, SUM, AVG, VARIANCE, STDDEV and PERCENTILE`. DBEst also supports `GROUP BY` operators. To be concrete, the following is typical in queries in TPC-DS [114]. Given a table `store_sales`, with the column of interest, `ss_quantity`, return the average value of `ss_ext_discount_ amt` within a specific range.

```
SELECT ss_store_sk, SUM(ss_sales_price)
FROM store_sales
WHERE ss_sold_date_sk BETWEEN lb AND ub
GROUP BY ss_store_sk;
```

returns the sum of `ss_sales_price` within the given range of `ss_sold_date_sk`, for each `ss_store_sk` group value.

Thus, DBEst supports straightforwardly selections on numerical and ordinal categorical attributes for all AFs mentioned above, optionally coupled with `GROUP BY` operators. DBEst can also provide support for selections on nominal categorical attributes, as will be discussed later.

DBEst does not support ad hoc join queries with no prebuilt models. In these cases, DBEst will revert to the underlying AQP engine (e.g., QuickR [87], or VerdictDB [124]).

DBEst supports joins for predictable/popular joined tables using two alternative approaches. The first approach follows the following steps: First, precompute the join result table, then build a (small) sample over it, and finally build regression and density estimator models over this sample. Please note that this is particularly possible for DBEst, as neither the original join result nor any large sample of it need be maintained - both join result and the sample can simply be discarded after models are built. Only the models need be stored and used during query processing. And models are very small in size (typically a few 100s KBs). This is in contrast to the state of the art AQP

engines [87, 124] which must compute the join based on (universe/hashed) per-table samples (each having typically 10s of millions of tuples) online during query execution. The performance evaluation section will provide details and quantify the resulting benefits.

The second approach improves on the precomputation time when joining very large tables. Specifically, each large join table can be sampled (using hashed samples) and the join be computed based on these samples (a la VerdictDB and QuickR). Finally, a small uniform sample is built from the sample join and models are built from this small sample.

DBEst does not support general nested queries. When nested queries can be 'flattened' using joins, (as done in VerdictDB [124]) then DBEst can support these nested queries using the above explained support for joins. Finally, DBEst does not currently support UDFs.

4.2.3 DBEst Query Processing Foundations

In contrast to competing AQP approaches where samples are generated and maintained to answer queries, DBEst chooses an alternative approach. It builds models, specifically regression models and density estimators, through which aggregate queries are answered with high accuracy and efficiency and at lower costs. In this section, we describe in detail the mathematical foundations for providing approximate answers for analytical queries.

DBEst uses the density estimator and/or the regression model to compute the AF answer. Based on whether a regression model is involved in making the approximate prediction, the supported aggregate functions are divided into two categories: *density-based* and *regression-based* . For density-based aggregate functions, only the density estimator D(x) is needed to make the prediction; for regression-based aggregate functions, both the density estimator D(x) and the regression model R(x) are involved. Table 4.1 contains the notation used in this section.

We now discuss how each aggregate is processed in DBEst. The PERCENTILE AF has a syntax a la HIVE, which is:

Notation	Description
T	original table
Q	a query
AF	a supported aggregate function, is one of COUNT, SUM, AVG, VARIANCE, STDDEV, PERCENTILE
x	the independent variable (column), usually accompanied with a condition.
y	the dependent variable (column) in the query, which is the aggregate attribute
p	the p^{th} percentile point for a `PERCENTILE` query.
CP	a unique column pair, consisting of x and y
N	the size of Table T
n	the size of the sample
S(CP,n)	the sample, for column pair CP, with the sample size of n
lb	the lower bound of x for the aggregate query
ub	the high bound of x for the aggregate query
R(x)	the regression model of x, training from [x,y] pairs.
D(x)	the density estimator over column x, which is normalized to unity.

Table 4.1: Notation in Section 4.2

```
SELECT  PERCENTILE(x, p) FROM T;
```

which returns an approximate p^{th} percentile of the numeric column `x` for Table `T`.

For regression-based aggregate functions, as a regression model $R(x)$ is built between `y` and `x`, $R(x)$ is used to provide an approximate answer for y. DBEst could answer two kinds of VARIANCE AFs: regression-based and density-based. Density-based VARIANCE AFs take the following general form:

```
SELECT  VARIANCE(x) FROM T
WHERE x BETWEEN lb AND ub;
```

where only (column) x is involved in the query. Regression-based VARIANCE queries take the following form:

```
SELECT  VARIANCE(y) FROM T
WHERE x BETWEEN lb AND ub;
```

having both independent and dependent variables. These require both the density estimator and the regression model.

Computing Aggregates with Density Estimators

Density-Based Aggregate Functions include `COUNT, VARIANCE, STDDEV` and `PERCENTILE`.

COUNT

Formally:

$$COUNT(y) \approx N \cdot \int_{lb}^{ub} D(x)dx \tag{4.1}$$

The integral of the density estimator is evaluated in interval given in the range selection operator, i.e., $\int_{lb}^{ub} D(x)dx$, yielding the proportion of data points that lie within this range. N (the size of the table), scales up $\int_{lb}^{ub} D(x)dx$ to be an approximate representation of the total number of points in this range.

VARIANCE and STDDEV

Formally:

$$\begin{aligned} VARIANCE\_x(x) &= \mathbb{E}\left[x^2\right] - [\mathbb{E}\left[x\right]]^2 \\ &= \frac{\int_{lb}^{ub} x^2 D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} xD(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2 \end{aligned} \tag{4.2}$$

$$\begin{aligned} STDDEV\_x(x) &= \sqrt{VARIANCE\_x(x)} \\ &= \sqrt{\frac{\int_{lb}^{ub} x^2 D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} xD(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2} \end{aligned} \tag{4.3}$$

By definition, the variance of x is equal to $\mathbb{E}\left[x^2\right] - [\mathbb{E}\left[x\right]]^2$. The expectation of x and x^2 could be calculated via the integrals involving the density estimator $D(x)$ as shown above.

PERCENTILE

In general, `PERCENTILE` returns the value p, for which $P(x < \alpha) = p$. Thus, given the probability density estimator $D(x)$ and the p_{th} percentile point, the problem translates to finding the value α that meets $\int_{-\infty}^{\alpha} D(x)dx = p$. Note, $\int_{-\infty}^{\alpha} d(x)dx$ is the cumulative distribution function (CDF), and is usually denoted as $F(x)$. Thus, the problem becomes finding the root for equation

$$F(x) = p \tag{4.4}$$

If the reverse of the CDF, $F^{-1}(p)$, could be obtained, then the p^{th} percentile for Column `x` is

$$\alpha = F^{-1}(p) \tag{4.5}$$

However, there is usually not a theoretical solution for $F^{-1}(p)$, and a more practical solution adopted in DBEst is to find the solution for Equation 4.4 through an iterative process, which is the Naive Bisection method for finding the root [85].

Computing Aggregates with Regression Models

Aggregates that can be computed using regression models include `SUM, AVG, VARIANCE` and `STDDEV`.

AVG

Formally:

$$\begin{aligned} AVG(y) &= \mathbb{E}[y] \\ &\approx \mathbb{E}[R(x)] \\ &= \frac{\int_{lb}^{ub} D(x)R(x)dx}{\int_{lb}^{ub} D(x)dx} \end{aligned} \tag{4.6}$$

The average value of y, or its expectation $\mathbb{E}[y]$, could be approximately treated as the expectation of $R(x)$, which is $\mathbb{E}[R(x)]$. To calculate the average

value of a continuous function $R(x)$, we only need to know its density function.

SUM

Formally:

$$\begin{aligned} SUM(y) &= COUNT(y) \cdot AVG(y) \\ &\approx COUNT(y) \cdot \mathbb{E}\left[R(x)\right] \\ &= N \cdot \int_{lb}^{ub} D(x)dx \cdot \frac{\int_{lb}^{ub} D(x)R(x)dx}{\int_{lb}^{ub} D(x)dx} \\ &= N \cdot \int_{lb}^{ub} D(x)R(x)dx \end{aligned} \tag{4.7}$$

The sum of y equals the product of the count and the average value of y. From Equation 4.1 and 4.6, we get the approximate representations of the count and average value of y: multiplying equation 4.1 by equation 4.6, we get the approximate representation of SUM(y), which is $N \cdot \int_{lb}^{ub} D(x)R(x)dx$.

`VARIANCE` **and** `STDDEV`

(Please refer to *Density-Based Aggregate Functions* for the density-based `VARIANCE` and `STDDEV` AFs). Formally:

$$\begin{aligned} VARIANCE\_y(y) &= \mathbb{E}\left[y^2\right] - \left[\mathbb{E}\left[y\right]\right]^2 \\ &\approx \mathbb{E}\left[R^2(x)\right] - \left[\mathbb{E}\left[R(x)\right]\right]^2 \\ &= \frac{\int_{lb}^{ub} R^2(x)D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} R(x)D(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2 \end{aligned} \tag{4.8}$$

$$\begin{aligned} STDDEV\_y(y) &= \sqrt{VARIANCE\_y(y)} \\ &\approx \sqrt{VARIANCE\_x(R(x))} \\ &= \sqrt{\frac{\int_{lb}^{ub} R^2(x)D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} R(x)D(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2} \end{aligned} \tag{4.9}$$

By definition, the variance of y is equal to $\mathbb{E}\left[y^2\right] - \left[\mathbb{E}\left[y\right]\right]^2$. Replacing y with $R(x)$, gives an approximation of `VARIANCE`(y).

Supporting Group By

DBEst supports `GROUP BY` queries of the form:

```
SELECT  z, AVG(y) FROM T
WHERE x BETWEEN lb AND ub
GROUP BY z;
```

DBEst's rationale is to treat each value of z as a separate data set over which to evaluate the given AF. Therefore, during sampling, a sample is recorded per each z value. Subsequently, the models are built and used per each such sample to compute the AFs, as detailed above.

Thus, given a `GROUP BY` query, DBEst will call all models built for the z values, and the predictions from all models form the result for this particular query.

Supporting Multivariate Selection Operators

So far, supported queries included a range predicate over a single attribute. The multivariate range-selection operator can be straightforwardly supported. The mathematical foundation for multivariate aggregate query processing is similar to the univariate query processing. Take `AVG` queries as an example and an aggregate query with the following form:

```
SELECT  AVG(y) FROM T
WHERE x1 BETWEEN lb1 AND ub1
AND   x2 BETWEEN lb2 AND ub2;
```

The `AVG` aggregate of y could be approximately treated as:

$$\begin{aligned} AVG(y) &= \mathbb{E}[y] \\ &\approx \mathbb{E}[R(x_1, x_2)] \\ &= \frac{\int_{lb_1}^{ub_1}\int_{lb2}^{ub2} D(x_1,x_2)R(x_1,x_2)dx_2dx_1}{\int_{lb_1}^{ub_1}\int_{lb_2}^{ub_2} D(x_1,x_2)dx_2dx_1} \end{aligned} \tag{4.10}$$

And this could be extended to higher dimensions, as well as other aggregates, following the formulas given earlier.

Supporting Categorical Attributes

For ordinal attributes the treatment is straightforward as attribute values essentially map to ordered numbers. Hence, supported queries include range predicates on such attributes. For nominal attributes there is no simple way to transfer the values to meaningful numbers. DBEst's support for nominal categorical attributes mimics the support for `GROUP BY` attributes by maintaining regression and density estimator models for each nominal value, such as store_ids, city, or classes of products in a commercial application, etc.

Limitations

We note that `GROUP BY` queries with large numbers of groups pose special challenges for DBEst. First, the number of models grows linearly with the number of groups. For instance, if there are ten `GROUP BY` values, DBEst has to train models for each of the groups. This affects overall training time. (Fortunately, this task is embarrassingly parallelizable). Likewise, this affects also query response times: Each model (one per group) needs be evaluated; again, this is embarrassingly parallelizable.

Similarly, although per-model the space savings of DBEst are very large, the required space grows linearly with the number of groups. DBEst has the following alternatives: First, to not build models when the number of groups is too large. This is inline with what VerdictDB does for "large cardinality" groups, reverting to an exact answer QP engine for such queries. Admittedly, alas, the problem for DBEst is more serious. Second, to 'sacrifice' DBEst's space savings in order to just enjoy the large speedups when processing queries over the models instead of on (sampled) data.

Even further, DBEst can store models for queries having very large group cardinalities in an SSD. We have implemented this creating *model bundles*, each of which bundles all the models needed by a query with a large number of groups. Concretely, consider a query that requires a join of a (10m-row sample of a) large fact table with a small dimension table and 500 of groups

(models). Serializing this bundle of 500 models amounts to 97MBs. Reading from the SSD and deserializing such a bundle takes <132ms. Added to the ca. 600ms needed to process this join with `GROUP BY` query in DBEst gives a total of <800ms response time. To put this in context, VerdictDB requires ca. 8secs for such a query, a speedup of 10x.

Small groups pose additional limitations. Specifically, building models over small groups is an overkill; it is preferable to just keep and process the small number of tuples in the group. This is inline with what state of the art AQP engines: they do not build samples over small tables. Even QuickR [87] which builds samples online, discovered that a 25% of all queries in TPC-DS cannot be supported due to groups not having enough support.

Finally, unlike sampling-based AQP engines, DBEst currently does not provide a priori error guarantees.

4.3 Implementation

The code of DBEst is available at `https://github.com/qingzma/DBEstPy`. It is written in python with more than 13,000 lines of code. To train the models for a specific query, as shown in Figure 4.1, a sample is firstly generated for every column set of interest. After that, the sample is used to train a regression model and a density estimator, which are in turn then used to answer analytical queries on this column set.

4.3.1 Sampling

Stratified sampling [102] is usually the first choice when we try to filter or group data. It avoids the difficulties when dealing with rare groups and highly skewed data distributions. However, it also increases the difficulty if we try to build a regression model or a density estimator over a stratified sample. DBEst relies solely on reservoir sampling [155] to generate uniform samples over the original table. Different `GROUP BY` values are recorded from the original table during the training process, and they are further used to check whether any

group is underrepresented in the samples. Our experiments show that this suffices to provide excellent performance with respect to accuracy and efficiency for all AFs and for `GROUP BY`, across all tested data sets.

As DBEst is a model-based AQP engine, any samples it builds are deleted after model training. Thus, DBEst significantly reduces memory requirements, as its models are significantly smaller than the samples. Also, as accuracy depends on sample sizes, this indirectly affords the opportunity to use larger samples for training models.

4.3.2 Density Estimator

There are many existing density estimation methods, including the kernel estimator, the nearest neighbor method, the variable kernel method, orthogonal series estimators, etc [147]. Histograms are the simplest form of density estimators and have enjoyed a prominent role in DBs [5] for enhancing query processing performance. However, their discrete nature is at odds with the continuous-function view employed within DBEst. Therefore, the kernel density estimation method is chosen as the density estimator in DBEst as it has been found to be highly accurate and efficient.

The density estimation implementation is based on `sklearn.neighbors.KernelDensity` from the scikit-learn package [126], which uses the Ball Tree or KD Tree. In addition, kernel density estimation can be performed in any number of dimensions, allowing DBEst to extend its support for multivariate query processing.

