DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models

Qingzhi Ma
University of Warwick
q.ma.2@warwick.ac.uk

Peter Triantafillou
University of Warwick
p.triantafillou@warwick.ac.uk

ABSTRACT

In the era of big data, computing exact answers to analytical queries becomes prohibitively expensive. This greatly increases the value of approaches that can compute efficiently approximate, but highly-accurate, answers to analytical queries. Alas, the state of the art still suffers from many shortcomings: Errors are still high, unless large memory investments are made. Many important analytics tasks are not supported. Query response times are too long and thus approaches rely on parallel execution of queries atop large big data analytics clusters, in-situ or in the cloud, whose acquisition/use costs dearly. Hence, the following questions are crucial: Can we develop AQP engines that reduce response times by orders of magnitude, ensure high accuracy, and support most aggregate functions? With smaller memory footprints and small overheads to build the state upon which they are based? With this paper, we show that the answers to all questions above can be positive. The paper presents DBEst[1], a system based on Machine Learning models (regression models and probability density estimators). It will discuss its limitations, promises, and how it can complement existing systems. It will substantiate its advantages using queries and data from the TPC-DS benchmark and real-life datasets, compared against state of the art AQP engines.

CCS CONCEPTS

• **Information systems → Database query processing**; **Query optimization**; *Online analytical processing engines.*

KEYWORDS

Analytics, big data, efficiency, machine learning

ACM Reference Format:
Qingzhi Ma and Peter Triantafillou. 2019. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3324958

1 INTRODUCTION

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) 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:

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

where range predicates on attributes (x) 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. [33]). 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)

[1]'Est' derives from the Latin verb 'to be'and the prefix from 'estimator'

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3324958

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., [17, 58, 60]). 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.

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.

How long does it take a state of the art AQP engine to answer an analytical query of the types above over a mid-sized table (say, of a few Billion tuples)? Are, say, several minutes, acceptable? Is it acceptable to rely on the use of large clusters of tens or hundreds of nodes/cores upon which each query is sent to execute in order to bring response times to a few seconds? What if analysts cannot afford to procure such infrastructures or pay the cost to use them online on clouds? What are the implications of having each query execute across all nodes/cores on system throughput? Even then, can we do better than seconds? How much space is required to achieve such response times? Is ca. 10% relative error acceptable given the above time/space/money investments? And, how many analytical operators (i.e., AFs and other analytics tasks) can be supported, even at such costs?

This paper 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.

Contributions

This paper presents 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), while offering, compared to the state of the art:

- orders of magnitude shorter query response times,
- orders of magnitude higher throughput
- significantly smaller memory footprints per query,
- high accuracy,
- support for all mentioned aggregate functions, and
- short state-building times.

The paper 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 paper 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.

2 THE DBEST AQP ENGINE

2.1 System Overview

Fig. 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.

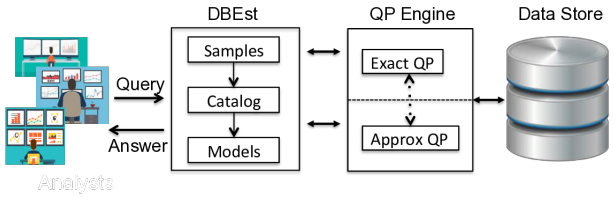


Figure 1: DBEst architecture.

There are three major components: (1) The sampling module interacts with the storage layer to build samples; (2) The models module, consists of 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. When a query arrives, DBEst reads the model catalog to check for models that could answer it. If so, the particular models will be used to process the query. If not, the query will be sent to an underlying system in the level below it, as shown in the architecture. This can be another AQP engine (e.g., one with online sampling, QuickR [29]) or it can go directly to an exact answer QP engine.

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 [36]. 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 [29], or VerdictDB [39]).

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 [29, 39] 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 [39]) then DBEst can support these nested queries using the above explained support for joins. Finally, DBEst does not currently support UDAs.

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 1 contains the notation used in this section.

