NeuroSketch: Fast and Approximate Evaluation of Range Aggregate Queries with Neural Networks

SEPANTA ZEIGHAMI, University of Southern California, USA
CYRUS SHAHABI, University of Southern California, USA
VATSAL SHARAN, University of Southern California, USA

Range aggregate queries (RAQs) are an integral part of many real-world applications, where, often, fast and approximate answers for the queries are desired. Recent work has studied answering RAQs using machine learning (ML) models, where a model of the data is learned to answer the queries. However, there is no theoretical understanding of why and when the ML based approaches perform well. Furthermore, since the ML approaches model the data, they fail to capitalize on any query specific information to improve performance in practice. In this paper, we focus on modeling "queries" rather than data and train neural networks to learn the query answers. This change of focus allows us to theoretically study our ML approach to provide a distribution and query dependent error bound for neural networks when answering RAQs. We confirm our theoretical results by developing NeuroSketch, a neural network framework to answer RAQs in practice. Extensive experimental study on real-world, TPC-benchmark and synthetic datasets show that NeuroSketch answers RAQs multiple orders of magnitude faster than state-of-the-art and with better accuracy.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Approximate Query Processing, Machine Learning, Theory of Learned Databases

ACM Reference Format:
Sepanta Zeighami, Cyrus Shahabi, and Vatsal Sharan. 2023. NeuroSketch: Fast and Approximate Evaluation of Range Aggregate Queries with Neural Networks. *Proc. ACM Manag. Data* 1, 1, Article 100 (May 2023), 26 pages. https://doi.org/10.1145/3588954

1 INTRODUCTION

Range aggregate queries (RAQs) are intrinsic to many real-world applications, e.g., calculating net profit for a period from sales records or average pollution level for different regions for city planing [23]. Due to large volume of data, exact answers can take too long to compute and fast approximate answers may be preferred. In such scenarios, there is a time/space/accuracy trade-off, where algorithms can sacrifice accuracy for time or space. For example, consider a geospatial database containing latitude and longitude of location signals of individuals and, for each location signal, the duration the individual stayed in that location. A potential RAQ on this database, useful for understanding the popularity of different Points of Interests, is to calculate the average time spent by users in an area. Approximate answers within a few minutes of the exact answer can be acceptable in such applications. We use this scenario as our running example.

Authors' addresses: Sepanta Zeighami, University of Southern California, USA, zeighami@usc.edu; Cyrus Shahabi, University of Southern California, USA, shahabi@usc.edu; Vatsal Sharan, University of Southern California, USA, vsharan@usc.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/5-ART100
https://doi.org/10.1145/3588954

Research on RAQs has focused on improving the time/space/accuracy trade-offs. Various methods such as histograms, wavelets and data sketches (see [13] for a survey) have been proposed to model the data for this purpose. Recent efforts use machine learning (ML) [15, 23, 35] to improve the performance. Such approaches learn models of the data to answer RAQs. Experimental results show ML-based methods outperform non-learning methods in practice.

Nonetheless, there is no theoretical understanding of when and why an ML based approach performs well. This is because modeling data makes it difficult to reason about the performance of specific queries. That is, some queries may be easier to answer than others, e.g., average value of one attribute may be constant for different query ranges, while that of another attribute might change drastically. Furthermore, modelling the data misses the opportunity to utilize information about queries in practice. For instance, patterns in query answers can be used to learn a compact representation of the data *with respect to the queries*, improving the performance, while there may be no such patterns within the entire dataset.

In this paper, instead of learning *data models*, we propose to learn *query models*. In our example of calculating the average visit duration for a POI, the input to a query model is the POI location and the model is trained to output the average visit duration for the POI. Query modeling skips learning explicitly the data distribution and instead learns query answers, so that we can explicitly relate errors in modeling to errors in query answering. Nevertheless, this is non-trivial and requires a detailed study of modelling errors. To the best of our knowledge, no existing attempt in the literature theoretically relates data and query properties to the error of a learned model when answering RAQs.

We utilize neural networks as our query model. Specifically, we consider training a neural network that takes as input an RAQ and outputs the answer to the query. We theoretically study this approach, and provide, for the first time, a *Data distribution and Query Dependent error bound* (hereafter referred to as DQD bound) for neural networks when answering RAQs. DQD bound theoretically relates properties of the data distribution and the RAQ to the accuracy a neural network can achieve when answering the query.

In our theoretical analysis, we consider `AVG`, `COUNT` and `SUM` queries, assume the database is a collection of i.i.d samples from a data distribution and make a suitable Lipschitz assumption on the query and data distribution. We then use VC-sampling theory and our novel result on neural network approximation power to show the existence of a neural network that can answer the queries on the database with bounded error. The bound gets tighter (i.e., more accurate neural networks can be learned) as the data size, or query range, increases. Alternatively, a smaller neural network can be used to answer queries with a fixed desired error when the data size, or query range, increases. Intuitively, this is a result of the reduction in variance (due to sampling) of query answers when the database is larger, because more data points are sampled from the data distribution. Furthermore, our results utilize the Lipschitz property to provide a complexity measure that quantifies the difficulty of answering a query from a data distribution. Using the complexity measure, our results show settings where existence of a small neural network with low query answering error is guaranteed.