4.3.3 Regression Model Selection

High performance regression models include XGBoost [30], catboost [130], LightGBM [88], gradient boosting (GBoost) [61], etc. DBEst resorted to boosted regression tree models since its models must be powerful so to generalize as they are built from small samples.

We used standard scikit-learn packages (GridSearchCV) to tune the models using cross-validation. Note that as samples increase, the regression tree models

use deeper and more trees. However, the choice of an appropriate regression model is complex: Different models work better for different data regions. Our implementations have used various regression models from piece-wise linear models to XGBoost, and GBoost and also built an ensemble regressor based on XGBoost and GBoost. First, each model is trained separately. Subsequently, the accuracy performance of each of these models is evaluated, using random queries over the independent attribute's domain. This evaluation data was then used to train a classifier, which learned which of the constituent regressors is best for a given range predicate.The XGBoost classifier was used for this purpose. To shed light into the impact of the regression model, our evaluation section provides more details for the related times-accuracy-speed trade-offs.

4.3.4 Selecting which Models to Build

This is a generic problem faced by all related efforts in AQP, that build, a priori state (samples, sketches, or ML models, as we do) for popular/predictable queries. Approaches range from trying all combinations for column sets (e.g. [2, 29]), mining query logs, like BlinkDB [6] which showed that interesting column sets can be identified early in the execution of a typical workload, or depending on users, like VerdictDB [123], to identify popular tables, etc. DBEst is rather orthogonal to this - any of the above approaches can be used. All experiments assume knowledge of said column sets of interest, given which DBEst builds samples, models, and evaluates queries.

4.3.5 Integral Evaluation

The efficiency of the integral evaluation implementation has a great impact on the performance of DBEst, with interesting accuracy-efficiency trade-offs. Fortunately, this is a well-studied problem. The integral evaluation package adopted in DBEst comes from the `integrate` module in `SciPy` [85], which uses a technique from the Fortran library `QUADPACK` [127]. The underlying Gauss-Kronrod quadrature sums are fundamental to many of the automatic

subroutines in `QUADPACK`. Given an integral

$$I_w[lb, ub]f = \int_{lb}^{ub} w(x)f(x)dx \tag{4.11}$$

over an interval [lb, ub], where `w(.)` is a weight function, then a quadrature sum yields an approximation

$$I_w[lb, ub]f \approx \sum_{k=1}^{n} w_k f(x_k) \tag{4.12}$$

In Equation 4.12, the numbers $x_1, x_2, ..., x_n$ are nodes, and $w_1, w_2, ..., w_n$ are weights corresponding to these nodes. To calculate a numerical approximation for the integral problem within absolute accuracy ϵ_a or a relative accuracy ϵ_r, `QUADPACK` computes the sequences

$$\{R_{n_k}, E_{n_k}\}, k = 1, 2, ..., N \tag{4.13}$$

Where R_{n_k} is an estimation to the integral value, and E_{n_k} is an error estimation at the iteration step n_k. `QUADPACK` chooses an adaptive approach that the position of the integration points of the n_{th} iterate depends on the information gathered from iterate $1, ..., n-1$.

4.3.6 Parallel/Distributed Computation

Much of DBEst's internal functioning is embarrassingly parallelizable and can be performed on centralized data nodes or on clusters of data nodes within big data analytics stacks. First, sampling is easily parallelizable, as different nodes storing dataset partitions can independently participate in the sampling process. Secondly, model training can be performed in parallel. And, for models supporting `GROUP BY` queries, samples for each group can be distributed and model training can occur in parallel. As mentioned, the chosen regression model is an ensemble, consisting currently of two different regression models (gradient boosting and XGBoost). Each of these can be trained in parallel. In fact, several open-source packages are available. For the parallel implementation of

DBEst we have used these packages.

Thirdly, query processing can easily be parallel. Alternatively, DBEst query execution can remain sequential and additional nodes/cores in the system can be utilized to process other queries, improving significantly system throughput. Our evaluation section will quantify such savings.

For `GROUP BY` queries, evaluation of the models of the different group attribute values can be done in parallel. Our implementation for this feature is currently suboptimal: (i) as it is Python-based, we ran into the Global Interpreter Lock problem (only 1 thread can use the interpreter at a time) and the fix we implemented (based on using multiple separate processes is suboptimal); (ii) the packages we use for model evaluation and integral computation are amenable to parallel execution but currently we have implemented no control to orchestrate overall core/node assignment to tasks. As a result, these subtasks conflict with each other for resources. Despite this, our results show that parallel DBEst can achieve speedups per query which can be >10x faster for queries involving joins with or without `GROUP BY`.

Actually, a key goal is to avoid relying on big data clusters or multi-core/node use during query execution as much as possible. And as we shall see, sequential DBEst, even for large data sets, often outperforms multi-core VerdictDB.

4.4 Performance Evaluation

We have evaluated DBEst using queries using column sets queried in the TPC-DS queries and its schema/data and synthetic queries over real-life UCI-ML repository datasets. Additionally, as we wanted to study the sensitivity of DBEst on key parameters (selectivity of predicates, sample sizes, supported AFs, etc.) we used synthetic queries over selected column sets from TPC-DS. The above allows us to systematically study separately the effects of `GROUP BY` and join operations, as well as the impact of using multiple cores/nodes on DBEst and competing solutions. Finally, in addition, we have experimented

with a few complex queries as found in TPC-DS for stress-testing purposes.

Experiments ran on an Ubuntu 18.04 Server with 12 Intel Xeon X5650 cores, 64GB RAM and 4TB of SSD disk space.

4.4.1 Experimental Setup

TPC-DS Workload

We used scale factors 40-1000, resulting in the largest table having ≈2.6 Billion rows and >1TB of data. The queries involved 16 column pairs from 4 tables. We performed 5 experiments: a.) *Multi-column-pair analysis* contains ≈100 SELECT-FROM-WHERE queries with a range predicate on one attribute and an AF on another, involving all 16 column pairs. b.) *Sensitivity analysis* consists of 1,000 queries, measuring performance under various AFs, varying query ranges, and sample sizes. The column pair [`ss_list_price, ss_wholesale_cost`] is selected and 200 queries are randomly generated for each of `COUNT, SUM, AVG, PERCENTILE, VARIANCE` and `STDDEV`. Sample sizes are 10k, 100k, 1 million tuples, and the query range varies from 0.1%,0.5%, 1% to 10% of the range-attribute's domain. c.) *Group-by analysis* contains 30 randomly generated queries for the column pair `ss_sold_date_sk, ss_sales_price` with the `Group By` attribute `ss_store_sk`. d.) *Join analysis* contains 42 randomly generated join queries between table `store_sales` and table `store` on join key `ss_store_sk`. The performance of aggregates on `ss_whole_sale_cost` and `ss_net _profit` is analyzed by varying `s_number_of_employees`. e.) *Complex TPC-DS* uses query number 7 and (complex subqueries of) query 5 and 77 from TPC-DS involving 2-way and 5-way joins, 2-4 AFs, and 57 to 25,000 groups (in Section 4.4.10).

Combined Cycle Power Plant Workload

CCPP [158] contains 9568 rows showing hourly average ambient variables of a power plant. It is scaled up to 2.6 billion records. There are 5 columns: Temperature (T), Ambient Pressure (AP), Relative Humidity (RH), Exhaust

Vacuum (V), and Net hourly energy output (EP). 108 queries are randomly generated for three column pairs `[T,EP]`, `[AP,EP]` and `[RH, EP]`, with query ranges varying from 0.1%,1%,5% to 10%.

Beijing PM2.5 Workload

This data set [100] contains PM2.5 data of Beijing International Airport and US Embassy. There are 43824 records totally and this dataset is similarly scaled up. The target is to predict pm2.5[PM25] level, given Dew Point (DEWP), Pressure (PRES), Temperature (TEMP), Cumulated wind speed (IWS), etc. 72 queries are randomly generated for four column pairs [DEMP, PM25], [PRES, PM25], [TEMP, PM25] and [IWS, PM25], and similar range-query selectivity was used.

Baseline Comparison Setup

We compare DBEst against state of the art AQP engines: VerdictDB [123] (source code obtained from [124]), BlinkDB [6] (source code obtained from [136]). We also compare with the results from an exact-answer columnar analytics RDBMS (MonetDB [79]) using uniform samples to approximate results (in Section 4.4.9). Initially, DBEst is configured to run using a single thread and BlinkDB is deployed in pseudo-cluster mode in order to compare fairly (without hiding costs for acquiring/using large clusters). After that, DBEst is configured to use all cores. VerdictDB always runs over Spark using all 12-cores.

4.4.2 DBEst Sensitivity Analysis

We stress-test DBEst under varying (i) range-query sizes (selectivity), (ii) sample sizes (used to build the density estimator and regression models), and (iii) across all AFs.

Sample Size Effect

Query ranges are set at 1% of the domain size. Sample sizes vary from 10k, 100k, 1M, and 5M records. Figure 4.2 shows DBEst's relative errors. The

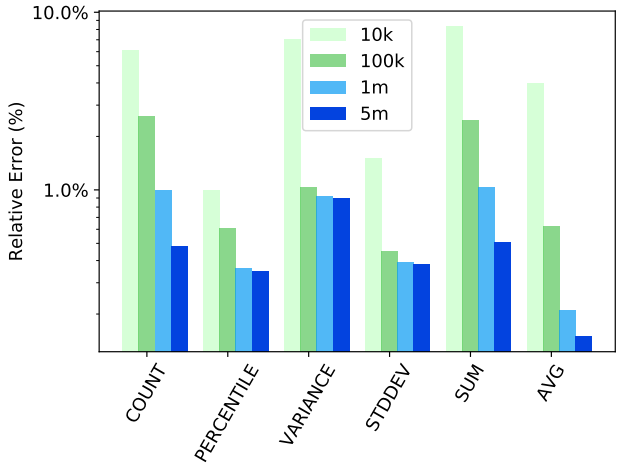


Figure 4.2: Influence of Sample Size on Relative Error

relative error is less than 10%, regardless of sample size, and as the sample size increases the relative error drops significantly, bringing it to below 1% when the sample has 1 million records.

Figure 4.3 shows the corresponding query response times. As expected, smaller samples yield shorter response times and that approximately, an order of magnitude smaller sample yields an order of magnitude shorter response time. The message here is that with a sample of 10k records, response times are well below 100 milliseconds! And this buys a relative error of $< 10\%$. Investing into samples of 100k records, brings down relative errors to below 1% for `PERCENTILE, VARIANCE, STDDEV, AVG` and to a few % for `COUNT, SUM` while response times hover around 0.3 second.

To provide more context, Figure 4.4(a) shows results for DBEst and VerdictDB for state-building times (sampling + model training time for DBEst and sampling time for VerdictDB). Results show that envisaged DBEst sample sizes, yield big improvements in state building times compared to VerdictDB. It is noticed that when the sample size goes beyond 5m, the state building time of DBEst is larger than VerdictDB. We argue that it is not necessary to generate a sample larger than 5m as DBEst achieves a good prediction accuracy when the sample size is 100k or 1m.

Figure 4.4(b) shows the space overhead of DBEst and VerdictDB. Recall,

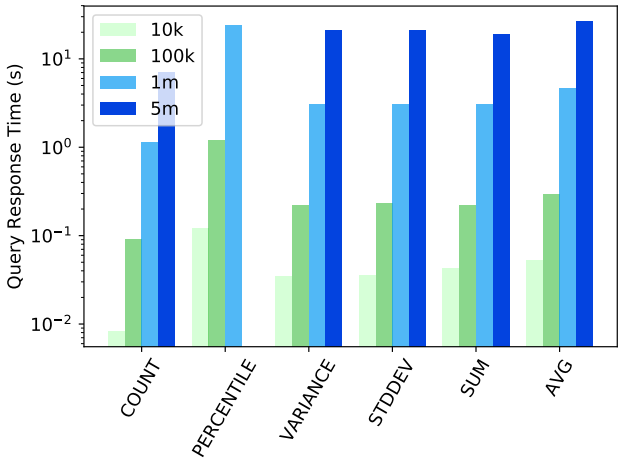
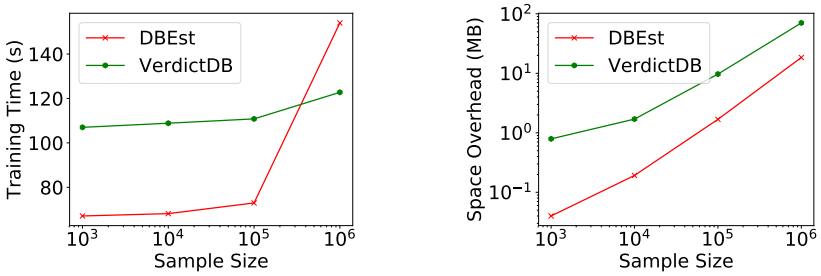


Figure 4.3: Influence of Sample Size on Response Time



(a) Training Time (b) Space Overhead

Figure 4.4: DBEst vs VerdictDB Overheads

DBEst needs to maintain only the models and not the samples for query execution. The space overhead of DBEst is 1 to 2 orders of magnitude less than VerdictDB's.

Query Range Effect

In this section, we will analyze the effect of query selectivity on the performance of DBEst. In a RDBMS, a large query selectivity means more rows will be scanned during query processing, which leads to higher query response time. To control the query selectivity, we select (0.1%, 1% to 10%) of the query domain as the query ranges, and the sample sizes are fixed at 100k. In Figure 4.5, as ranges increase, we see a decrease in the error for all AFs. This is expected as smaller samples are pressed hard to find enough representatives. However,

accuracy performance is nonetheless excellent.

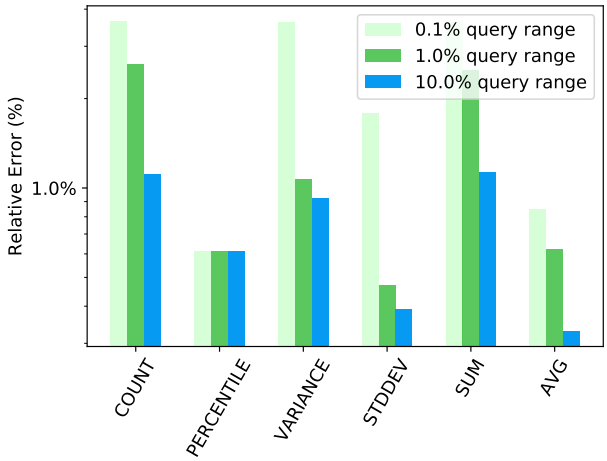


Figure 4.5: Influence of Query Range on Relative Error

Figure 4.6 shows response times. Except for `PERCENTILE`, (as multiple

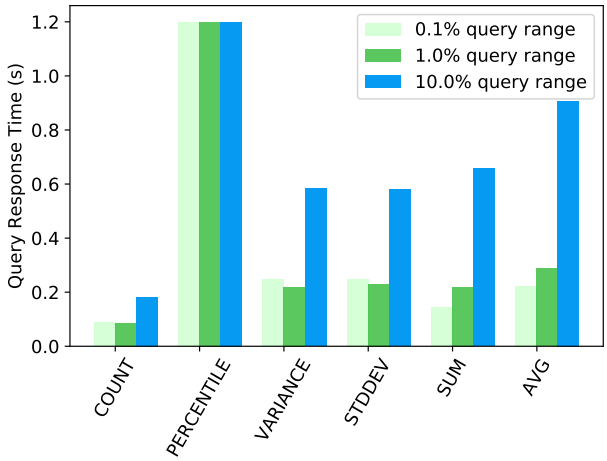


Figure 4.6: Influence of Query Range on Response Time

integrals are involved in finding the p_{th} point and times are 1.2secs) the query time for all other AFs is less than 1 second. As expected, query times increase as query ranges increase, as integral evaluation take more time.