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 1: Notation in §2

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

```
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.

2.3.1 *Computing Aggregates with Density Estimators.*

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

COUNT. Formally:

$$COUNT(y) \approx N \cdot \int_{lb}^{ub} D(x)dx \tag{1}$$

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:

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

$$\begin{aligned} STDDEV\_x(x) &= \sqrt{VARIANCE\_x(x)} \\ &= \sqrt{\frac{\int_{lb}^{ub} x^2 D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} xD(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2} \end{aligned} \tag{3}$$

By definition, the variance of x is equal to $\mathbb{E}\left[x^2\right] - [\mathbb{E}[x]]^2$. 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

$$F(x) = p \tag{4}$$

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

$$\alpha = F^{-1}(p) \tag{5}$$

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 through an iterative process, which is the Naive Bisection method for finding the root [27].

2.3.2 Computing Aggregates with Regression Models. Aggregates that can be computed using regression models include SUM, AVG, VARIANCE and STDDEV.

AVG. Formally:

$$\begin{aligned} AVG(y) &= \mathbb{E}[y] \\ &\approx \mathbb{E}[R(x)] \\ &= \frac{\int_{lb}^{ub} D(x)R(x)dx}{\int_{lb}^{ub} D(x)dx} \end{aligned} \tag{6}$$

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:

$$\begin{aligned} SUM(y) &= COUNT(y) \cdot AVG(y) \\ &\approx COUNT(y) \cdot \mathbb{E}[R(x)] \\ &= N \cdot \int_{lb}^{ub} D(x)dx \cdot \frac{\int_{lb}^{ub} D(x)R(x)dx}{\int_{lb}^{ub} D(x)dx} \\ &= N \cdot \int_{lb}^{ub} D(x)R(x)dx \end{aligned} \tag{7}$$

The sum of y equals the product of the count and the average value of y. From Equation 1 and 6, we get the approximate representations of the count and average value of y: multiplying equation 1 by equation 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:

$$\begin{aligned} VARIANCE\_y(y) &= \mathbb{E}\left[y^2\right] - [\mathbb{E}[y]]^2 \\ &\approx \mathbb{E}\left[R^2(x)\right] - [\mathbb{E}[R(x)]]^2 \\ &= \frac{\int_{lb}^{ub} R^2(x)D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} R(x)D(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2 \end{aligned} \tag{8}$$

$$\begin{aligned} STDDEV\_y(y) &= \sqrt{VARIANCE\_y(y)} \\ &\approx \sqrt{VARIANCE\_x(R(x))} \\ &= \sqrt{\frac{\int_{lb}^{ub} R^2(x)D(x)dx}{\int_{lb}^{ub} D(x)dx} - \left[\frac{\int_{lb}^{ub} R(x)D(x)dx}{\int_{lb}^{ub} D(x)dx}\right]^2} \end{aligned} \tag{9}$$

By definition, the variance of y is equal to $\mathbb{E}\left[y^2\right] - [\mathbb{E}[y]]^2$. 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:

$$\begin{aligned} AVG(y) &= \mathbb{E}[y] \\ &\approx \mathbb{E}[R(x_1, x_2)] \\ &= \frac{\int_{lb_1}^{ub_1}\int_{lb2}^{ub2} D(x_1, x_2)R(x_1, x_2)dx_2dx_1}{\int_{lb_1}^{ub_1}\int_{lb_2}^{ub_2} D(x_1, x_2)dx_2dx_1} \end{aligned} \tag{10}$$

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. 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 [29] 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.

3 IMPLEMENTATION

As shown in Fig. 1, a sample is firstly generated for every column set of interest, which 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.

Sampling

Stratified sampling [34] 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 [56] 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.

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 [52]. Histograms are the simplest form of density estimators and have enjoyed a prominent role in DBs [3] 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 [42], 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.

Regression Model Selection