To confirm our theoretical results, we design *NeuroSketch*, a neural network framework that answers RAQs orders of magnitude faster than state-of-the-art and with better accuracy. NeuroSketch uses DQD results to allocate more model capacity to queries that are difficult to answer, thereby reducing error without increasing query time. While DQD provides a theoretical grounding for NeuroSketch, in practice NeuroSketch is not limited to some of the assumptions we made to prove DQD bounds, for example, it can answer more general RAQs, such as `STD` and `MEDIAN`.

To summarize, our major contributions are:

- We present the first theoretical analysis for using ML to answer RAQs. This includes a novel analysis framework, a novel use of VC-sampling theory and a novel result on neural network approximation power.
- We show theoretically how data distribution, data size, query range and aggregation function are related to the neural network error when answering RAQs. This opens the possibility for a query optimizer that, for a data distribution, decides when to build and use a neural network for query processing.
- To confirm our theoretical results, we design *NeuroSketch*, the first neural network framework to answer generic RAQs.
- Extensive experiments show that NeuroSketch provides orders of magnitude gain in query time and better accuracy over state-of-the-art, (DBEst [23] and DeepDB [15]) on real-world, TPC-benchmark and synthetic datasets.

We present our problem definition in Sec. 2, DQD bound in Sec. 3, NeuroSketch in Sec. 4, our empirical study in Sec. 5, related work in Sec. 6 and conclude in Sec. 7.

2 PROBLEM DEFINITION

Problem Setting. Consider a dataset D with n records and $\bar{d}$ attributes, A_1, ..., $A_{\bar{d}}$. Assume records of D are random i.i.d samples from a data distribution χ and $A_i \in [0, 1]$ with probability 1 for all $1 \leq i \leq \bar{d}$ (otherwise the attributes can be normalized). We first consider the following SQL query and discuss extensions to general RAQs in Sec.4.3.

```
SELECT AGG(A_m) FROM D WHERE
c_1 ≤ A_1 < c_1 + r_1  AND  ...  AND  c_d̄ ≤ A_d̄ < c_d̄ + r_d̄
```

For any i, c_i and $c_i + r_i$ are the lower and upper bounds on the attribute A_i. c_i and r_i can be 0 and 1, respectively, in which case there are no restrictions on the values of A_i in the query. We say that an attribute is *not active* in the query in that case, and is *active* otherwise. AGG is a user defined *aggregation function*, with examples including SUM, AVG and COUNT aggregation functions. A_m is called the *measure attribute*, where m is an integer between 1 and $\bar{d}$. Let $\mathbf{c} = (c_1, ..., c_{\bar{d}})$ and $\mathbf{r} = (r_1, ..., r_{\bar{d}})$ be $\bar{d}$-dimensional vectors. We call the pair $\mathbf{q} = (\mathbf{c}, \mathbf{r})$ a *query instance*. Different query instances correspond to different range predicates for the measure attribute A_m and aggregation function AGG. We define the function $f_D(.)$ so that for a query $\mathbf{q}$, $f_D(\mathbf{q})$ is the answer to the above SQL statement. We call $f_D : [0, 1]^d \rightarrow \mathbb{R}$ a *query function*, where $d = 2\bar{d}$ is the dimensionality of the query function. Furthermore, we define $\mathcal{Q} = \{(\mathbf{c}, \mathbf{r}) \in [0, 1]^d, c_i + r_i \leq 1 \forall i\}$ as the set of all possible queries.

Example 2.1. Consider a database of user location reports and the duration a user stayed in the reported location, shown in Fig. 1 (left). On this database, consider the RAQ of returning average visit duration of users in a 50m×50m rectangle with bottom left corner at the geo-coordinate (c_1, c_2). The query function, $f_D(c_1, c_2) := f_D(c_1, c_2, 50m, 50m)$, takes as input the geo-coordinate of the rectangle and outputs the average visit duration of data points in the rectangle (we have fixed r_1 and r_2 to 50m in this example). Fig. 1 (right) plots $f_D(c_1, c_2)$, which shows, $f_D(-95.3615, 29.758, 50m, 50m) = 9$, i.e., for query instance (-95.3615, 29.758, 50m, 50m) the answer is 9.

Neural Networks to Answer RAQs We learn a neural network, $\hat{f}(.; \theta)$, to approximate the query function, $f_D(.)$. The neural network takes as input an RAQ, $\mathbf{q}$. The model forward pass outputs an answer, $\hat{f}(\mathbf{q}; \theta)$. The goal is to train a neural network so that its answer to the query, $\hat{f}(\mathbf{q}; \theta)$, is similar to the ground-truth, $f_D(\mathbf{q})$. If such a neural network is small and can be evaluated fast, we can use the neural network to directly answer the RAQ efficiently and accurately, by performing a forward pass of the model.