To summarize this section, we conduct the sensitivity analysis of DBEst by varying the query range and sample size. Increasing the sample size will improve the prediction accuracy of DBEst at the cost of slightly increased query response time. In terms of selectivities, DBEst tends to achieve better

accuracy for queries with a larger selectivity, while the query response time might increase.

4.4.3 CCPP Workload Performance

CCPP is scaled to include 2.6 billion records, (similar to the scaled-up TPC-DS) totaling around 1.4TB in size. 108 queries are randomly generated for `COUNT`, `SUM` and `AVG` for 3 column pairs, stress-testing with low-selectivity query ranges (0.1%, 0.5% to 1%). We compare the accuracy performance between DBEst, VerdictDB, and BlinkDB over samples sizes varying of 10k to 100k.

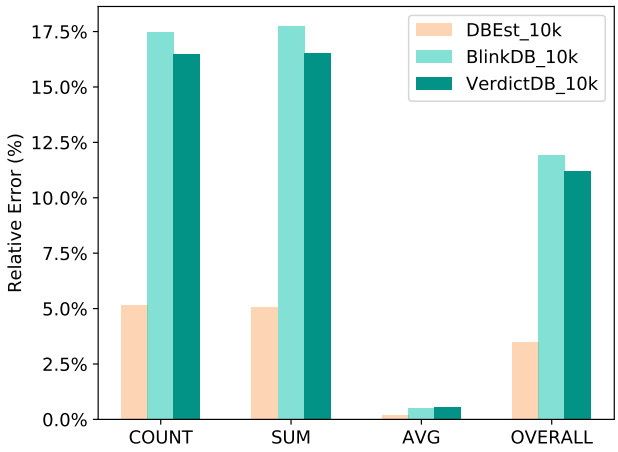


Figure 4.7: Relative Error: CCPP Dataset (10k Sample)

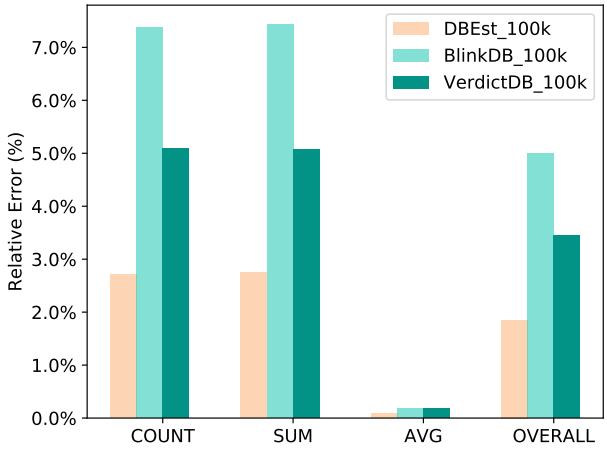
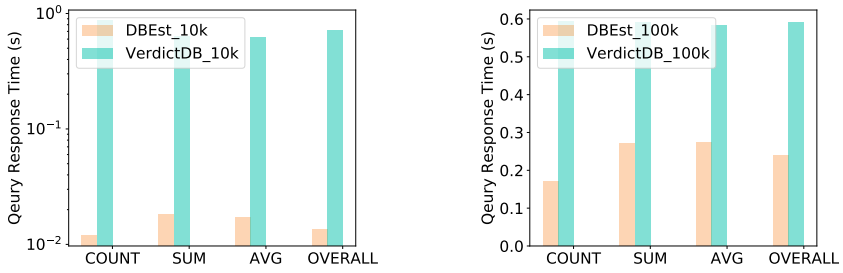


Figure 4.8: Relative Error: CCPP Dataset (100k Sample)

Figure 4.7 and Figure 4.8 summarize the average relative error of DBEst, VerdictDB and BlinkDB. The overall error of DBEst is 3.5%, while for the other QP engines, the corresponding error is more than 10% for 10k samples (especially for `COUNT` and `SUM`). For 100k samples, DBEst error drops to 1.9% and the error of VerdictDB drops to 3.5%. Thus, to achieve the same accuracy, VerdictDB acquires one order of magnitude larger sample size. The accuracy of BlinkDB is worse than VerdictDB.

It is worth pointing out that the error of `AVG` queries is always less than `COUNT` and `SUM` queries. This could be explained by Equation (4.1), Equation (4.6) and Equation (4.7). For `COUNT` and `SUM`, there is a scaling factor N, which is used to scale up the prediction from samples to represent the actual data. For `AVG`, there is no such scaling factor, and we assume that the average value from the sample is identical to the average value of the population.

Figure 4.9 shows the query times for DBEst and VerdictDB. For DBEst they are less than 0.3 seconds. The average query response time is around 0.02 seconds if the sample size is 10k, and increases to 0.27 seconds for 100k samples. The time cost for VerdictDB varies between 0.6 to 0.9 seconds. Hence, DBEst brings speedups from ca. 4x to ca. 30x. Please note that VerdictDB uses all 12 cores, while DBEst uses 1 thread.



(a) 10k sample size (b) 100k sample size

Figure 4.9: Response Time for CCPP Dataset

We also conduct the experiments using MonetDB [79], and the comprehensive comparison results between DBEst and MonetDB are shown in Section 4.4.9.

4.4.4 TPC-DS Workload Performance

Using appropriate values (the sample size and query range from the sensitivity analysis) we evaluate accuracy, response times and time/space overheads for both DBEst and VerdictDB for TPC-DS.

Accuracy

Figure 4.10 shows the average relative errors. Given the same sample size, DBEst always achieves better prediction accuracy than VerdictDB for aggregates `COUNT, SUM` and `AVG`. For this workload, if/when the sample size is 10k,

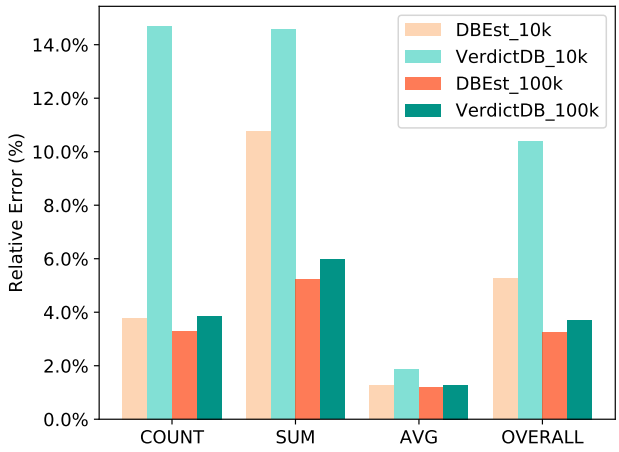


Figure 4.10: Relative Error: DBEst vs VerdictDB

there is a big difference in accuracy: Overall, DBEst achieves 5.26% relative error, while VerdictDB involves more than 10% relative error. For 100k samples, both DBEst and VerdictDB have excellent error, and DBEst wins only slightly. The relative error is less than 8% for DBEst and BlinkDB, which shows their great abilities to provide approximate answers.

Again, the relative error for `AVG` queries is much smaller than `COUNT` and `SUM` queries. This is due to the error caused by the scaling factor for `COUNT` and `SUM`, please refer to Section 4.4.3 for a detailed explanation.

Query Response Time

Figure 4.11 shows corresponding query times of DBEst and VerdictDB. For

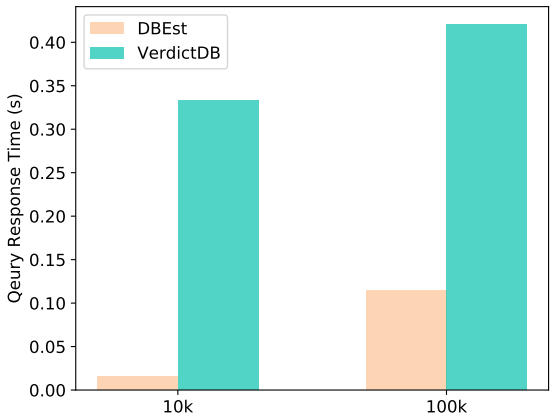


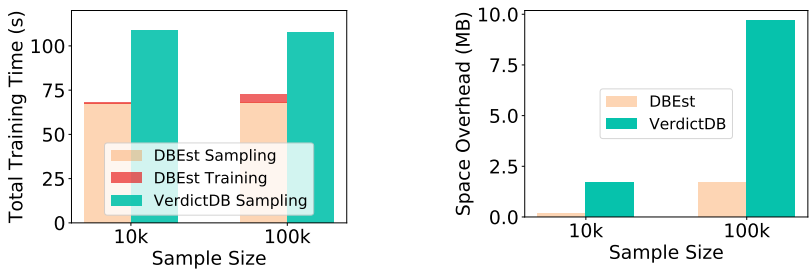
Figure 4.11: Response Time: DBEst vs VerdictDB

10k samples, DBEst takes less than 0.02 seconds to process a query, while VerdictDB takes around 0.33 second. Query response times of DBEst increase to 0.12 second for 100k samples while VerdictDB requires >0.40 seconds to process the same queries. Juxtaposing the last two figures yield consistent conclusions with what we observed from the sensitivity analysis with respect to trade-offs between response times and accuracy. Overall, DBEst enjoys speedups from ca. 3.5x to ca. 16x than VerdictDB and better accuracy. Please note that VerdictDB times use all 12 cores, while DBEst uses just 1 thread.

Space Overheads

We now evaluate both the training time and space overheads of DBEst and VerdictDB. Figure 4.12(a) summarizes the averaged model sample+training time of DBEst and VerdictDB's sampling time per column pair.For 10k samples, DBEst takes around 68s to generate a sample and 0.65s to build the models. While the average time for VerdictDB to generate the sample is around 108s. When the sample size increases to 100k, the time cost to generate samples remains the same, while it takes around 4.97s for DBEst to build the models. Overall, the total state building time of DBEst is less than that of VerdictDB.

Figure 4.12(b) shows the space overheads. For 10k samples it takes DBEst about 0.192MB to keep one regression model and one density estimator, while



(a) Training Time (b) Space Overhead

Figure 4.12: Overheads: DBEst vs VerdictDB

the memory overhead for VerdictDB is 1.7MB to keep the sample. For 100k samples, DBEst needs about 1.68MB to keep the models, while VerdictDB needs ca. 9.7MB for its samples. So, in terms of space DBEst offers an improvement from 5x to 9x.

In this section, we evaluate the performance of DBEst for the TPC-DS dataset from several perspectives. DBEst outperforms VerdictDB in prediction accuracy, while achieving orders of magnitude savings in query response time and space overheads.

4.4.5 Beijing Workload Performance

The Beijing data set is scaled up to 100 million records, and 72 queries are randomly generated across AFs.

Figure 4.13 displays relative errors obtained by DBEst and VerdictDB. We notice a big difference in accuracy when small samples are used. For 10k samples, the average relative error by DBEst is 4.72%, while the relative error by VerdictDB is 9.57%. For 100k samples, the relative errors drop to 1.67% and 4.41%, respectively. Thus, as before, sample-based AQPs give higher errors if the sample size is small (especially when range predicate selectivity is small). As DBEst adopts models on top of samples, which can generalize, DBEst requires smaller samples to make more accurate estimations.

Figure 4.14 shows the corresponding query response times for various sample sizes. Even if the sample size is 10k, VerdictDB still needs at least 0.38s to produce the answer, while around 0.6s are needed for 100k samples.

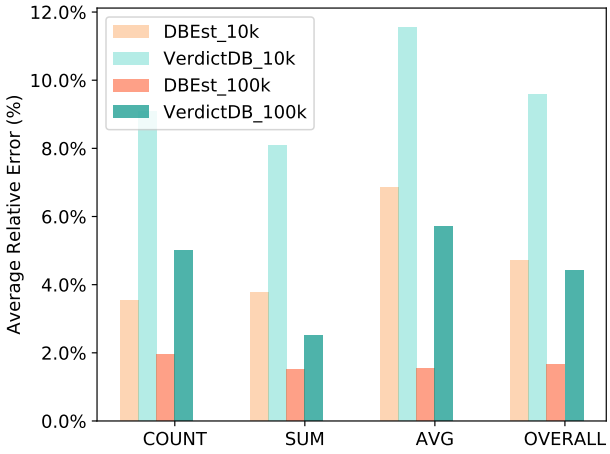


Figure 4.13: Accuracy: DBEst vs VerdictDB

With 10k samples, DBEst needs only 0.013s to provide an answer; with 100k samples, DBEst needs around 0.23s, This agrees with the above sensitivity study. Overall, DBEst brings speedups from ca. 3x to ca. 30x compared to VerdictDB. Please note that VerdictDB times use all 12 cores, while DBEst uses just 1 thread.

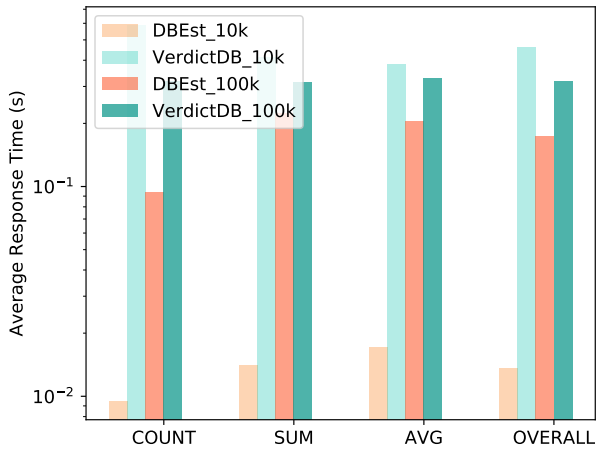


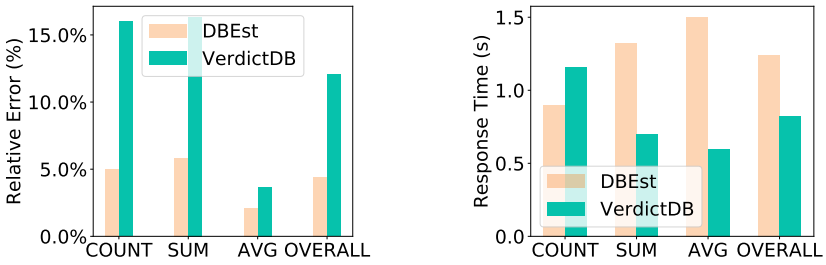
Figure 4.14: Response Time: DBEst vs VerdictDB

Again, DBEst outperforms VerdictDB in both prediction accuracy and query response time for a different data set.

4.4.6 TPC-DS Group By Performance

In total, 90 queries are used for the `[ss_wholesale_cost_sk, ss_list_price]` column pair from the TPC-DS workload, having 30 queries for each of `COUNT`, `SUM` and `AVG`, where the `GROUP BY` attribute is `ss_store _sk`. The table used is `store_sales` and is scaled up to include 100 million tuples. There are 57 distinct values for the `GROUP BY` attribute. The sample size for DBEst is chosen so that on average there will be 10k rows for each `GROUP BY` value.