High performance regression models include XGBoost [12], catboost [45], LightGBM [30], gradient boosting (GBoost) [21], 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.

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. [1, 10]), mining query logs, like BlinkDB [4] which showed that interesting column sets can be identified early in the execution of a typical workload, or depending on users, like VerdictDB [38], 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.

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` [27], which uses a technique from the Fortran library `QUADPACK` [43]. The underlying Gauss-Kronrod quadrature sums are fundamental to many of the automatic subroutines in `QUADPACK`. Given an integral

$$I_w[lb, ub]f = \int_{lb}^{ub} w(x)f(x)dx \tag{11}$$

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

$$I_w[lb, ub]f \approx \sum_{k=1}^{n} w_k f(x_k) \tag{12}$$

In Equation 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

$$\{R_{n_k}, E_{n_k}\}, k = 1, 2, ..., N \tag{13}$$

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 $1, ..., n-1$.

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 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.

4.1 Experimental Setup

4.1.1 *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 Appendix D).

4.1.2 *Combined Cycle Power Plant Workload.*

CCPP [57] 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%.

4.1.3 *Beijing PM2.5 Workload.*

This data set [33] 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.

4.1.4 *Baseline Comparison Setup.*

We compare DBEst against state of the art AQP engines: VerdictDB [38] (source code obtained from [39], BlinkDB [4] (source code obtained from [48]). We also compare with the results from an exact-answer columnar analytics RDBMS (MonetDB [25]) using uniform samples to approximate results (in Appendix C). 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.

4.2 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.

4.2.1 *Sample Size Effect.*

Query ranges are set at 1% of the domain size. Sample sizes vary from 10k, 100k, 1M, and 5M records. Fig. 2 shows DBEst's relative errors. The relative error is less than 10%,

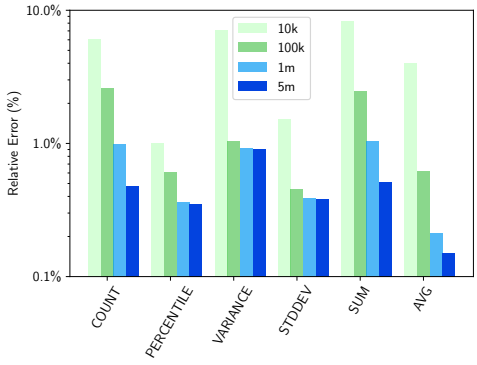


Figure 2: Influence of Sample Size on Relative Error

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.

Fig. 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, Fig. 4 (a) shows results for DBEst and VerdictDB for state-building times (sampling + model training time for DBEst and sampling time for VerdictDB).

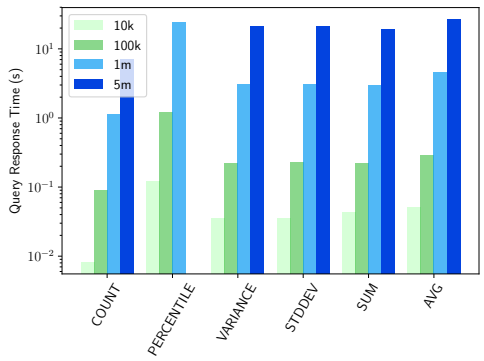


Figure 3: Influence of Sample Size on Response Time

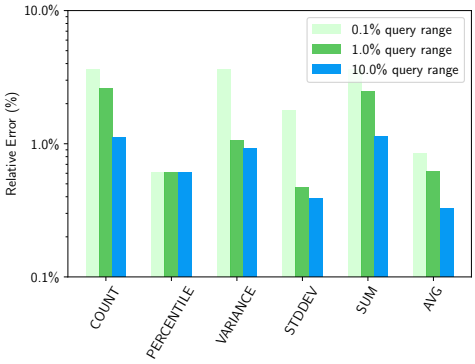
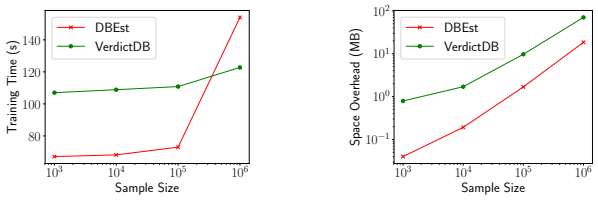


Figure 5: Influence of Query Range on Relative Error



(a) Training Time **(b) Space Overhead**

Figure 4: DBEst vs VerdictDB Overheads

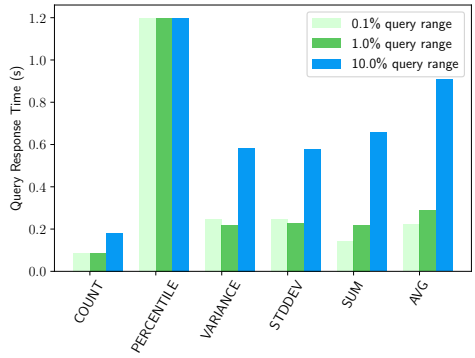


Figure 6: Influence of Query Range on Response Time

Results show that envisaged DBEst sample sizes, yield big improvements in state building times compared to VerdictDB.

Fig. 4(b) shows the space overhead of DBEst and VerdictDB. Recall, 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.

4.2.2 Query Range Effect.
Sample sizes are fixed at 100k, and query ranges vary (0.1%, 1% to 10%). In Fig. 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.

Fig. 6 shows response times. Except for `PERCENTILE`, (as multiple 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.

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.

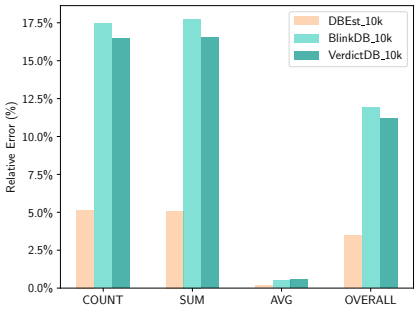


Figure 7: Relative Error: CCPP Dataset (10k Sample)

Fig. 7 and Fig. 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

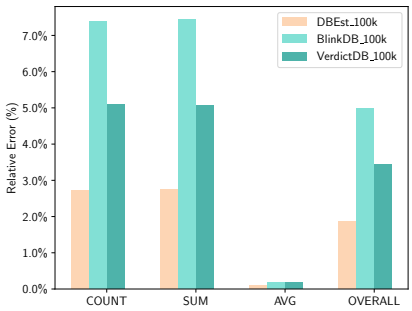