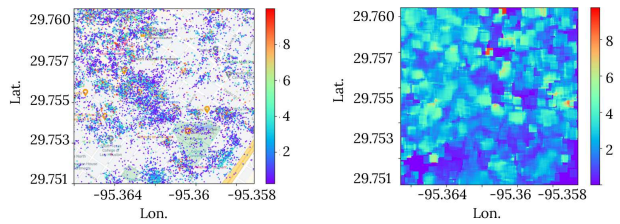

Fig. 1. (left) Database of location signals. (right) Avg. visit duration query function. Color shows visit duration in hours

Problem Statement. Let $\Sigma(\hat{f})$ be the space complexity of the neural network, which is the amount of space required to store all its parameters. Let $\tau(\hat{f})$ be its query time complexity, which is the time it takes to perform a forward pass of the neural network. We study the error, $\|f_D - \hat{f}\|$, in answering queries, where we mostly consider the 1-norm, defined as $\|f_D - \hat{f}\|_1 = \int_{\mathbf{q}\in\mathcal{Q}} |f_D(\mathbf{q}) - \hat{f}(\mathbf{q})|$ or ∞-norm, defined as $\|f_D - \hat{f}\|_\infty = \sup_{\mathbf{q}\in\mathcal{Q}} |f_D(\mathbf{q}) - \hat{f}(\mathbf{q})|$. The problem studied in this paper is learning to answer range aggregate queries with time and space constraints, formulated as follows.

Problem 1. *Given a query function f_D, class of possible neural networks, $\mathcal{F}$, and time and space requirements t and s, find*

$$\arg\min_{\hat{f}\in\mathcal{F}} \|f_D - \hat{f}\| \text{ s.t. } \Sigma(\hat{f}) \leq s, \tau(\hat{f}) \leq t.$$

Notation. Bold face letters, e.g., $\mathbf{c}$, denote vectors, and subscripts denote the elements of a vector, e.g., c_i is the i-th element of $\mathbf{c}$.

3 DQD BOUND FOR NEURAL NETWORKS ANSWERING RAQS

We theoretically study the relationship between the accuracy a neural network can achieve when answering RAQs and data and query properties. Sec. 3.1, states our Data distribution and Query Dependent error bound (DQD bound) when considering SUM and COUNT aggregation functions, and discusses its implications. We prove the bound in Sec. 3.2. We present results for AVG query function in Sec. 3.3 and discuss how our techniques can be generalized to other query functions and modelling choices.

3.1 DQD Bound Statement

3.1.1 Incorporating Data Distribution. The data distribution, χ, underlying a database, D, impacts the difficulty of answering queries on the database with a neural network. For instance, in Example 2.1, if all users have the same visit duration for all their visits, the query function $f_D(c_1, c_2)$ will be constant, and thus can be easily modeled. On the other hand, the skewness in the data distribution, as depicted in Fig. 1, can make answering queries more difficult as the query function $f_D(c_1, c_2)$ changes drastically from one location to another. Importantly, this is a property of the data distribution, χ, and not only of the observed database D. For instance, we expect similar skewness in observations if we collect more user data (i.e., as D grows), or if location data are collected from a different period of time not covered in D (i.e., a different sample of χ). Thus, by incorporating data distribution in our analysis, we are able to study the impact of data size as well as properties intrinsic to the distribution (that will be unaffected by the randomness in observations) on answering RAQs. To do so, (1) we need to capture the dependence of query answers on data distribution and (2) find a means of measuring the complexity of modeling query answers when data follows a certain distribution.

To capture the dependence on data distribution, we define *distribution query function*, $f_\chi(\mathbf{q})$, as the expected value of the query function, i.e., $f_\chi(\mathbf{q}) = \mathbb{E}_{D\sim\chi}[f_D(\mathbf{q})]$, where D is sampled from data

distribution, χ. We refer to the query function, $f_D(\mathbf{q})$, as *observed query function* to distinguish it from distribution query function.

To capture the difficulty of modeling a function, we use the ρ-Lipschitz property. A function, f, is ρ-Lipschitz if $|f(\mathbf{x}) - f(\mathbf{x}')| \leq \rho\|\mathbf{x} - \mathbf{x}'\|_1$, for all $\mathbf{x}$ and $\mathbf{x}'$ in the domain of the function, where we consider ρ-Lipschitz property in 1-norm. Intuitively, ρ captures the magnitude of correlation between $\mathbf{x}$ and $f(\mathbf{x})$. It bounds how much $f(\mathbf{x})$ can change with a change in $\mathbf{x}$. If ρ is large, f can change abruptly even with a small change in $\mathbf{x}$. This makes the function more difficult to approximate, as more model parameters will be needed to account for all such possible abrupt changes.

Combining the above, we propose to use the Lipschitz constant of the normalized Distribution Query function, referred to as *LDQ*, as a measure of difficulty of answering RAQs. LDQ is the Lipschitz constant of the function $f(\mathbf{q}) = \frac{f_\chi(\mathbf{q})}{n} = \frac{1}{n}\mathbb{E}_{D\sim\chi}[f_D(\mathbf{q})]$. We normalize the distribution query function by data size to account for its change in magnitude when data size increases (for sum and count queries, magnitude of $f_D(\mathbf{q})$ increases as data size increase). LDQ is a property of χ and f_D. For ease of reference, we often implicitly assume a given data distribution χ and refer to LDQ as a property of a query function.

3.1.2 Theorem Statement. Let f_D^S and f_D^C be query functions with aggregation functions `SUM` and `COUNT`, respectively, and let ρ_S and ρ_C be their respective LDQs. For $i \in \{S, C\}$, we study the time, space and accuracy of a neural network, $\hat{f}$, when approximating f_D^i, as formalized below.