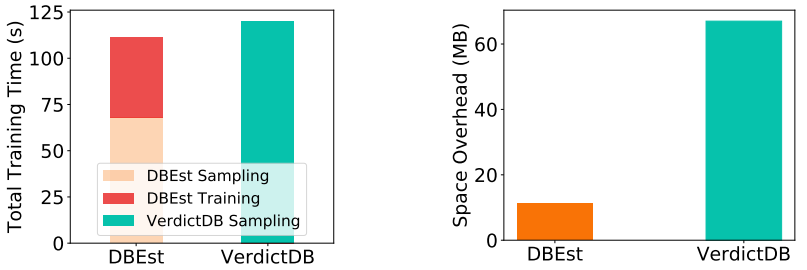
Figure 4.15(a) shows average relative errors (averaged over all 57 groups). For `COUNT` and `SUM`, DBEst outperforms VerdictDB significantly. For `AVG`, both have similar relative error, which is less than 3%, and DBEst performs slightly better.



(a) Relative Error (b) Response Time

Figure 4.15: Query Performance for 57 Group Values

Query response times are shown in Figure 4.15(b). VerdictDB takes slightly less time than DBEst for a `GROUP BY` query. Note, VerdictDB uses all cores, while DBEst only uses one. section 4.4.7 will show a DBEst with parallel `GROUP BY` processing.



(a) Training Time (b) Space Overhead

Figure 4.16: Overheads for 57 Group Values

Figure 4.16 shows the time/space overheads for building the states required by DBEst and VerdictDB. The conclusions for the `Group By` case are consistent with all previous results on overheads of DBEst and VerdictDB. Note, DBEst models are currently trained in sequence. If the models are trained in parallel, the training time is 1 order of magnitude smaller.

We now study error performance for individual groups. Figure 4.17 shows the histogram of the relative error for the 57 groups for `SUM` queries. The average error for the `SUM` queries in DBEst is 5.84% and for VerdictDB 16.32%. More than 80% of the 57 groups have a relative error <7.0% for DBEst. For VerdictDB, the minimum achieved error is around 10%. Note also that variance around the mean is smaller for DBEst and large for VerdictDB. The maximum relative errors are ca. 10% and 24%, respectively, and several groups suffer from errors >20% with VerdictDB.

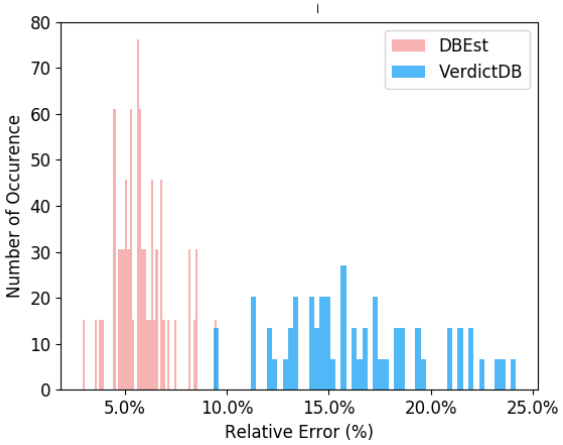
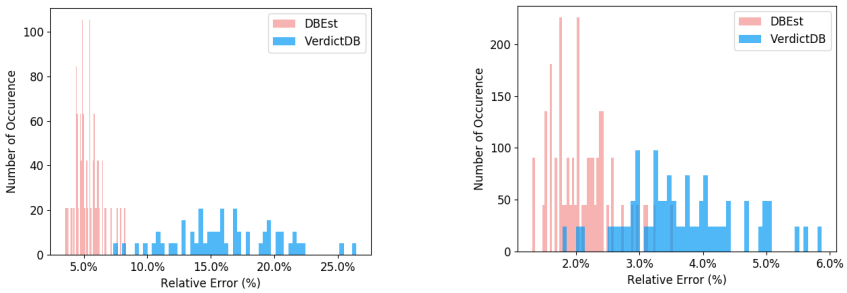


Figure 4.17: Accuracy Histogram: SUM for 57 Groups



(a) COUNT (b) AVG

Figure 4.18: Accuracy Histogram for 57 GROUPS

Here, we also demonstrate the same for AFs `COUNT` and `AVG`, as shown in Figure 4.18. For `COUNT`, the average relative errors by DBEst and VerdictDB are 5.34% and 16.13%, respectively. It is also noticeable that the error has a smaller variance from DBEst, while VerdictDB tends to produce a bad prediction with a big variance. VerdictDB, as a sample-based AQP engine, uses samples to produce the answer. There will be fewer records for rare groups, which leads to higher variance in the prediction accuracy across groups. The generalization of models in DBEst helps to reduce the variance between groups. The same conclusion holds for AF `AVG` as well.

4.4.7 Parallel Query Execution

Previous experiments had DBEst run with a single thread, while VerdictDB (or Spark) made use of all 12 cores. Here, we show DBEst's performance when running in parallel.

Parallel GROUP BY

If there are n distinct groups DBEst builds n models uses them all to answer the query. The n models can be evaluated in parallel. Recall that, our current implementation for parallel model evaluation is suboptimal as python has the global interpreter lock issue during parallel inference.

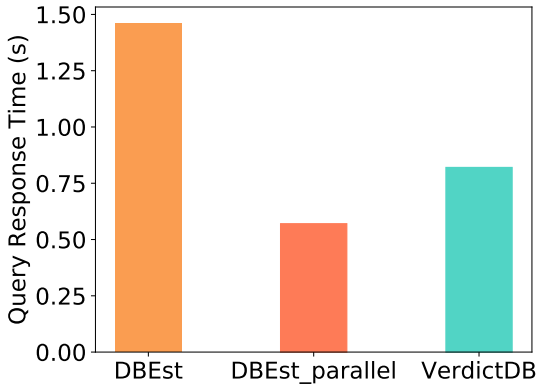


Figure 4.19: Group By Query Response Time Reduction

Single-threaded DBEst needs 1.46s, while VerdictDB using all 12 cores

needs 0.82s. Multi-core DBEst brings the total query response time down to 0.57s, as shown in Figure 4.19.

To be fair, with the current implementation, DBEst would take more time than VerdictDB when the number of groups exceeds ca. 100. But this is largely an implementation issue, as in principle, per-group model evaluation is embarrassingly parallelizable. For queries with a large number of groups, we could distribute models among a cluster, and each node is only responsible for a partition of the groups. In this way, we could further reduce the query response time. It is also important to note that the `Group By` queries tested did not involve any joins. As will be shown later, processing even relatively small joins is ca. 60x more expensive in VerdictDB (as it needs to compute the join of million-tuple samples), whereas DBEst does not. In such cases, `Group By` in DBEst becomes better practically always than VerdictDB (since when the number of groups becomes very high, none of the systems would develop samples/models and let the exact-answer QP engine handle such queries). Nonetheless, as we see below, it may be preferable to accept longer query processing times, even using per-query single-threaded execution, in order to increase system throughput.

Throughput with Parallel Execution

All state of the art AQP engines utilize many or all nodes/cores in the system *for each query execution* - *intra-query parallelism* - in order to reduce response times to acceptable levels. In principle, this will reduce overall system throughput, as concurrently executing queries would conflict for threads/cores. DBEst allows for large *inter-query parallelism* levels, as most queries execute using a single thread.

Figure 4.20 displays the impact of the number of cores on the total query response time for the CCPP dataset. With DBEst, time decreases as more cores are used. With 12 cores, the total time drops from 35.4s to 5.78s (from 4.6 to 0.9) for 100k (10k) samples. However, for VerdictDB, as each query uses all cores, the total query processing time remains unaffected. DBEst improves

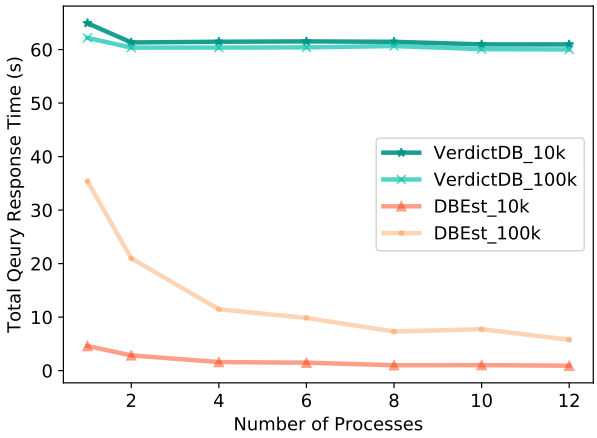
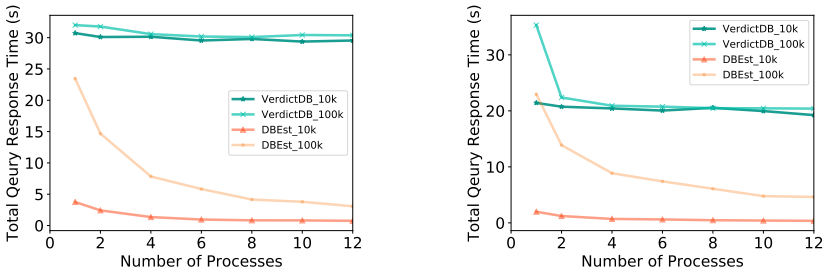


Figure 4.20: Throughput of Parallel Execution (CCPP)

throughput by ca. 6x to 30x.



(a) TPC-DS (b) Beijing PM2.5

Figure 4.21: Throughput with Parallel Query Execution

Here we further demonstrate DBEst's abilities in processing queries for the TPC-DS and Beijing PM2.5 Datasets. Figure 4.21(a) shows the overall time for processing all 97 queries in the TPC-DS dataset, as the number of processes increases from 1 to 12. For example, with 100k-samples, it takes ca. 24s for DBEst to process all 97 queries with a single process. This time drops to 3.07s when 12 processes are running in parallel. This means that to answer a single query, DBEst only needs around 32ms on average. The same conclusions hold for the Beijing PM2.5 dataset, (Figure 4.21(b)). We observe speedups up to ca. 10x by utilizing all available cores.

4.4.8 Join Query Processing

We now demonstrate DBEst's performance for join queries. Two tables from TPC-DS `store_sales` and `store`, are joined on `ss_store_sk`. Aggregates on `ss_net_profit` and `ss_ wholesale_cost` are analyzed by varying `store.s_number _of_employees`.

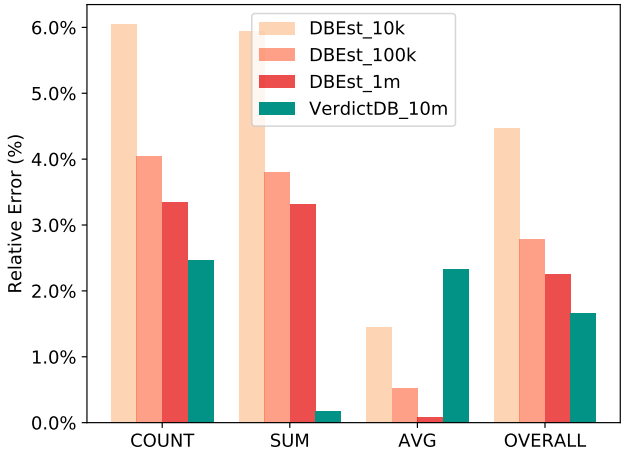
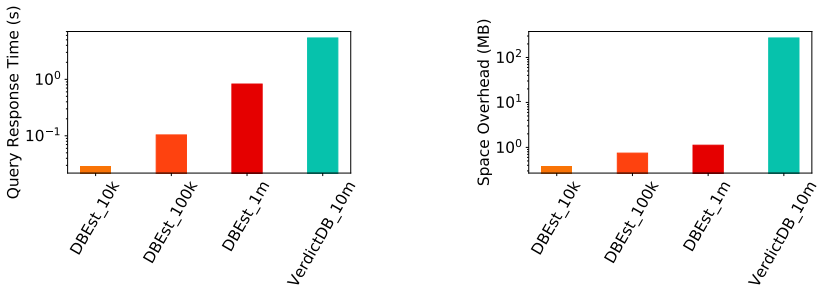


Figure 4.22: Join Accuracy Comparison

42 queries are used for the [`s_number_of_ employees`, `ss_net_profit`] and [`s_number_of_employees`, `ss_wholesale _cost`] column pairs, having 14 queries for each of `COUNT, SUM` and `AVG`. VerdictDB joins a sample of the large fact table (default size of 10m tuples) with the actual small 60-row dimension table.

Figure 4.22 shows the overall DBEst error is 4.48% (10k samples) and 2.24% (1m samples). As VerdictDB uses a very large 10m sample, the error is slightly better (1.66%). Section 4.4.10 will show cases where VerdictDB has a higher error than DBEst. Section 4.4.9 will also show how robust DBEst accuracy is for joins even when stressed with skewed join-attribute distributions, unlike other approaches.

Turning to Figure 4.23, for 10k samples, DBEst needs only 0.028s and 0.37MB. For 1m samples, it needs 0.82s and 1.12MB. VerdictDB, takes 6.7s, while requiring >270MB. Overall, DBEst achieves speedups from 8x to >200x and smaller space overhead from ca. 100x to 250x. The improvements would be much larger if two large tables were joined.



(a) Response Time (b) Space Overhead

Figure 4.23: Join Performance Comparison

4.4.9 Comparison With MonetDB

It is instructive to discuss how a standard exact-answer system, turned into an approximate query engine by operating on samples, would fair against DBEst and other AQP engines. Here we address this issue using a state-of-the-art columnar DB for analytics, MonetDB.

There is little argument that such systems, like MonetDB, could crunch very efficiently samples and produce approximate answers. The point here is not about response times. Using an exact-answer QP engine over a sample, could yield large errors, unless samples got very large. The relative error bounds achievable with such techniques are well understood. For example, (using 0.9 probabilty Hoeffding bounds) [65] for COUNT, relative errors are ca. 1.22 / $(s \times \sqrt{(n)})$, where s is the selectivity in the query result before the aggregate operation (i.e., combined selectivity of all selection and join operations) and n is the sample size. As these are bounds, we conducted the experiments below using our datasets and queries from above.

As many models are needed to support the `GROUP BY` queries, DBEst needs significantly higher query response time to process each group sequentially, but, in any case, overall response time is 360ms (single-threaded) or 107ms (12 cores), while MonetDB processes the query with a few ms. Turning to errors, Figure 4.24 compares the performance between DBEst and MonetDB for the TPC-DS `GROUP BY` workload. Using 10k samples, DBEst achieves an overall relative error of 4.43%, while MonetDB's corresponding relative error

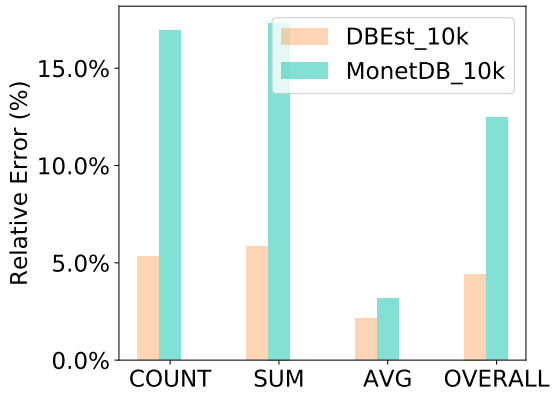
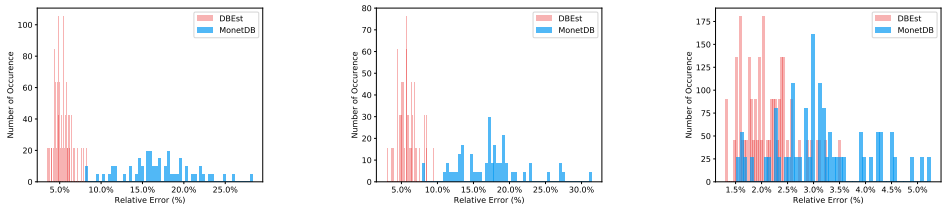


Figure 4.24: Error vs MonetDB : TPC-DS Group By

is 12.46%. To shed more light into accuracy-performance, the histograms of relative errors for `COUNT`, `SUM` and `AVG` are summarised in Figure 4.25.