Figure 8: Relative Error: CCPP Dataset (100k Sample)

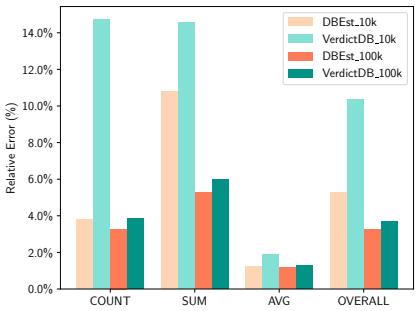
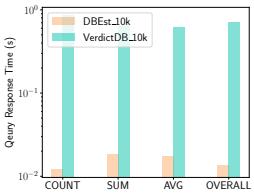


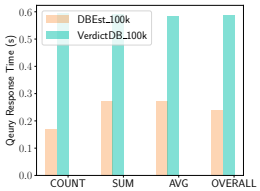
Figure 10: Relative Error: DBEst vs VerdictDB

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.

Fig. 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 9: Response Time for CCPP Dataset

We also conduct the experiments using MonetDB [25], and the comprehensive comparison results between DBEst and MonetDB are shown in Appendix C.

4.4 TPC-DS Workload Performance

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

4.4.1 Accuracy.

Fig. 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, 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.

4.4.2 Query Response Time.

Fig. 11 shows corresponding query times of DBEst and VerdictDB. For 10k samples, DBEst takes less than 0.02 seconds

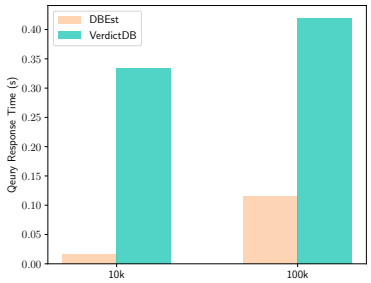


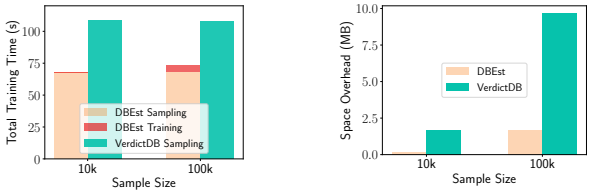
Figure 11: Response Time: DBEst vs VerdictDB

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.

4.4.3 Overheads.

We now evaluate both the training time and space overheads of DBEst and VerdictDB. Fig. 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.



(a) Training Time **(b) Space Overhead**

Figure 12: Overheads: DBEst vs VerdictDB

Fig. 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 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.

4.5 Beijing Workload Performance

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

Fig. 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

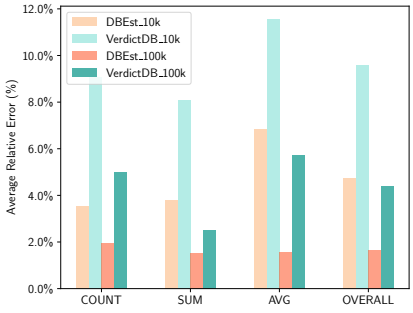


Figure 13: Accuracy: DBEst vs VerdictDB

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.

Fig. 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. 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.

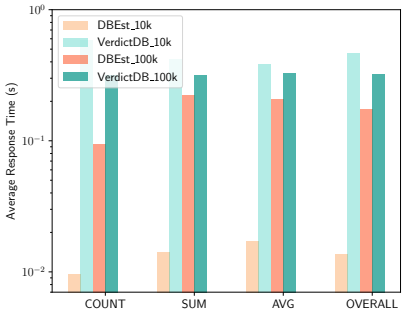


Figure 14: Response Time: DBEst vs VerdictDB

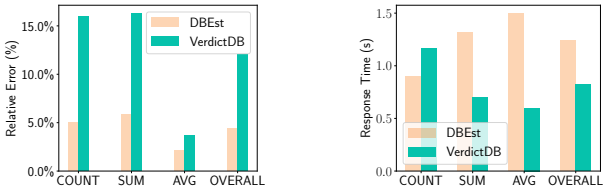
4.6 TPC-DS Group By Performance

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.

Fig. 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.

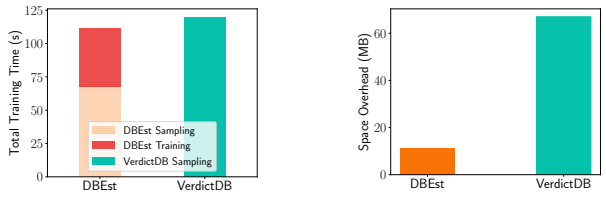
Query response times are shown in Fig. 15(b). VerdictDB takes slightly less time than DBEst for a `GROUP BY` query. Note, VerdictDB uses all cores, while DBEst only uses one. §4.7 will show a DBEst with parallel `GROUP BY` processing.

Figure 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.



(a) Relative Error (b) Response Time

Figure 15: Query Performance for 57 Group Values



(a) Training Time (b) Space Overhead

Figure 16: Overheads for 57 Group Values

We now study error performance for individual groups. Fig. 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. Fig. 22 in Appendix A shows the histograms and same conclusions for COUNT and AVG.

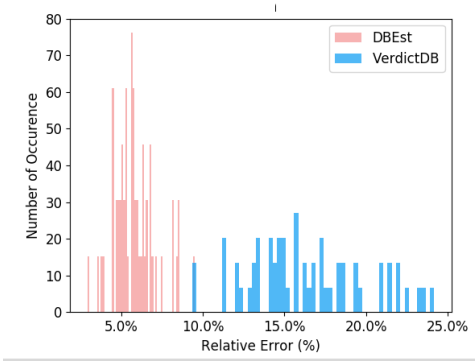


Figure 17: Accuracy Histogram: SUM for 57 Groups

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.

4.7.1 *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.

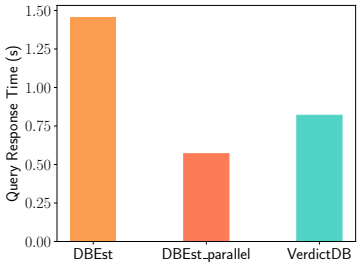


Figure 18: Group By Query Response Time Reduction

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 Fig. 18.

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. 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.

4.7.2 *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.

Fig. 19 displays the impact of the number of cores on the total query response time. With DBEst, time decreases as more

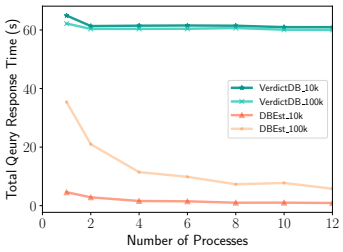


Figure 19: Throughput of Parallel Execution (CCPP)

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. The same experiments are run for the TPC-DS and Beijing PM2.5 datasets, and the same conclusion hold, see Appendix B for more information. DBEst improves throughput by ca. 6x to 30x.