Theorem 3.1 (DQD Bound). *For $i \in \{S, C\}$, there exists a neural network $\hat{f}$ with space and query time complexity $\tilde{O}(d(\varkappa\rho d\varepsilon_1^{-1} + 1)^d)$, where $\tilde{O}$ hides logarithmic factors, s.t.*

$$\mathop{\mathbb{P}}_{D\sim\chi}\left[\frac{1}{n}\|\hat{f} - f_D^i\|_1 \geq \varepsilon_1 + \varepsilon_2\right] \leq \varkappa_2^{d+1} d\varepsilon_2^{-d} \exp\left(-\varkappa_2^{-1}\varepsilon_2^2 n\right),$$

Where $\varkappa_1$ and $\varkappa_2$ are universal constant.

Proof of Theorem 3.1 is presented in Sec. 3.2. Here, we discuss the theorem statement and its implications.

A Confidence/Error Analysis. DQD bound relates, with a desired probability (i.e., confidence level), error a neural network can achieve when answering RAQs to its query time and space complexity through data dependent properties. The error is scaled by data size, n, to account for the change in the magnitude of query answers when data size changes. Parameter ε_1 allows trading-off accuracy for space or time complexity and ε_2 allows trading-off accuracy for confidence in the bound. The probability is over sampling a database from the data distribution. That is, DQD states that, when observing a database D that follows a distribution χ, with high probability, there exists a neural network that can answer RAQs on D and achieve the specified time/space/accuracy trade-off.

Distribution Dependent Complexity Measure. DQD bound establishes LDQ of the query function as a measure of complexity when answering RAQs with neural networks. It implies that query time will be faster when LDQ is small. LDQ is a property of the data distribution and the query in question. Thus, Theorem 3.1 allows us to quantify how easy or difficult it is to approximate query answers for a data distribution using a neural network. We provide specific examples of LDQ for different data distributions in Sec. 3.1.3 and empirically confirm impact of LDQ on query answering in Sec. 5.7.

Faster on Larger Databases. Let the confidence in the DQD bound be $\delta = \varkappa_2^{d+1} d\varepsilon_2^{-d} \exp\left(-\varkappa_2^{-1}\varepsilon_2^2 n\right)$. Fixing the value of δ, we observe that n and ε_2 are negatively correlated, where increasing data size n leads to reduction in ε_2. That is, for a fixed confidence parameter, the error of a neural network decreases as data size increases. Now let $\varepsilon = \varepsilon_1 + \varepsilon_2$ be the total neural network error. Also fixing ε in addition to δ but allowing ε_1 to vary, we observe that increase in data size results in smaller

query time and space complexity, for a fixed neural network error and confidence level. Thus, DQD bound shows the counter-intuitive result that when answering queries with a neural network query time can be lowered by increasing the database size. We empirically confirm this phenomenon in Sec. 5.7. Intuitively, this happens because when data size is larger the model only needs to learn the patterns in the data distribution, while for smaller databases, the observed database can be different from the data distribution and the model has to memorize all the points, making it more challenging.

Low-Error Cases. DQD bound shows that a neural network can answer queries fast and accurately if the data size is large and LDQ of a query function is small. Thus, DQD bound shows scenarios when using a neural network can provide good performance and presents a property of data distribution that can guarantee low error for a neural network framework when answering RAQs. Nonetheless, it does not preclude neural networks from performing well in other scenarios, which requires further theoretical investigation.

Achieving Zero Error. For a fixed and small data size, even if neural network size is allowed to approach infinity, the DQD bound provides a non-zero error bound. That is, letting neural network size go to infinity by reducing ε_1 to zero does not achieve total zero error (we empirically verify this in Sec. 5.7), as the total error in that case will be equal to ε_2 (which can be large depending on n). This is because f_D can be discontinuous even though f_χ is assumed to be Lipschitz continuous, so that no neural network can approximate it exactly. Points of discontinuity can be seen in Fig. 1 (right), where the query answer can suddenly change. Such points of discontinuity happen when the query boundary matches a data point, because in such cases, arbitrarily small changes to the query boundary can change the query answer. As data size increases, f_D behaves more like a continuous function (because f_χ is Lipschitz continuous), so the achievable error by a neural network goes down. Note that techniques that create a discontiuous function approximator, e.g., quantizating the query space, can potentially help a neural network achieve zero error, as a large enough neural network can memorize a fininte set of points exactly [43]. However, our DQD bound is for queries over space of reals (i.e., approximation of infinite set of points), and without input preprocessing or quantization.

3.1.3 Impact of Distribution and LDQ. The model complexity needed to answer RAQs depends on data distribution through LDQ of f_D^S and f_D^C. We provide examples of LDQ for different distributions.

Example 3.2. Let χ be a 1-dimensional uniform distribution. By definition, we have $f_\chi^C(c_1, r_1) = n\mathbb{P}_{p\sim\chi}[p \in (c_1, r_1)]$, where (c_1, r_1) defines a query range (see Sec. 2) and p is a data point sampled from χ. χ is uniform so $\mathbb{P}_{p\sim\chi}[p \in (c_1, r_1)] = r_1$. Differentiating and using the definition, $\frac{\partial f_\chi^C(c_1,r_1)}{\partial c_1} = 0$ and $\frac{\partial f_\chi^C(c_1,r_1)}{\partial r_1} = n$, so that $\frac{1}{n} f_\chi^C(c_1, r_1)$ is ρ-Lipschitz with $\rho = 1$. A similar result also holds for $\frac{1}{n} f_\chi^S(c_1, r_1)$. The small Lipschitz constant matches the intuition that uniform distribution is easy to approximate.