(a) COUNT (b) SUM (c) AVG

Figure 4.25: Error Histogram vs MonetDB: TPC-DS GROUP By Workload

Take `SUM` as an example. The maximum (minimum) error produced by DBEst is ca. 10% (2%). While for MonetDB, the corresponding relative errors are >30% (ca. 8%). So, DBEst provides estimations with low mean error and variance among groups, while for several groups MonetDB's error is unacceptably high.

Figure 4.26 summarizes DBEst and MonetDB's accuracy for CCPP. The same conclusions hold. DBEst error is better than MonetDB's, even when the latter uses an order of magnitude larger samples. As models are ca. one order of magnitude more compact than actual samples, the above translates to achieving lower error with ca. 53x smaller space requirements (340KB vs 18.24MB).

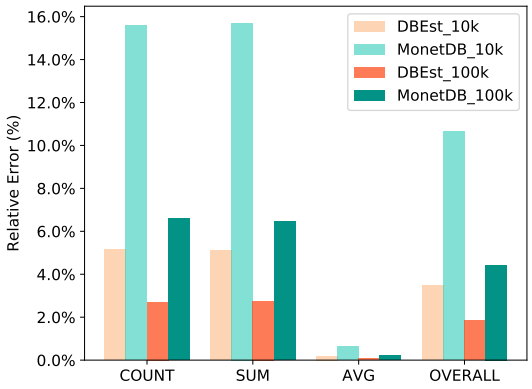


Figure 4.26: Error vs MonetDB: CCPP Workload

Approximate MonetDB and Joins

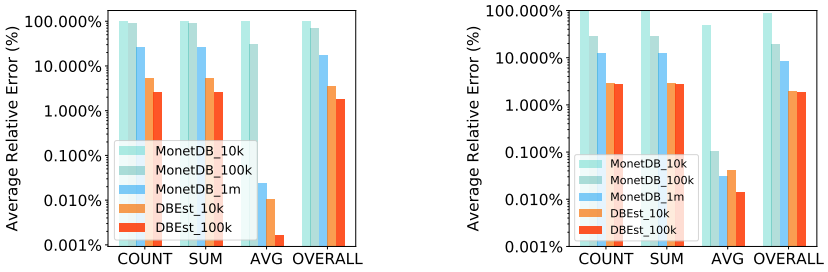
We now study approximate MonetDB for join queries. [28, 173] state that the join result accuracy is greatly influenced by the distribution of the join attribute. We create two tables A(x,y) and B(z,y), whose join attribute y follows the Zipf distribution with density function $p(x = k) = k^{-s}/\zeta(s)$ where k is the rank, s is the Zipf parameter ($s \geq 1$ and higher values yield more skewed distribution) and $\zeta()$ is the Riemman's zeta function. Here, $s = 2$.

Table A (B) has 100k (100m) records. And y in Table B has a skewed and a non-skewed region. MonetDB answers 20 queries (10 for the skewed region) of the form:

```
SELECT COUNT(z), SUM(z), AVG(z)
FROM A, B
WHERE A.y=B.y
```

using 10k, 100k, and 1m samples from Table B and small Table A.

Figure 4.27 shows that MonetDB is significantly more challenged with such distributions. Unlike DBEst, MonetDB error is unacceptably high and it could not answer any query (3 queries) with the 10k (100k) samples. This is expected as MonetDB will at most achieve the same accuracy as VerdictDB (sometimes even worse, depends on the sampling method), as they both use samples to make the prediction.



(a) Skewed Region (b) Non-Skewed Region

Figure 4.27: Accuracy Comparison for Join Queries

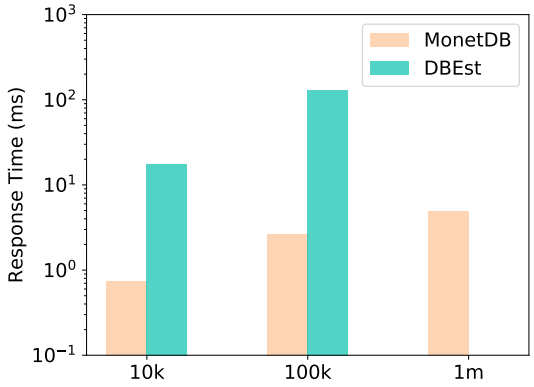


Figure 4.28: Query Response Time Comparison

Even with 1m samples, MonetDB errors for `COUNT` and `SUM` are 25.41% and 25.39%. In contrast, DBEst always achieves high accuracy, with errors ranging from 1.74% to 3.51%.

Figure 4.28 shows that MonetDB is much more efficient than DBEst: for 10k (100k) sample, DBEst takes 17.57ms (129ms), while MonetDB only requires 0.74ms (2.65ms). This is as expected, since MonetDB has been optimized for over 2 decades now (and it is also written in C which is inherently much faster than Python). Nonetheless, DBEst's time is in absolute terms very fast (less than 100ms), while also guaranteeing high accuracy. As you can see, DBEst trained over a 100k sample usually outperforms MonetDB trained with a 1m sample. This showcases DBEst's all-around strong performance.

To summarize this section, we compare the performance of DBEst against MonetDB for simple and join queries. MonetDB requires far less time to respond to a query as it has been optimized for decades. In terms of prediction

accuracy, MonetDB acts as a sample-based AQP engine, does not work well for the workloads. DBEst achieves much better accuracy at the cost of slightly higher query response time.

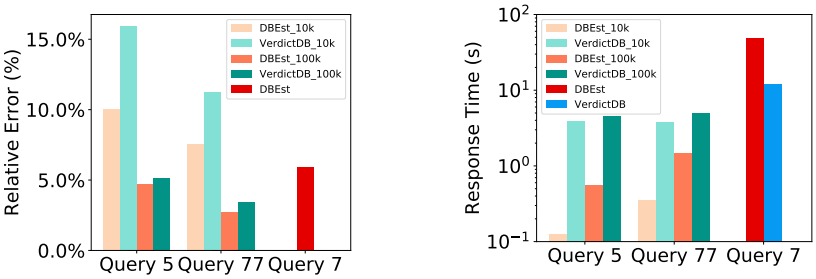
4.4.10 Complex TPC-DS Queries

We now turn to DBEst's performance for complex queries as they appear exactly in the benchmark: Namely, Query 7 and (complex subqueries of) Query 5 and Query 77. These queries involve 2 to 4 AFs, 2- to 5-way joins, as well as nested subqueries (flattened out and materialized for DBEst).

There are 57 groups for Query 5 and 77, and >25,000 for Query 7. As stressed earlier, queries like Query 7 would not be handled by DBEst due to the very large number of groups. In fact, query 7 will not be handled by systems like QuickR either (as groups have a very low support – less than 20 entries per group). So, this represents an extreme stress-test.

Performance is summarized in Figure 4.29. Sample sizes vary from 10k to 100k. Overall, DBEst achieves higher accuracy and significantly smaller response times: For Query 77 and 10k-samples, as an example, DBEst (VerdictDB) achieves an overall relative error of 7.56% (11.24%). If the sample size increases to 100k, the relative error drops to 2.76% and 3.42%, respectively. Note this is in contrast to the accuracy observed in Figure 4.22. For Query 7, as the joined tables have fewer than 10m rows, VerdictDB computes the exact answer (zero error). Given that each of the 25k groups consists of <20 records, DBEst is trained on the complete join-table instead of on samples. The overall error is <6%, although a small percentage of groups have relative errors higher than 20%.

As explained, DBEst's query response time is greatly influenced by the number of groups in the queries. For Query 7, the query response time is ca. 600s using a single-threaded implementation. Figure 4.29(b) shows performance with multi-threaded DBEst, which exploits all 12 cores and reduces the response time to ca. 50 seconds. Dividing the 25k groups into N model bundles and using a scale-out cluster of N such 12-core nodes would reduce further the time



(a) Relative Error (b) Response Time

Figure 4.29: Performance for TPC-DS Queries 5, 7, 77

by a factor equal to N. So, a cluster of ca. 50 such nodes would be required to attain a sub-second response time.

In this section, we compare the performance of DBEst against VerdictDB for complex queries as they appear exactly in the benchmark. We show that for queries with a small number of groups, DBEst achieves better performance (in both query response time and prediction accuracy). However, for queries with a large number of groups, we see an increase in the query response from DBEst. This is one of the limitations of DBEst, which will be handled in the following chapter.

4.5 Summary

With this chapter we presented DBEst, an SML-model-based AQP engine. DBEst's salient feature is that it processes queries using regression and density-estimator models. Its key insight is that derived models can generalize nicely, thus able to attain high accuracy despite being built from very small samples. These facts allow DBEst to offer highly accurate AQP with dramatic speedups, while being very frugal in memory requirements, as models are very compact. DBEst's philosophy additionally departs from related work in being frugal with respect to demands for system resources during AQP: often single-threaded DBEst outperforms multi-core AQP engines. This chapter studied DBEst's sensitivity on key parameters and systematically evaluated it against two state of the art AQP engines, studying separately the effects of range predicate

selectivities, `GROUP BY`, and join operations, as well as the impact of using multiple cores/nodes. In the next chapter, we will introduce an improved AQP engine based on deep learning networks. It overcomes the major drawbacks of DBEst and will have better support for categorical attributes and model updating.

Chapter 5

DBEst++: An Improved Model-Based AQP Engine

In Chapter 4, we introduce the model-based AQP engine DBEst. It is based on regression models and density estimators to produce approximate answers to SQL queries. DBEst achieves better accuracy than other AQP solutions, including VerdictDB and BlinkDB, while enjoys orders of magnitude savings in space overheads and query response time. However, DBEst has some limitations. For example, DBEst is not an ideal solution for queries with a large number of groups. In this chapter, we aim to overcome these drawbacks by applying deep learning networks and word embeddings. We will also look into the updateability issue of models.

5.1 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 [57] for Microsoft SQL Server, MADLib [76] over PostgreSQL), or deep neural networks (like [77, 116], and for tasks such as deriving learned cost models [91, 148]), workload forecasting [103], database tuning [96, 160, 172], cardinality and selectivity estimation [74, 77, 169] and learned indexing [95] etc).

Approximate query processing has traditionally relied on sampling approaches. These are largely classified as online [87] (i.e., the sample is generated after the query arrives) or offline (i.e., samples are generated in advance, either for popular queries [6] or for all possible queries for the schema [123]). 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 [11–14, 105, 137, 138]. 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 [104], as introduced in Section 4.2, followed by DeepDB [77], and [153], 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.

5.2 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 for AQP tasks. In other words, prediction accuracy is not the only requirement, we should consider other factors, including space overheads and response time.

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, compared with DBEst as introduced in Chapter 4, our key concern and contribution with this chapter 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 better query execution times (especially being embarrassingly parallelizable, reducing query times at will with additional investment), and
- offering better 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.

The remainder of this chapter is organized as follows. Section 5.2 presents the rationale, motivations, and overall vision and contributions of *DBEst++*. Section 5.3 overviews the *DBEst++* internals. It explains its core ML models, how models are trained, and how models are used for inference. Section 5.4 demonstrates the performance of *DBEst++* for the TPC-DS and the Flights datasets and compares it against DeepDB and VerdictDB. Section 5.5 concludes.

5.3 System Overview

5.3.1 DBEst++ Query Processing Foundations

This chapter shares the similar mathematical foundations as DBEst, which is introduced in Chapter 4. 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.

$$SUM(y) = N \cdot \int_{lb}^{ub} D(x)R(x)dx \tag{5.1}$$

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

Chapter 4. They are omitted for space reasons.

Unlike DBEst which used KDEs and a Regressor (XGBoost), *DBEst++* employs Mixture Density Networks (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 [62].

$$\begin{aligned} Var(x) &= E[(x-\mu)^2] \\ &= \sum_{i=1}^{m} w_i(\sigma_i^2 + \mu_i^2 - \mu^2) \end{aligned} \tag{5.2}$$

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

5.3.2 System Architecture

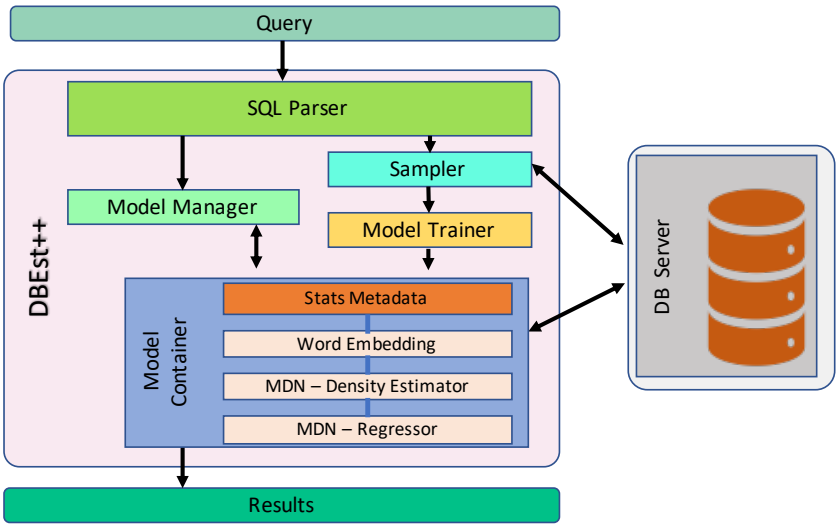


Figure 5.1: *DBEst++* System Architecture

Figure 5.1 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 BY`s); (iii) The Model Manager selects the appropriate models from the Model Container to use per query and also selects representative data points from the range of variable values specified in selection range predicates with which to call for MDN predictions. The representative points are selected so that the range of interest is uniform divided. After that, these representative values from MDNs 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`. Reservoir sampling is fast in producing a uniform sample with a fixed size. 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 $P(y|g,x)$.