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`. 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.

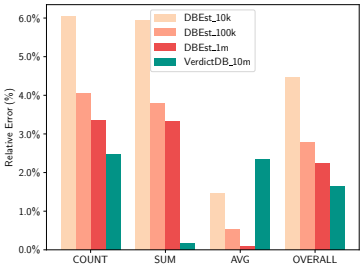
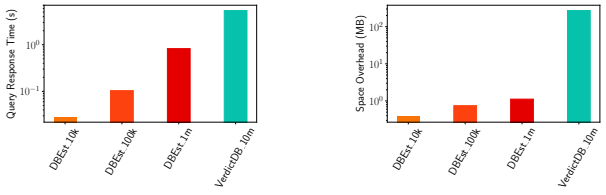


Figure 20: Join Accuracy Comparison

Fig. 20 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%). Appendix D will show cases where VerdictDB has a higher error than DBEst. Appendix C will also show how robust DBEst accuracy is for joins even when stressed with skewed join-attribute distributions, unlike other approaches.

Turning to Fig. 21, for 10k samples, DBEst needs only 0.028s and 0.37MB. For 1m samples, it needs 0.82s and 1.12MB.



(a) Response Time **(b) Space Overhead**

Figure 21: Join Performance Comparison

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.

4.9 Major Lessons

The major lessons learned are as follows: DBEst

- brings big (orders of magnitude) gains in query processing times. Even single-threaded DBEst often outperforms a 12-core installation of a state of the art AQP engine.
- allows for inter-query parallelism, which can lead to up to 30x improvement in throughput.
- is particularly frugal in system resource demands, e.g., in cores and memory, while achieving low response times.
- ensures high accuracy with small per-group variance, with small overheads (state building times, and space).
- can trade-off accuracy vs efficiency: multiple models with different-size samples can be developed and the one that caters to efficiency vs accuracy targets can be used.
- operation (sampling, training, and evaluation) can be parallelized, making it suitable for scale-out/up installations.
- can process joins with up to ca. 250x speedups, compared to the state of the art. Furthermore, its accuracy is high even when stress-tested, with skewed and non-skewed join-attribute distributions, unlike related work.
- is significantly space/time-challenged with large numbers of groups in `Group By`. To ameliorate this, DBEst can create model bundles, serialize and store them, say, in SSDs. Thus only the small state crucial for the query at hand is needed in memory. Our experiments show the time for IO and deserialization is very small (ca. 100ms for 500-group bundles), ensuring still large speedups. Additionally, the bundles can be processed in parallel by different cores and/or different nodes in a scale-out cluster, able to attain thus sub-second latency even for large group numbers.

5 RELATED WORK

Big data infrastructures, (e.g., SPARK [60], Hadoop [17], and TEZ [47]) and QP engines over them, such as Hive [55],

Spark SQL [8], Impala [32], Amazon Redshift [23], and Shark [19] have been a catalyst for analytical query processing. Efficient analytical QP engines with columnar data representations have also been developed (e.g., MonetDB [25] or, for streaming environments, Trill [18], which can also run in distributed .NET environments, (Orleans [9]). A related thread concerns applying data pre-fetching techniques [26], including semantic windows [28], or developing caches for analytical query results e.g., Data Canopy [59].

With respect to AQP research, the existing solution space is quite complex. Some approaches based on data sketches have received considerable attention [14–16]. Others focus on progressive/online aggregation [13, 18, 24, 37, 46]. Nonetheless, AQP research has been largely dominated by sampling-based approaches [1, 2, 4, 10, 20, 22, 29, 38, 40, 41, 44]. A different perspective is to think of forgetting data (items). DBs with 'amnesia'[31] 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.