Example 3.3. Let χ be a 1-dimensional Gaussian distribution with standard deviation σ and $\mu = 0$, we have that

$$\left|\frac{\partial \mathbb{P}_{p\sim\chi}[p \in (c_1, r_1)]}{\partial c_1}\right| = \left|\frac{1}{\sigma\sqrt{2\pi}}\left(e^{-\frac{1}{2}\left(\frac{c_1+r_1}{\sigma}\right)^2} - e^{-\frac{1}{2}\left(\frac{c_1}{\sigma}\right)^2}\right)\right| \leq \frac{2}{\sigma\sqrt{2\pi}}$$

and that

$$\left|\frac{\partial \mathbb{P}_{p\sim\chi}[p \in (c_1, r_1)]}{\partial r_1}\right| = \left|\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{c_1+r_1}{\sigma}\right)^2}\right| \leq \frac{1}{\sigma\sqrt{2\pi}}$$

so that $\frac{1}{n} f_\chi^C(c_1, r_1)$ is ρ-Lipschitz with $\rho = \frac{3}{\sigma\sqrt{2\pi}}$. Thus, for smaller σ the function becomes more difficult to model, as the neural network has to model a sharp change in the function.

3.1.4 Measuring Complexity in Practice. DQD bound can help decide whether to use neural networks to answer RAQs, or to design complexity aware algorithms for practical use-cases (as we do in Sec. 4). Such use-cases require measuring LDQ, which can be difficult in practice. For two queries $\mathbf{q}$ and $\mathbf{q}'$, the Lipschitz constant bounds the *maximum* change in the function, f, normalized by distance, $\frac{|f(\mathbf{q}) - f(\mathbf{q}')|}{\|\mathbf{q} - \mathbf{q}'\|}$. Since this maximum is calculated over all query pairs, it is difficult to estimate. Furthermore, it depends on the data distribution itself, while we only have access to samples from the distribution. In practice, we observed that the *Average* Query function Change, AQC, can be used as a proxy for LDQ. Specifically, we define $AQC = \frac{1}{\binom{|Q|}{2}} \sum_{\mathbf{q},\mathbf{q}' \in Q} \frac{|f(\mathbf{q}) - f(\mathbf{q}')|}{\|\mathbf{q} - \mathbf{q}'\|}$, where $Q \subseteq \mathcal{Q}$ is a set of queries sampled from all possible queries. We experimentally verify the usefulness of this complexity measure in Sec. 5.5.

3.2 DQD Bound Proof

3.2.1 Analysis Framework. For a neural network $\hat{f}$ when modelling a query function, f_D, we decompose its error, $\Delta = \frac{1}{n}\|f_D - \hat{f}\|_1$, into two terms, *approximation error* and *sampling error*:

$$\Delta \leq \underbrace{\frac{1}{n}\|f_\chi - \hat{f}\|_1}_{\text{approximation error, } \Delta_a} + \underbrace{\frac{1}{n}\|f_\chi - f_D\|_1}_{\text{sampling error, } \Delta_s} \tag{1}$$

Approximation error, Δ_a, quantifies how accurately the neural network can approximate the distribution query function. Δ_a depends on the space/time complexity of the neural network. For instance, larger neural networks have more representation power and can approximate a distribution query function more accurately. Sampling error, Δ_s, quantifies the difference, due to sampling, between the distribution and observed query functions. Δ_s depends on data size: the more data sampled, the more similar observed and distribution query functions will be (latter is the expected value of the former). We bound each term separately in Secs. 3.2.2 and 3.2.3. Sec. 3.2.4 combines the results which yields Theorem 3.1.

3.2.2 Bounding Approximation Error. For a desired bound on approximation error, Δ_a, we characterize the time/space complexity required for a neural network to achieve the error bound. Universal function approximation theorem [18, 29] guarantees existence of a neural network of *arbitrary* time/space complexity that can achieve any desired error value, but does not show its time/space complexity. Recent work (e.g., [22, 28, 41]) study *number* of neural network parameters needed to achieve a desired error. However, number of neural network parameters cannot be related to its space complexity, because magnitude of the parameters can be unbounded, thus leading to unbounded storage cost even for a fixed number of parameters. We present the following theorem, showing the required time/space complexity to achieve a desired error bound, ε_1 (see Sec. 6 for a comprehensive discussion of related work).

Theorem 3.4. *Given a ρ-Lipschitz function f, there exists a neural network, $\hat{f}$, with space and time complexity $\tilde{O}(d(\varkappa\rho d\varepsilon_1^{-1} + 1)^d)$, where $\tilde{O}$ hides logarithmic factors in ρ, d and k, such that*

(a) $\|f - \hat{f}\|_1 \leq \varepsilon_1$.

(b) Furthermore, if $d \leq 3$, $\|f - \hat{f}\|_\infty \leq \varepsilon_1$,

Where $\varkappa$ is a universal constant.