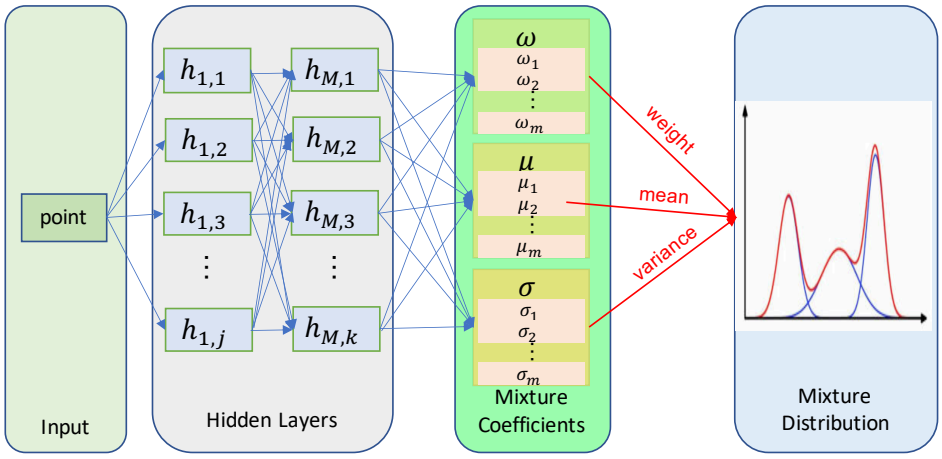


Figure 5.2: Structure of Mixture Density Networks

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.

5.3.3 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 [68, 69] with high success. For instance, Apple uses MDNs for speech recognition [151].

Figure 5.2 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

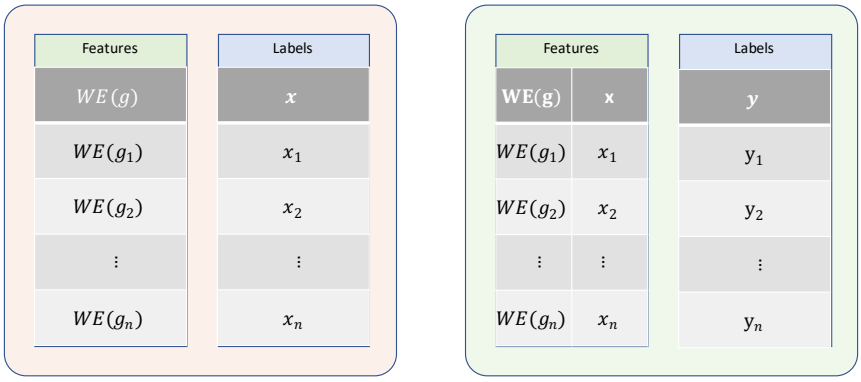
labels are y, NLL takes the format of

$$\underset{\theta}{\operatorname{argmin}}\, l(\Theta) = -\frac{1}{|\mathbb{D}|} \sum_{(\mathbf{x},y)\in\mathbb{D}} \log p(y|\mathbf{x}) \tag{5.3}$$

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 5.3.4), 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 5.3 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 5.3: Input features and labels for training MDNs

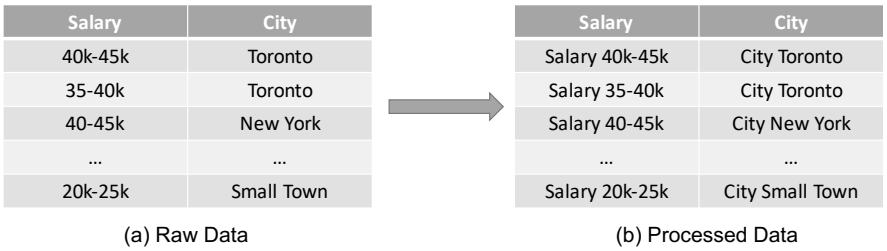
5.3.4 Word Embeddings

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 [133] and Association Rule Mining methods [4], 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 [36], [98] and [66], but Skip_Gram [112] have been highly successful. In Section 5.4.4 we will compare the performance of word embedding against one-hot/binary encoding in *DBEst++*, and 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 5.4(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.



Salary	City
40k-45k	Toronto
35-40k	Toronto
40-45k	New York
...	...
20k-25k	Small Town

(a) Raw Data

Salary	City
Salary 40k-45k	City Toronto
Salary 35-40k	City Toronto
Salary 40-45k	City New York
...	...
Salary 20k-25k	City Small Town

(b) Processed Data

Figure 5.4: 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.

5.3.5 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 leaned 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. We use this method as a baseline in our study.

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 [67, 92, 132] 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 5.4.

5.4 PERFORMANCE EVALUATION

All code, datasets, and query workloads used in the following experiments can be found at: https://github.com/qingzma/DBEst_MDN. *DBEst++* is written in python with more than 11,000 lines of code.

5.4.1 Experimental Setup

We have evaluated *DBEst++* using column sets from queries in TPC-DS dataset [115]. 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

[1] `https://www.kaggle.com/usdot/flight-delays`

used in the DeepDB paper. IDEBench [52] 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.

5.4.2 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.

Universal Models

Figure 5.5 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)

[2] `https://radimrehurek.com/gensim/models/word2vec.html`

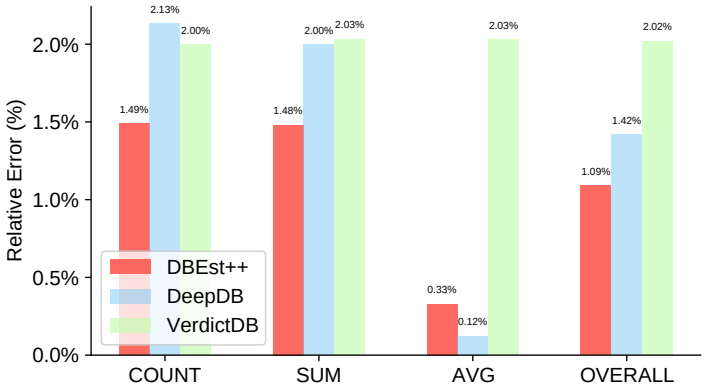


Figure 5.5: Relative Error for SUM / COUNT / AVERAGE Queries over the TPC-DS Dataset (SF=10)

equal to 100 and 1000. Relative errors for `COUNT` and `SUM` are shown in Figures 5.6 and 5.7. Again, *DBEst++* achieves smaller errors across all SFs.

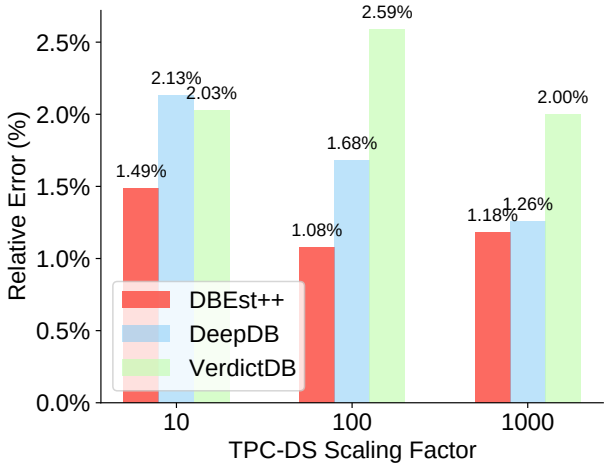


Figure 5.6: Scalability for COUNT Queries Varying SF

Figure 5.8 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. Specifically, when a sample is used to produce the query result, the whole sample is scanned, which is a time-consuming process. While models are faster to respond. 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%

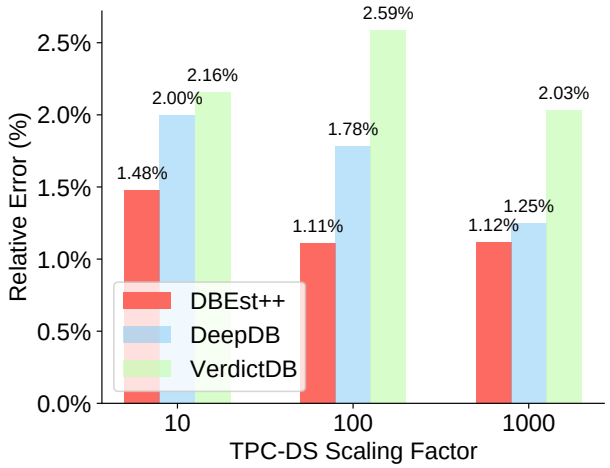


Figure 5.7: Scalability for SUM Queries Varying SF

to 40%.

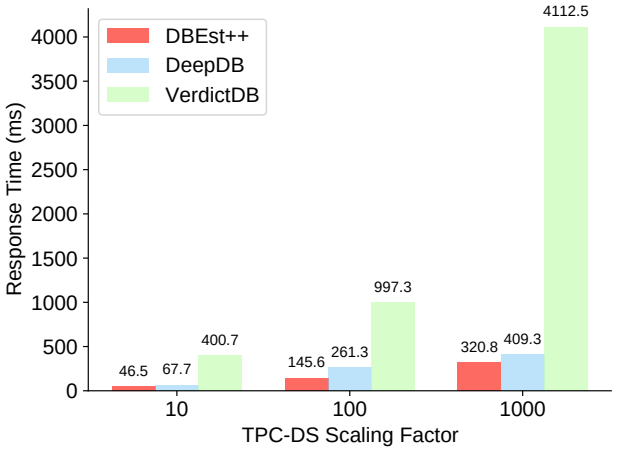


Figure 5.8: Comparison of Query Response Times for Queries over the TPC-DS Dataset

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 5.9, *DBEst++* achieves ca. 3 orders of magnitude savings compared to DeepDB or VerdictDB. Interestingly, universal DeepDB requires even higher space overheads than VerdictDB.

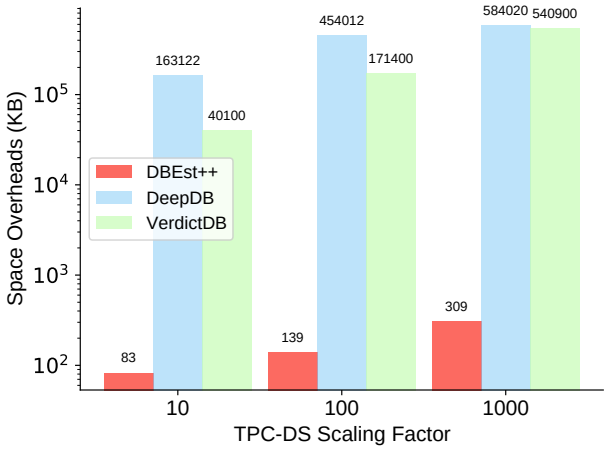


Figure 5.9: Comparison of Space Overhead for Queries over the TPC-DS Dataset

Compact Models

Figure 5.10 corresponds to Figure 5.5, showing the relative errors for `COUNT`, `SUM` and `AVG` queries over the TPC-DS dataset with SF=10. Figure 5.11 corresponds to Figures 5.6 and 5.7, 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 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 5.6). The corresponding relative error obtained by compact DeepDB is 1.84% (see Figure 5.10). This shows that building a universal model for all types of queries may lead to higher errors.

Figure 5.12 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.

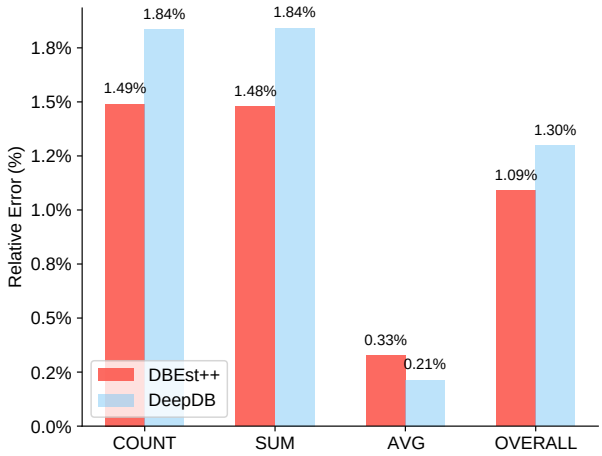


Figure 5.10: Accuracy comparison for Compact Models for Queries over the TPC-DS Dataset (SF=10)

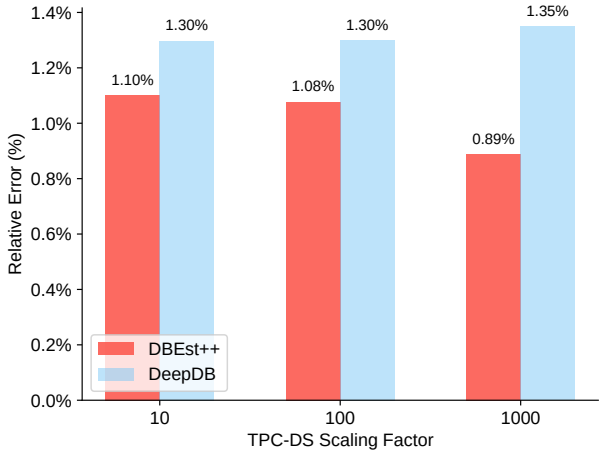


Figure 5.11: Comparison of Overall Relative Error for Queries over the TPC-DS Dataset

5.4.3 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 5.13 shows the relative errors for the Flights dataset. This dataset is much simpler and easier to achieve great performance than TPC-DS. We

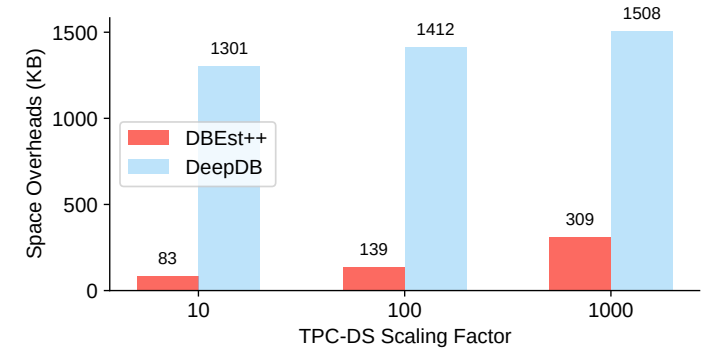


Figure 5.12: Space Overheads for Queries over the TPC-DS Dataset.

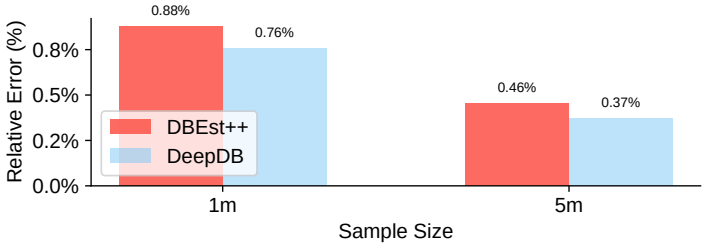


Figure 5.13: Accuracy Comparison for Queries over the Flights Dataset

include it here only in order to have a common test dataset to compare against DeepDB. It is noted that DeepDB achieves slightly better accuracy than *DBEst++*while both enjoy extremely high accuracy. As the flight dataset is simple, it is hard to reduce the error further. 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.

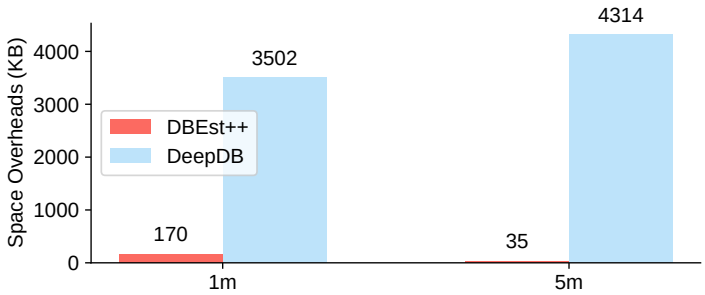


Figure 5.14: Space Overheads for Queries over the Flights Dataset

Figure 5.14 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.