State of the art sampling-based AQP approaches are broadly divided into two categories and in general no single approach is a 'silver buller'[53]. 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., [29]) only deliver a ca. 2x speedup. The second category can bring much bigger speedups [1, 4, 10, 38, 41] 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 [10] 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 [4] showed that such templates can be identified from a small prefix of a workload. VerdictDB [38] 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.

Works on predictable queries with prebuilt, a priori samples are closest to DBEst. BlinkDB [4] relies on uniform and stratified sampling and can trade-off performance vs accuracy, while it supports the `COUNT, SUM, AVG` AFs. DBL [41], builds a 'learning'layer on top of AQPs (like BlinkDB) in an effort to learn how to reduce errors. VerdictDB [38] 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. DBEst extends the state of the art in this domain. It uniquely combines regression and density estimator 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 DBEst does not provide a priori error guarantees.

With respect to model-based approaches, [51] 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, [54] 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. More recent research on applying ML models include DBL [50] and [5–7]. Also [49] uses regression models to explain approximate query answers. Recent work on learning to forecast workloads [35] may be helpful to the above works. DBEst builds SML models and queries are answered without any sample, or base data accesses and DBEst models are "first class citizens", unlike DBL, being the only way to answer queries. Unlike [5–7] it does not depend on a large number of prior queries to learn while it handles many, not just one AF.

6 CONCLUSION

With this paper 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. The paper 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. Future work focuses on sampling strategies, density estimators, and regression models offering better efficiency-overheads-accuracy trade-offs, SML models for advanced selection operations on nominal categorical attributes, and better integration of DBEst with other AQP and exact-answer engines.

ACKNOWLEDGMENTS

This work was supported in part by the the 'Tools, Practices and Systems'theme of the UKRI Strategic Priorities Fund (EPSRC Grant EP/T001569/1) & The Alan Turing Institute (EPSRC grant EP/N510129/1).

REFERENCES

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

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

[3] D Adams. 2014. Oracle Database Online Documentation 12c Release 1 (12.1). *Application Development* (2014).

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

[5] C. Anagnostopoulos and P. Triantafillou. 2015. Learning Set Cardinality in Distance Nearest Neighbours. In *Proceeding of IEEE International Conference on Data Mining, (ICDM15)*.

[6] C. Anagnostopoulos and P. Triantafillou. 2017. Efficient Scalable Accurate Regression Queries in In-DBMS Analytics. In *Proceeding of IEEE International Conference on Data Engineering, (ICDE17)*.

[7] C. Anagnostopoulos and P. Triantafillou. 2017. Query-Driven Learning for Predictive Analytics of Data Subspace Cardinality. *ACM Trans. on Knowledge Discovery from Data, (ACM TKDD)* (2017).

[8] 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. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 1383–1394.

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

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

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

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

[13] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

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

[15] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