Theorem 3.4 (a) bounds Δ_a by considering f_χ as the function, f, in the theorem statement. Theorem 3.4 (b) provides a stronger guarantee that can provide an ∞-norm DQD bound in low dimensions. For conciseness, we have not stated that version of DQD bound since the ideas are

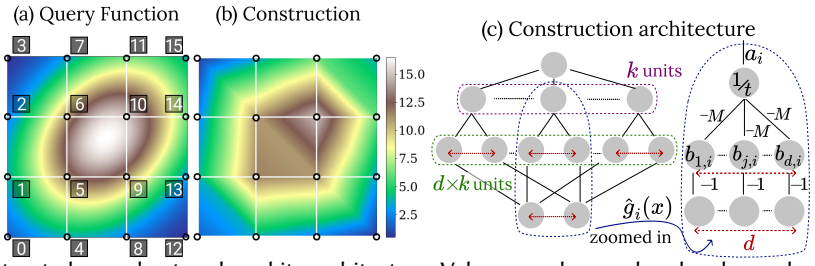

Fig. 2. Constructed neural network and its architecture. Values on edges and nodes show edge weight and unit bias.

similar. Theorem 3.4 is a step towards characterizing neural network approximation power in a data management context. We expect tighter characterizations to be possible, especially for high dimensions. Our theoretical framework for DQD bound can readily benefit from such tighter characterizations. Nonetheless, d is small for many practical applications when answering RAQs. For instance, in Example 2.1 that mimics a real-world use-case, the query function is 4-dimensional.

Proof Sketch of Theorem 3.4. We uniformly partition the space into cells and construct a neural network that estimates cell vertices exactly. This *memorization* property is used to bound error within each cell. For instance, Fig. 2 (a) shows the distribution query function for a COUNT query with fixed range $r = 0.1$ on a two-dimensional Gaussian data distribution. A 3x3 grid on input space creates 16 vertices, shown in Fig. 2 (a). Our construction ensures that the error for these 16 vertices is zero, as shown in Fig. 2 (b).

Network Architecture. We construct a ReLU neural network, $\hat{f}$, with two hidden layers, shown in Fig. 2 (c). $\hat{f}$ can be written as a summation of k smaller units, called g-units. Each g-unit ensures that a cell vertex is memorized correctly and k controls neural network size. The i-th g-unit, $\hat{g}_i$ for $1 \leq i \leq k$, is constructed as shown in Fig. 2 (c). It has d inputs, d units in its first layer and 1 unit in its second layer. Each input is only connected to one of the units in the first layer with weight -1. All units in the first layer are connected to the unit in the second layer, and their weight is $-M$, where M is a constant at least equal to 1. The j-th unit, $1 \leq j \leq d$ in first layer has bias $b_{j,i}$ and the unit in second layer has bias $\frac{1}{t}$ for an integer t. The output of the second unit is multiplied by a parameter a_i. Then, the neural network is $\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{k} \hat{g}_i(\boldsymbol{x}) + b$, where b is the bias of the third layer. The tunable parameters of the neural network are a_i, $b_{j,i}$, and b for $1 \leq i \leq k$ and $1 \leq j \leq d$.

Network Parameters. Let the set of cell vertices in the uniform grid be $P = \{(i_1, ..., i_d)/t, i_r \in \mathbb{Z}, 0 \leq i_r \leq t\}$, for an integer t so that $k = |P| = (t+1)^d$ (recall that input space is $[0,1]^d$). Also let $\boldsymbol{\pi}^i$ be the base $t+1$ representation of an integer i written as a vector, i.e., $\boldsymbol{\pi}^i = (\pi^i_1, ..., \pi^i_d)$ so that $i = \sum_{r=1}^{d} \pi^i_r (t+1)^{d-r}$. For example, when $t = 3$, $\pi^6 = (1, 2)$, since $6 = 1(t+1) + 2$. Note that $\frac{\boldsymbol{\pi}^i}{t} \in P$ and $\langle \frac{\boldsymbol{\pi}^0}{t}, ..., \frac{\boldsymbol{\pi}^{k-1}}{t} \rangle$ is an ordering of cell vertices. Alg. 1 enumerates using this ordering over the cell vertices and sets, at the i-th iteration, the parameters of the i-th g-unit so that $\frac{\boldsymbol{\pi}^i}{t}$ is correctly memorized. It calculates, $\hat{y}$, the estimate of the neural network for point $\frac{\boldsymbol{\pi}^i}{t}$ based on the g-units set before the i-th iteration (line 3). Then it sets the parameter of the i-th g-unit to account for the difference between $\hat{y}$ and the true value, $f(\frac{\boldsymbol{\pi}^i}{t})$. Fig. 3 shows this process in our example. On the left, Fig. 3 shows, at the end of each iteration i, the function $b + \sum_{j=1}^{i} \hat{g}_j(\boldsymbol{x})$ (define $\sum_{j=1}^{0} \hat{g}_j(\boldsymbol{x}) = 0$). On the right it shows that at the 10-th iteration, the model sets $\hat{g}_{10}$ to memorize the 10-th point correctly. Alg. 1 and g-unit architecture are designed so that when the 10-th point is memorized, the neural network value for the previously memorized points does not change.