5.4.4 Impact of Word Embedding

Section 5.3.4 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 5.4.2. 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.

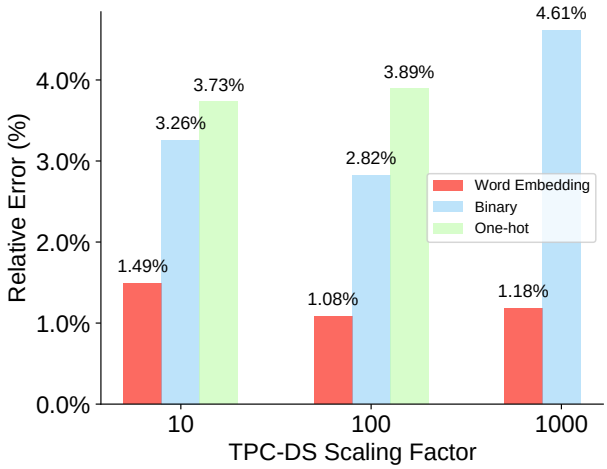


Figure 5.15: Comparison of Relative Error Between Word Embedding, One-hot and Binary Encoding for COUNT Queries.

Figures 5.15 and 5.16 summarize the relative error of *DBEst++* for quer-

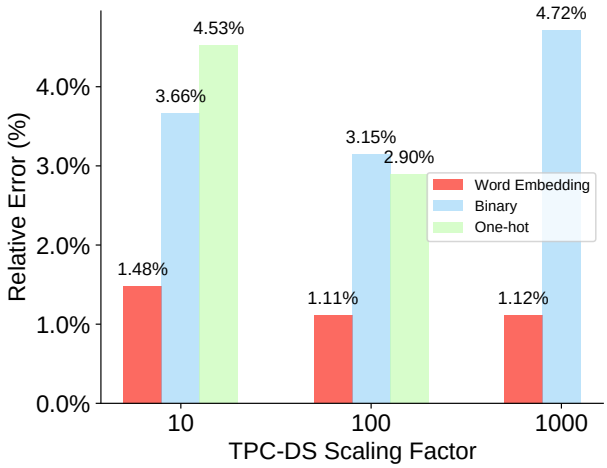


Figure 5.16: Comparison of Relative Error Between Word Embedding, One-hot and Binary Encoding for SUM Queries.

ies over the TPC-DS dataset using one-hot and binary encoding, and word embedding. Take `COUNT` queries as an example, as shown in Figure 5.15. 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.

5.4.5 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.

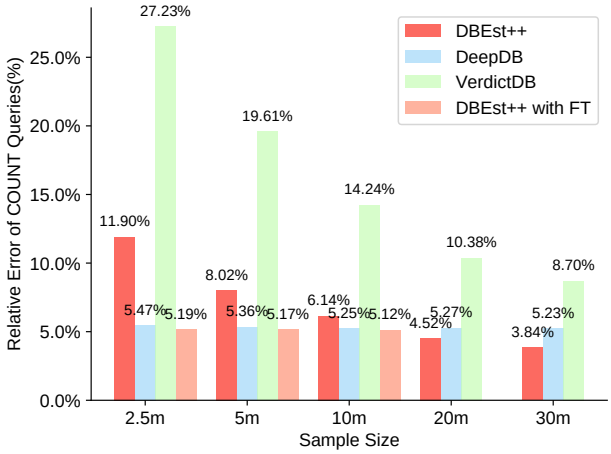


Figure 5.17: Comparison of Sensitivity on Large Groups for COUNT Queries.

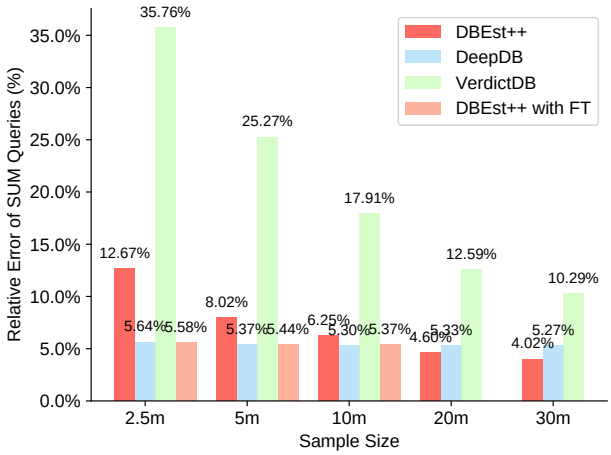


Figure 5.18: Comparison of Sensitivity on Large Groups for SUM Queries.

Figures 5.17 and 5.18 show the effect of sample size on relative error for *DBEst++*, DeepDB and VerdictDB. Take `COUNT` queries as an example, as shown in Figure 5.17. 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. In other words, for queries with large groups, there will be fewer rows in each group, which leads to higher error for each group. Due to the generalization of models, model-based AQP engines tend to perform better. 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. (5.1)). When sample sizes are smaller, as shown in Figures 5.17 and 5.18, *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 5.17 and 5.18. 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.

5.4.6 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.

As explained in Section 5.3.5, we will use different strategies to update the models. Figure 5.19-21 will demonstrate the performance of these updating strategies.

Figure 5.19 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.

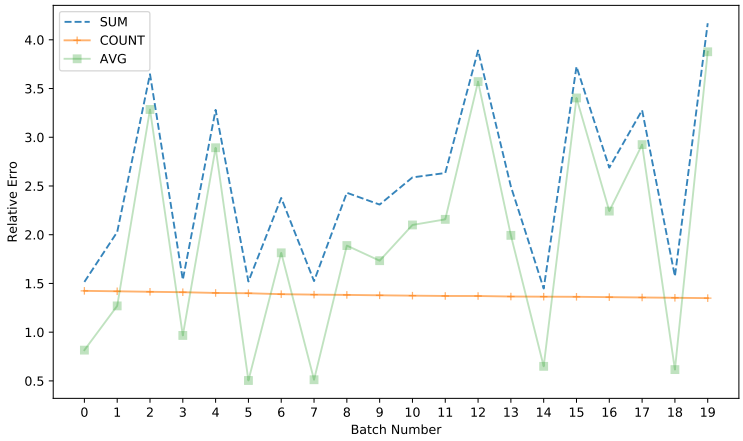


Figure 5.19: Relative Error When FTs Are Updated Only.

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 5.20 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".

To rectify this increased error we turn to using a smaller learning rate

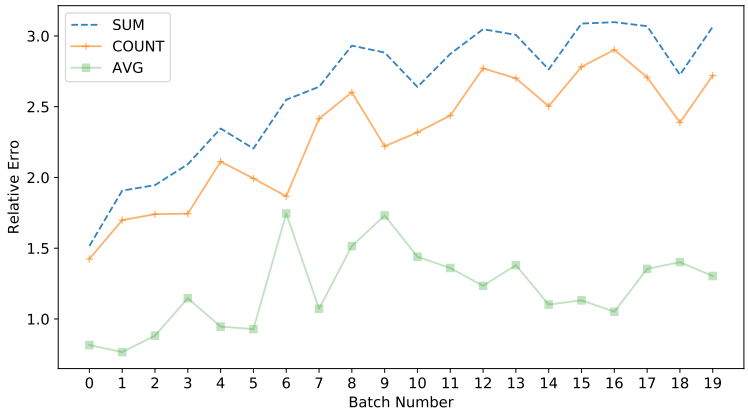


Figure 5.20: Relative Error When FTs and MDNs Are Updated.

when we feed-forward the new items through the MDNs. Figure 5.21 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 5.1 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 (2-46s).

Table 5.1: 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

To summarize this section, we evaluate the updateability of *DBEst++* using three strategies. In the first experiment, we assume the new data has the same data distribution as the old data, and only the FTs are updated. However, such an updating strategy is not realistic and will only work for simple cases.

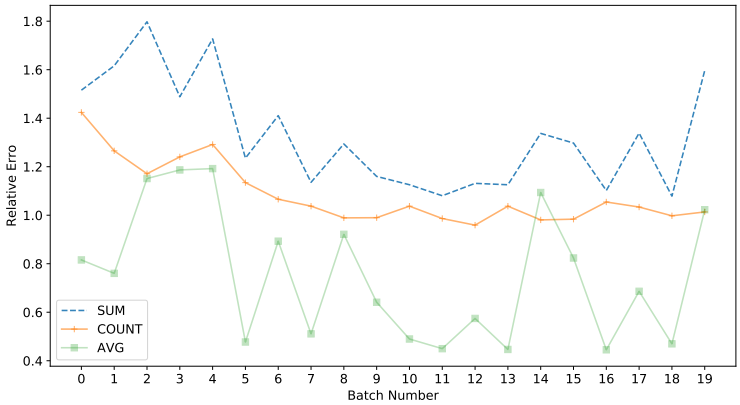


Figure 5.21: Fine-tuning Models With a Smaller Learning Rate

In the second experiment, we update both the FTs and models. After several batches of updates, the updated model "forgets" the old distribution, and we see increased prediction error. Finally, we update the model with a reduced learning rate. In this way, the updated model enjoys high accuracy for both old workload and new workload. We believe updating the model with a reduced learning rate is a correct direction for model updating.

5.4.7 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 5.22 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.

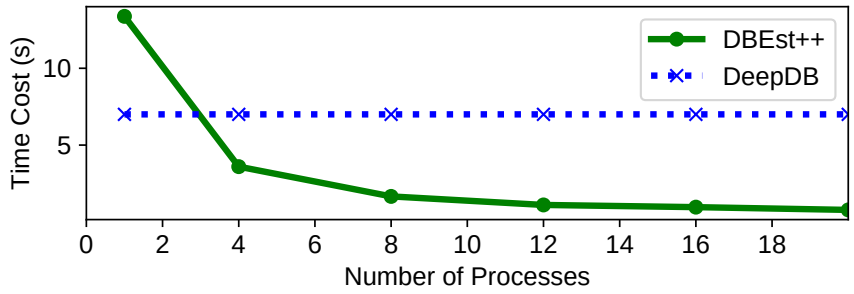


Figure 5.22: Query Response Time Reduction with Varying Degrees of Parallelism

5.5 Summary

In this chapter, 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, as introduced in Chapter 4. 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.

Chapter 6

Conclusions and Future Work

In the era of big data, it takes a long time to get the exact query result from a modern database. Aiming to reduce the query response time by providing approximate answers, approximate query processing is receiving greater attention due to the rapid growth of data volume. Previous AQP solutions were dominated by sample-based approaches. They provide orders of magnitude reduction in query response time at the cost of minor query quality loss. Given the current speed of data growth and the requirement to handle massive queries, approximate query processing remains an essential tool for data analytical tasks.

In this thesis, we have presented our approximate query processing solutions by applying machine learning models in three stages, as listed below:

Chapter 3: We reveal and shed light into an interesting '*impedance mismatch*' phenomenon between ML and data systems: namely, as analysts typically target specific data subspaces to analyze (using range/search queries on attributes of interest), top-notch RMs perform poorly for different such data subspaces, despite the fact that said RMs have been trained to generalize and achieve great accuracy-performance over the whole data domain. We use eight real-life data sets and data from TPC-DS, with various dimensionalities to quantify this phenomenon. We also employ new appropriate metrics, substantiating the problem across a wide variety of popular RMs, ranging from simple linear models to advanced, state-of-the-art, ensembles. This essentially surfaces

a model-management problem. Second, we put forth and study a new type of solution, based on a classification-based ensemble, which bears a *query-centric* perspective: That is, the proposed solution will improve per-query accuracy, regardless of the dataset and the subspaces analyzed, using near-always the best model for the task at hand. Finally, we study the solution's scalability limitations and show how to overcome them.

Chapter 4: We present DBEst, a model-based approximate query processing engine. Unlike sample-based AQP engines where samples are generated and maintained to produce an approximate answer, models are used instead. As models can generalize nicely, they achieve higher accuracy than samples. Specifically, regression models and density estimators are the main tools for producing approximate answers for aggregate queries. We extensively analyze the performance of DBEst against two sample-based AQP engines for various datasets. And it turns out that DBEst achieves better accuracy, while enjoys orders of magnitude savings in space overheads and response time. We also conduct experiments to reveal DBEst's performance for GROUP BY and join queries. In addition, we demonstrate/stress-test a parallel version of DBEst for queries with a large number of groups.

Chapter 5: We learn lessons from DBEst and propose an improved AQP engine, coined *DBEst++*. This new engine is comprised of mixture density networks (MDNs) and word embedding. Specifically, MDNs are used for regression and density estimation tasks as they are capable of serving queries with a large number of groups accurately and efficiently. And word embedding improves the prediction accuracy significantly. Also, as MDNs are compact models and are fast to respond, *DBEst++* further achieves 1-2 orders of magnitude savings in space overheads and response time. In addition, we look into the issue of model updating, and propose a new updating strategy by applying a reduced learning rate.

As we have demonstrated, model-based AQP engines like *DBEst++* outperform sample-based AQP engines in various aspects, including prediction accuracy, space overheads and response time. We hope that the proposed tech-

niques in this thesis open up new directions for approximate query processing by machine learning. Now, we will list the key findings and future work in the following sections.

6.1 Key Findings

6.1.1 Overfitting-Generalization Dilemma

Regression Models (RMs) and Machine Learning models (ML) in general, aim to offer high prediction accuracy, even for unforeseen queries/datasets. This depends on their fundamental ability to generalize. However, overfitting a model, with respect to the current DB state, may be best suited to offer excellent accuracy. This overfit-generalize divide bears many practical implications faced by a data analyst. Best practice, which suggests to an analyst to use a top-performing ensemble, is misleading and leads to significant errors for large numbers of queries. The query-centric regression proposed in this study addresses this problem by improving per-query accuracy while also offering excellent overall accuracy.

6.1.2 Universal Versus Light Models

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, like DeepDB [77]. 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). Our suggestion is to build light models for popular query templates only.

6.2 Future Work

During the study of this thesis, we have discovered and realized several interesting opportunities as the future work to improve and promote the usage of AQP engines. Here we list four interesting topics.

6.2.1 Universal AQP Interface for All Databases

Modern AQP engines have no/limited support for popular back-end servers. The works in this thesis [104, 107] and other model-based AQP engines like [77] only support AQP over files in the local file systems. They act as a demonstration to show the promises brought by models for AQP. This greatly limits their applications as modern analytical tasks are typically carried on databases with tables residing in a distributed storage. Some AQP engines have better support for various back-end servers. BlinkDB allows Spark as the backend server [6]. VerdictDB [123], as a successful AQP engine, supports the majority of popular databases.

In addition, the current AQP solutions acts as an additional layer above the local file systems/back-end servers, and there is no implementation of AQP methods into databases [6, 77, 104, 107, 123]. It would be great if *DBEst++* is able to support various backend servers. And this is a big step further towards the wide usage of AQP engines in real workloads.