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

[17] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[18] B. Chandramouli et al. 2014. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment.

[19] C. Engle et al. 2012. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *Proceeding of ACM SIGMOD*.

[20] P. B. Gibbons et al. 1998. AQUA: System and Techniques for Approximate Query Answering. *Bell Laboratories* (1998).

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

[22] W. Gatterbauer and D. Suciu. 2015. Approximate lifted inference with probabilistic databases. In *Proceedings of VLDB*.

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

[24] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Acm Sigmod Record*, Vol. 26. ACM, 171–182.

[25] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database. (2012).

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

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

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

[29] S. Kandula. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*. ACM.

[30] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*. 3146–3154.

[31] Martin Kersten and Lefteris Sidirourgos. 2017. A Database System with Amnesia. In *Proceeding of CIDR*.

[32] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop.. In *Cidr*, Vol. 1. 9.

[33] Xuan Liang, Tao Zou, Bin Guo, Shuo Li, Haozhe Zhang, Shuyi Zhang, Hui Huang, and Song Xi Chen. 2015. 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 (2015), 20150257.

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

[35] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 631–645.

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

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

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

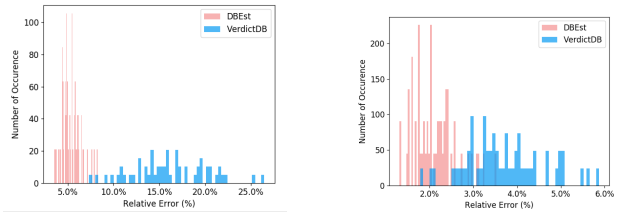
[39] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2019. VerdictDB. https://verdictdb.org/. (Accessed on 02/16/2019).

[40] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes

smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 587–602.
[41] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 587–602.
[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[43] Robert Piessens, Elise de Doncker-Kapenga, Christoph W Überhuber, and David K Kahaner. 2012. *QUADPACK: a subroutine package for automatic integration*. Vol. 1. Springer Science & Business Media.
[44] N. Potti and J. M. Patel. 2015. Daq: a new paradigm for approximate query processing. In *Proceedings of VLDB*.
[45] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2017. CatBoost: unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516* (2017).
[46] B. C. Ooi S. Wu and K.-L. Tan. 2010. Continuous Sampling for Online Aggregation over Multiple Queries. In *Proceedings of Acm Sigmod*. ACM.
[47] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. 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*. ACM, 1357–1369.
[48] Ariel Kleiner Ameet Talwalkar Sameer Agarwal, Henry Milner. 2013. BlinkDB: Sub-Second Approximate Queries on Very Large Data. https://github.com/sameeragarwal/blinkdb.
[49] F. Savva, C. Anagnostopoulos, and P. Triantafillou. 2018. Explaining Aggregates for Exploratory Analytics. In *Proceeding of IEEE International Conference on Big Data*.
[50] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 3–18.
[51] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. 1999. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. In *Proceeding of ACM SIGKDD*.
[52] Bernard W Silverman. 2018. *Density estimation for statistics and data analysis*. Routledge.
[53] Srikanth Kandula Surajit Chaudhuri, Bolin Ding. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM SIGMOD international conference on Management of data*.