Proving the Bound. We provide proof sketch for Theorem 3.4 (a), using lemmas formally stated and proven in Sec. A of our technical report [45]. Proof for Theorem 3.4 (b) is similar. Lemma A.1

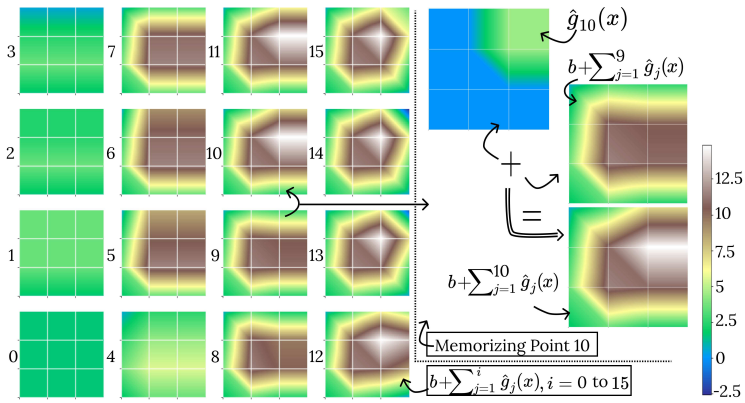

Fig. 3. Neural Network Construction Steps

Algorithm 1 Neural Network Construction

Input: A function f, a parameter t
Output: Neural network $\hat{f}$

1: $b \leftarrow f(0)$
2: **for** $i \leftarrow 1$ **to** $(t+1)^d - 1$ **do**
3: $\quad \hat{y} \leftarrow b + \sum_{j=1}^{i-1} \hat{g}_j(\frac{\boldsymbol{\pi}^i}{t})$
4: $\quad$ **for** $r \leftarrow 1$ **to** d **do**
5: $\quad\quad b_{r,i} \leftarrow \frac{\pi_r^i}{t}$
6: $\quad a_i \leftarrow t(f(\frac{\boldsymbol{\pi}^i}{t}) - \hat{y})$
7: **return** $\hat{f}$

states that $\hat{f}(\mathbf{x})$ achieves zero error at cell vertices, i.e.,

$$f(\mathbf{x}) = \hat{f}(\mathbf{x}), \forall \mathbf{x} \in P. \tag{2}$$

Furthermore, f is ρ-Lipschitz so its change is bounded within each cell. That is, for $\mathbf{x}, \mathbf{x}' \in C^i$, where $C^i = \{\frac{\boldsymbol{\pi}^i}{t} + \mathbf{z}, \mathbf{z} \in [0, \frac{1}{t}]^d\}$ is the subset of input space in the i-th cell, the Lipschitz property implies

$$|f(\mathbf{x}) - f(\mathbf{x}')| \leq \frac{\rho d}{t}. \tag{3}$$

Lemma A.2 proves that $\hat{f}$ change is bounded within each cell, i.e.

$$|\hat{f}(\mathbf{x}) - \hat{f}(\mathbf{x}')| \leq \phi(d, \rho, t, \mathbf{x}, \mathbf{x}') \tag{4}$$

for some function ϕ specified in Lemma A.2. ϕ depends on $\mathbf{x}$ and $\mathbf{x}'$ since the bound is different depending on where in space $\mathbf{x}$ and $\mathbf{x}'$ are. Using triangle inequality with Eq. 3 and 4, we have

$$|\hat{f}(\mathbf{x}) - f(\mathbf{x}) - (\hat{f}(\mathbf{x}') - f(\mathbf{x}'))| \leq \frac{d\rho}{t} + \phi(d, \rho, t, \mathbf{x}, \mathbf{x}'). \tag{5}$$

Letting $\mathbf{x}' = \frac{\boldsymbol{\pi}^i}{t}$ in Eq. 5 and using Eq. 2, we obtain

$$|\hat{f}(\mathbf{x}) - f(\mathbf{x})| \leq \frac{d\rho}{t} + \phi(d, \rho, t, \mathbf{x}, \frac{\boldsymbol{\pi}^i}{t}). \tag{6}$$

Lemma A.3 shows that integrating right hand side of Eq. 6 over $\mathbf{x}$ and across cells yields $\frac{3\rho d}{t}$ so we bound the 1-norm error as

$$\|\hat{f} - f\|_1 \leq \frac{3\rho d}{t}. \tag{7}$$

Lemma A.4 shows that space and time complexity of $\hat{f}$ is $\tilde{O}(kd)$. Setting $\varepsilon_1 = \frac{3\rho d}{t}$ and $\varkappa = 3$, recalling that $k = (t+1)^d$, and substituting $k = (\varkappa \rho d \varepsilon_1^{-1} + 1)^d$ in the space/time complextiy experssion

proves Theorem 3.4 (a). Lemma proofs require a detailed study of neural network behaviour, see Sec. A of technical report [45] .