6.2.2 Error Bound for DBEst

Currently, there is no error bound for the model-based AQP engines proposed in this thesis. Some AQP engines produce confidence intervals for the approximate answers [6, 77], while other engines like VerdictDB could automatically select the proper sample size based on the error threshold provided by users. In this study, we use regression and density estimator to produce the approximate answers. There are established theories for building confident intervals for regressions/density estimators. However, as an integral process is carried to aggregate the predictions from these models, the issue of calculating error

bound for our AQP methods remains open. To be specific, the new engine should be able to predict the error bound for the prediction it makes for each query. For instance, the new engine might guarantee that the error is less than 1% within a 95% confidence interval.

6.2.3 Support for Complex Queries

Another factor that affects the application of AQP engines is the limitation in the supported queries. Currently *DBEst++* supports queries with range predicates, equality conditions and/or a `GROUP BY` clause. However, more complex queries are not supported. For instance, join queries are only supported by firstly generating samples from the actual join result. While models built over different tables could not be extended to produce an approximate join result. Nested queries are not supported. Also, complex operations in the `SELECT` clause are not supported. For instance, *DBEst++* is not able to answer queries with such complex aggregate `SUM(income) - SUM(outcome)`.

6.2.4 Model Updateability - OLTP

Approximate query processing engines should support both Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP). However, the current solutions are focusing on OLAP and there is only limited support for OLTP. In the era of big data, the data volume is increasing at a fast speed, which opens up new research areas for updateability. For some scenarios only insertion occurs as new data are appended to the end of the tables. Other scenarios involve insertion, deletion and updates. Currently, *DBEst++* only supports insertion, while the solution for deletion remains open.

Bibliography

[1] Database pl/sql packages and types reference, Jul 2005. URL `https://docs.oracle.com/cd/B28359_01/appdev.111/b28419/u_nla.htm#CIABEFIJ`.

[2] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *ACM Sigmod Record*, volume 28, pages 574–576. ACM, 1999.

[3] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings of ACM Sigmod*. ACM, 1999.

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

[5] Drew Adams. Oracle database online documentation 12c release 1 (12.1). *Application Development*, 2014.

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

[7] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceeding of ACM SIGMOD*, 2017.

[8] Sam R Alapati and Charles Kim. Oracle database 11g, 2007.

[9] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[10] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. Aqwa: adaptive query workload aware partitioning of big spatial data. *Proceedings of the VLDB Endowment*, 8(13):2062–2073, 2015.

[11] Christos Anagnostopoulos and Peter Triantafillou. Learning set cardinality in distance nearest neighbours. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pages 691–696. IEEE, 2015.

[12] Christos Anagnostopoulos and Peter Triantafillou. Learning to accurately count with query-driven predictive analytics, 2015.

[13] Christos Anagnostopoulos and Peter Triantafillou. Query-driven learning for predictive analytics of data subspace cardinality. *ACM Trans. on Knowledge Discovery from Data, (ACM TKDD)*, 2017.

[14] Christos Anagnostopoulos and Peter Triantafillou. Efficient scalable accurate regression queries in in-dbms analytics. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 559–570, 2017.

[15] Anon. Xleratordb. `http://westclintech.com/`.

[16] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[17] Claudia Beleites, Ute Neugebauer, Thomas Bocklitz, Christoph Krafft, and Jürgen Popp. Sample size planning for classification models. *Analytica chimica acta*, 760:25–33, 2013.

[18] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA, 2007.

[19] Thierry Bertin-Mahieux, Daniel PW Ellis, Brian Whitman, and Paul Lamere. The million song dataset. 2011.

[20] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[21] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[22] Javier Burgués, Juan Manuel Jiménez-Soto, and Santiago Marco. Estimation of the limit of detection in semiconductor gas sensors through linearized calibration models. *Analytica chimica acta*, 1013:13–25, 2018.

[23] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[24] Zhuhua Cai, Zekai J Gao, Shangyu Luo, Luis L Perez, Zografoula Vagena, and Christopher Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1371–1382. ACM, 2014.

[25] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[26] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[27] Min-Te Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, 1982.

[28] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *ACM SIGMOD Record*, volume 28, pages 263–274. ACM, 1999.

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

[30] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[31] Tianqi Chen, Sameer Singh, Ben Taskar, and Carlos Guestrin. Efficient second-order gradient boosting for conditional random fields. In *Artificial Intelligence and Statistics*, pages 147–155, 2015.

[32] Eau Claire, Green Bay, Parkside La Crosse, and Stevens Point. Elementary statistics. 1952.

[33] William G Cochran. *Sampling techniques*. John Wiley & Sons, 2007.

[34] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[35] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

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

[37] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[38] Tyson Condie, Paul Mineiro, Neoklis Polyzotis, and Markus Weimer. Machine learning on big data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1242–1244. IEEE, 2013.

[39] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

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

[41] Graham Cormode, Minos 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.

[42] National Research Council et al. *Frontiers in massive data analysis*. National Academies Press, 2013.

[43] Dan Crankshaw, Peter Bailis, Joseph Gonzalez, Haoyuan Li, Zhao Zhang, Michael Franklin, Ali Ghodsi, and Michael Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *Conference on Innovative Data Systems Research (CIDR)*, 2015.

[44] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. *arXiv preprint arXiv:1612.03079*, 2016.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[46] Tewodors Deneke, Habtegebreil Haile, Sébastien Lafond, and Johan Lilius. Video transcoding time prediction for proactive load balancing. In *2014*

IEEE International Conference on Multimedia and Expo (ICME), pages 1–6. IEEE, 2014.

[47] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 73–84, 2006.

[48] Kevin K Dobbin and Richard M Simon. Sample size planning for developing classifiers using high-dimensional dna microarray data. *Biostatistics*, 8(1):101–117, 2006.

[49] Peter Dugan, Will Cukierski, Yu Shiu, Abdul Rahaman, and Christopher Clark. Kaggle competition. *Cornell University, The ICML*, 2013.

[50] Pavlos S Efraimidis. Weighted random sampling over data streams. *arXiv preprint arXiv:1012.0256*, 2010.

[51] Pavlos S Efraimidis and Paul G Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.

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

[53] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 689–692, 2012.

[54] FaceBook. Facebook q4 2020 earnings, January 2021. URL `https://investor.fb.com/home/default.aspx`.

[55] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. A proactive intelligent decision support system for predicting the popularity of online news. In *Portuguese Conference on Artificial Intelligence*, pages 535–546. Springer, 2015.

[56] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[57] Charles Flock and Joe Stampf. Xleratordb, 2009. URL `https://westclintech.com/`.

[58] David A Freedman. *Statistical models: theory and practice*. cambridge university press, 2009.

[59] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[60] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[61] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

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

[63] Wolfgang Gatterbauer and Dan Suciu. Approximate lifted inference with probabilistic databases. pvldb 8, 5 (2015), 629–640, 2015.

[64] L Gerhardt, CH Faham, and Y Yao. Scidb-py. `http://scidb-py.readthedocs.io/en/stable/`, May 2018.

[65] Phillip B Gibbons, Viswanath Poosala, Swarup Acharya, Yair Bartal, Yossi Matias, S Muthukrishnan, Sridhar Ramaswamy, and Torsten Suel. Aqua: System and techniques for approximate query answering. Technical report, Technical report, Bell Labs, 1998.

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

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

[68] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

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

[70] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data

cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[71] The PostgreSQL Global Development Group. *Documentation PostgreSQL 10.3*, 2018.

[72] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923. ACM, 2015.

[73] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. *Acm Sigmod Record*, 25(2):205–216, 1996.

[74] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1035–1050, 2020.

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

[76] 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. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

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

[78] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 137–152. ACM, 2015.

[79] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. Monetdb: Two decades of research in column-oriented database. 2012.

[80] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 277–281, 2015.

[81] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 277–281. ACM, 2015.

[82] Guido W Imbens and Tony Lancaster. Efficient estimation and stratified sampling. *Journal of Econometrics*, 74(2):289–318, 1996.

[83] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1): 79–87, 1991.

[84] Michael R Jiroutek, Keith E Muller, Lawrence L Kupper, and Paul W Stewart. A new method for choosing sample size for confidence interval–based inferences. *Biometrics*, 59(3):580–590, 2003.

[85] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[86] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Interactive data exploration using semantic windows. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 505–516. ACM, 2014.

[87] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*, page 631–646, 2016.

[88] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*, pages 3146–3154, 2017.

[89] Ken Kelley and Scott E Maxwell. Sample size for multiple regression: obtaining regression coefficients that are accurate, not simply significant. *Psychological methods*, 8(3):305, 2003.

[90] Martin Kersten and Lefteris Sidirourgos. A database system with amnesia. In *Proceeding of CIDR*, 2017.

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

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

[93] Georgia Koloniari, Yannis Petrakis, Evaggelia Pitoura, and Thodoris Tsotsos. Query workload-aware overlay construction using histograms. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 640–647, 2005.

[94] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. In *Cidr*, volume 1, page 9, 2015.

[95] 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*, pages 489–504, 2018.

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

[97] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.*, 44:17–22, 2016.

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

[99] Kim-Hung Li. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *ACM Transactions on Mathematical Software (TOMS)*, 20(4):481–493, 1994.

[100] Xuan Liang, Tao Zou, Bin Guo, Shuo Li, Haozhe Zhang, Shuyi Zhang, Hui Huang, and Song Xi Chen. Assessing beijing's pm2. 5 pollution: severity, weather impact, apec and winter heating. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 471 (2182):20150257, 2015.

[101] M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

[102] Sharon Lohr. *Sampling: design and analysis*. Nelson Education, 2009.

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

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

[105] Qingzhi Ma and Peter Triantafillou. Query-centric regression for in-dbms analytics. In *22nd International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*, pages 16–25, 2020.

[106] Qingzhi Ma and Peter Triantafillou. Query-centric regression. *Information Systems*, page 101736, 2021.

[107] Qingzhi Ma, Ali Mohhamedi Shanghooshabad, Meghdad Kurmanji, Mehrdad Almasi, and Peter Triantafillou. Learned approximate query processing: Make it light, accurate and fast. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[108] Oded Maimon and Lior Rokach. Data mining and knowledge discovery handbook. 2005.

[109] Scott E Maxwell, Ken Kelley, and Joseph R Rausch. Sample size planning for statistical power and accuracy in parameter estimation. *Annu. Rev. Psychol.*, 59:537–563, 2008.

[110] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[111] David Meyer, Friedrich Leisch, and Kurt Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169–186, 2003.

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

[113] Hannes Muehleisen, Anthony Damico, and Thomas Lumley. Monetdb.r. `http://monetr.r-forge.r-project.org/`, 7 2018.

[114] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006.

[115] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.

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

[117] Frank Olken. *Random sampling from databases.* PhD thesis, University of California, Berkeley, 1993.

[118] Carlos Ordonez. Statistical model computation with udfs. *IEEE Transactions on Knowledge and Data Engineering*, 22(12):1752–1765, 2010.

[119] Carlos Ordonez, Carlos Garcia-Alvarado, and Veerabhadaran Baladandayuthapani. Bayesian variable selection in linear regression in one pass for large datasets. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 9(1):3, 2014.

[120] Niketan Pansare, Vinayak R Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11): 1135–1145, 2011.

[121] Yongjoo Park. *Fast Data Analytics by Learning.* 2017. ISBN 0000000337. URL `https://deepblue.lib.umich.edu/handle/2027.42/138598`.

[122] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 587–602. ACM, 2017.

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

[124] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb. `https://verdictdb.org/`, 1 2019. (Accessed on 02/16/2019).

[125] Mary Parker. Sampling with replacement and sampling without replacement, January 2021. URL `https://web.ma.utexas.edu/users/parker/sampling/repl.htm`.

[126] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[127] Robert Piessens, Elise de Doncker-Kapenga, Christoph W Überhuber, and David K Kahaner. *QUADPACK: a subroutine package for automatic integration*, volume 1. Springer Science & Business Media, 2012.

[128] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, et al. Helios: Hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment*, 13(12):3231–3244, 2020.

[129] Navneet Potti and Jignesh M Patel. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment*, 8(9):898–909, 2015.

[130] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516*, 2017.

[131] Christopher Ré, Divy Agrawal, Magdalena Balazinska, Michael Cafarella, Michael Jordan, Tim Kraska, and Raghu Ramakrishnan. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 283–284. ACM, 2015.

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

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

[134] B. C. Ooi S. Wu and K.-L. Tan. Continuous sampling for online aggregation over multiple queries. In *Proceedings of Acm Sigmod*. ACM, 2010.

[135] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pages 1357–1369. ACM, 2015.

[136] Ariel Kleiner Sameer Agarwal, Henry Milner and Ameet Talwalkar. Blinkdb: Sub-second approximate queries on very large data. `https://github.com/sameeragarwal/blinkdb`, 2013.

[137] Fotis Savva, Christos Anagnostopoulos, and Peter Triantafillou. Explaining aggregates for exploratory analytics, January 2019.

[138] Fotis Savva, Christos Anagnostopoulos, Peter Triantafillou, and Kostas Kolomvatsos. Large-scale data exploration using explanatory regression functions. *ACM Transactions on Knowledge Discovery from Data*, 08 2020.

[139] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18, 2016.

[140] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

[141] Ali M Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. Pgmjoins: Random join sampling with graphical models. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[142] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. In *Proceeding of ACM SIGKDD*, 1999.

[143] Michael Shekelyan, Graham Cormode, Qingzhi Ma, Ali M Shanghooshabad, and Peter Triantafillou. Weighted random sampling over joins. In *(submitted)*, 2021.

[144] Bhadresh Shiyal. Synapse spark. In *Beginning Azure Synapse Analytics*, pages 119–150. Springer, 2021.

[145] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th*

symposium on mass storage systems and technologies (MSST), pages 1–10. Ieee, 2010.

[146] Riyaz Sikora et al. A modified stacking ensemble machine learning algorithm using genetic algorithms. In *Handbook of Research on Organizational Transformations through Big Data Analytics*, pages 43–53. IGI Global, 2015.

[147] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

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

[149] Srikanth Kandula Surajit Chaudhuri, Bolin Ding. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM SIGMOD international conference on Management of data*, 2017.

[150] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–6. IEEE, 2009.

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

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

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

[154] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[155] Srikanta Tirthapura and David P Woodruff. Optimal random sampling from distributed streams revisited. In *International Symposium on Distributed Computing*, pages 283–297. Springer, 2011.

[156] D. S. TKach. Information mining with the ibm intelligent miner. IBM White Paper, 1998.

[157] Jan E Trost. Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies. *Qualitative sociology*, 9(1): 54–57, 1986.

[158] Pınar Tüfekci. Prediction of full load electrical power output of a base load operated combined cycle power plant using machine learning methods. *International Journal of Electrical Power & Energy Systems*, 60:126–140, 2014.

[159] Graham Upton and Ian Cook. *A dictionary of statistics 3e*. Oxford university press, 2014.

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

[161] Daniele Varrazzo. Psycopg. `http://initd.org/psycopg/`, 2014.

[162] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[163] Rashmi Korlakai Vinayak and Ran Gilad-Bachrach. Dart: Dropouts meet multiple additive regression trees. In *Artificial Intelligence and Statistics*, pages 489–497, 2015.

[164] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[165] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 557–572, 2017.

[166] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 557–572. ACM, 2017.

[167] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[168] Denny Wong. Oracle data miner. *Oracle White Paper*, pages 1–31, 2013.

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

[170] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[171] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

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

[173] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539. ACM, 2018.