[54] A. Thiagarajan and S. Madden. 2008. Querying continuous functions in a database system. In *ACM SIGMOD*.
[55] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.
[56] Srikanta Tirthapura and David P Woodruff. 2011. Optimal random sampling from distributed streams revisited. In *International Symposium on Distributed Computing*. Springer, 283–297.
[57] Pınar Tüfekci. 2014. 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 (2014), 126–140.
[58] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.
[59] Abdul Wasay, Xinding Wei, Niv Dayan, and Stratos Idreos. 2017. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 557–572.
[60] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.
[61] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1525–1539.

A DETAILED ANALYSIS: GROUP BY

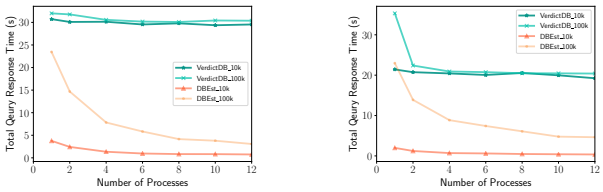
§4.6 shows in detail the error distribution for `GROUP BY` queries involving `SUM`. Here, we demonstrate the same for AFs `COUNT` and `AVG`, as shown in Fig. 22.



(a) COUNT (b) AVG

Figure 22: Accuracy Histogram for 57 GROUPS

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. The same conclusion holds for AF `AVG` as well.

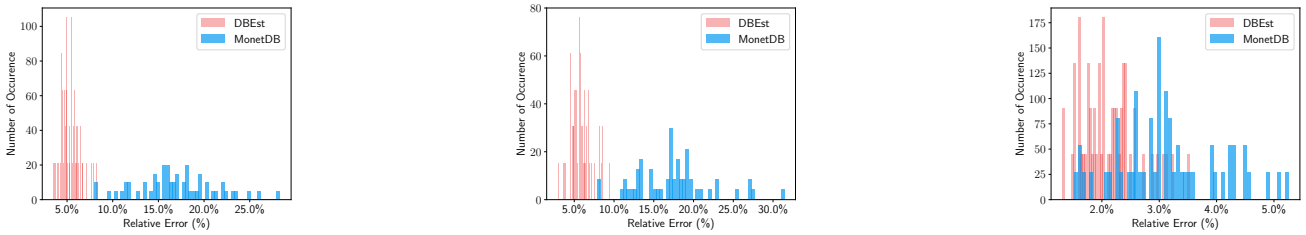


(a) TPC-DS (b) Beijing PM2.5

Figure 23: Throughput with Parallel Query Execution

B THROUGHPUT WITH INTER-QUERY PARALLELISM

§4.7.2 analyzes the total workload response time for CCPP. This section further demonstrate DBEst's abilities in processing queries for the TPC-DS and Beijing PM2.5 Datasets.



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

Figure 24: Error Histogram vs MonetDB: TPC-DS GROUP By Workload

Fig. 23(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, (Fig. 23(b)). We observe speedups up to ca. 10x by utilizing all available cores.

C 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) [20] 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, Fig. 25 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 is

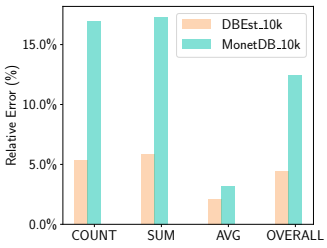


Figure 25: Error vs MonetDB : TPC-DS Group By

12.46%. To shed more light into accuracy-performance, the histograms of relative errors for `COUNT`, `SUM` and `AVG` are summarised in Fig. 24. 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.

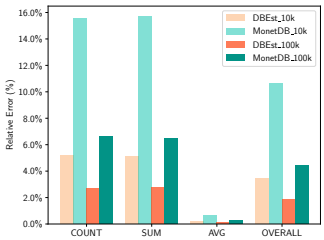


Figure 26: Error vs MonetDB: CCPP Workload

Fig. 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).

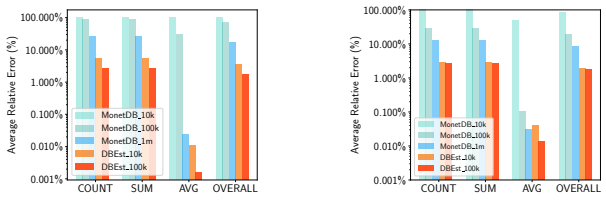
Approximate MonetDB and Joins

We now study approximate MonetDB for join queries. [11, 61] 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. Fig. 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. Even with 1m samples,



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

Figure 27: Accuracy Comparison for Join Queries

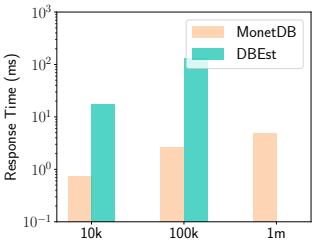


Figure 28: Query Response Time Comparison

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%.

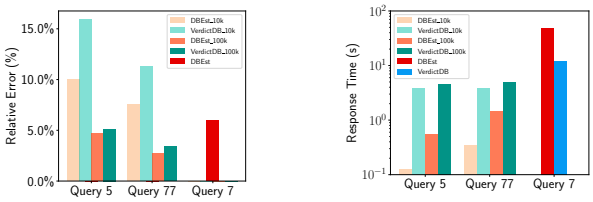
Fig. 28 shows that MonetDB is much more efficient than DBEst: for 10k (100k) sample, DBEst takes 17.57ms (129ms), while MonetDB only requires 0.74ms (2.65ms). This is as expected, since MonetDB has been optimized for over 2 decades now (and it is also written in C which is inherently much faster than Python). Nonetheless, DBEst's time is in absolute terms very fast, while also guaranteeing high accuracy. This showcases DBEsts' all-around strong performance.

D 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 Fig. 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 Fig. 20. 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%.



(a) Relative Error (b) Response Time

Figure 29: Performance for TPC-DS Queries 5, 7, 77

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