3.2.3 Bounding Sampling Error. We present the following theorem that bounds the sampling error with high probability.

Theorem 3.5. *Let f_χ^C and f_χ^S be distribution query functions for* COUNT *and* SUM *aggregation functions and f_D^C and f_D^S the corresponding observed query functions for a database, D, of n points in d dimensions sampled from χ. For $i \in \{S, C\}$,*

$$\mathbb{P}_{D\sim\chi}\left[\frac{1}{n}\|f_\chi^i - f_D^i\|_\infty > \varepsilon_2\right] \leq \varkappa^{d+1} d\varepsilon_2^{-d} \exp\left(-\varkappa^{-1}\varepsilon_2^2 n\right),$$

Where $\varkappa$ is a universal constant.

Theorem 3.5 provides a high probability bound on Δ_s in Eq. 1. The proof of Theorem 3.5 uses VC sampling theory, which presents a novel use of VC theory for the database literature. VC theory helps us understand the impact of the distribution a database follows on operations performed (e.g., answering RAQs) on the database. In fact, Theorem 3.5 is independent of our use of learned models, and simply characterizes impact of sampling when answering RAQs on a database that follows a certain data distribution. This is different from the typical use of VC theory in machine learning, where the goal is to study generalization of a trained model to unseen testing data. We present a proof sketch for the case of COUNT. Proof for SUM is similar, but uses a generalization of VC-dimension.

Proof Sketch of Theorem 3.5 for COUNT. We start by rewriting the query function. Define the indicator function h as

$$h_{\mathbf{q}}^C(\mathbf{p}) = \begin{cases} 1 & \text{if } \forall i, c_i \leq p_i < c_i + r_i \\ 0 & \text{otherwise.} \end{cases}$$

So $f_D(\mathbf{q}) = \sum_{\mathbf{p}\in D} h_{\mathbf{q}}^C(\mathbf{p})$ and $f_\chi(\mathbf{q}) = nE_{\mathbf{p}\sim\chi}[h_{\mathbf{q}}(\mathbf{p})]$. Let $\mathcal{H}^C = \{h_{\mathbf{q}}^C, \forall \mathbf{q}\}$, so to bound error $\sup_{\mathbf{q}} \frac{1}{n}|f_D(\mathbf{q}) - f_\chi(\mathbf{q})|$, we bound

$$\sup_{h\in\mathcal{H}^C} |\frac{1}{n}\sum_{\mathbf{p}\in D} h(\mathbf{p}) - \mathbb{E}_{\mathbf{p}\sim\chi}[h(\mathbf{p})]|. \tag{8}$$

VC-dimension of $\mathcal{H}^C$ is known to be $2d$ [32] (see Lemma A.12 of technical report [45]), so applying VC theory bounds [9] (stated in Theorem A.11 of technical report [45]) to Eq. 8 proves the theorem. □

3.2.4 Completing the Proof. Let ε_1 and ε_2 be the two error parameter, and let $\hat{f}$ be the neural network in Theorem 3.4 that achieves error ε_1. Furthermore, let E_1 be the event $\frac{1}{n}\|f_\chi^i - f_D^i\|_\infty \leq \varepsilon_2$ holds for a random D sampled from χ. Observe that if E_1 holds, by triangle inequality, the event E_2 defined as $\frac{1}{n}\|\hat{f} - f_D^i\|_1 \leq \varepsilon_2 + \varepsilon_1$ also holds. Thus, $\mathbb{P}[E_1] \leq \mathbb{P}[E_2]$. Taking the complement of both event, and observing that probability of complement of E_1 is bounded by Theorem 3.5 yields Theorem 3.1.

3.3 Other Query Functions and Model Choices

Proof of DQD bound for SUM and COUNT aggregation functions decomposes the error into approximation error and sampling error. Theorem 3.4, which bounds the approximation error, is independent of the aggregation function used and applies to any function. To utilize the theoretical framework for other query functions, we need to bound the corresponding sampling error (Theorem 3.5 is specific to SUM and COUNT). In Sec. 3.3.1, we discuss this for AVG aggregation function and provide a general discussion for other query functions in Sec. 3.3.2. In Sec. 3.3.3 we discuss the applicability of our analysis framework to other modeling choices.

3.3.1 AVG *Aggregation Function.* Our study of AVG aggregation function is a variation of that of SUM and COUNT. We discuss the differences, then present our sampling error bound.

First, we consider a variation of distribution query function, defined as $\bar{f}_\chi^A(\mathbf{q}) = \frac{f_\chi^S(\mathbf{q})}{f_\chi^C(\mathbf{q})}$. which we found to be easier to theoretically study ($\bar{f}_\chi^A$ is not the expected answer to AVG query, but expected answer to SUM query divided by expected answer to COUNT query). Since it depends on data distribution, it still allows us to study impact of data distribution on query answering. Second, we define LDQ as the Lipschitz constant of $\bar{f}_\chi^A$. LDQ in this case is not normalized by data size (as it was for SUM and COUNT in Sec. 3.1.1), since magnitude of query answers for AVG do no change as data size changes. Third, for small query ranges few points in the database may match the query, even if data size is large. In such cases, for AVG aggregation function, the observed query function will be a poor estimate of the distribution query function. For COUNT or SUM query functions, few data points in a range means that both SUM and COUNT values are small, but this is not the case for the AVG function whose distribution query answer is independent of the number of points sampled in the range. To capture this dependence on query range, we define $\mathcal{Q}_\xi = \{\mathbf{q}, s.t., f_\chi^C(\mathbf{q}) \geq \xi\}$. Our bound depends on ξ, which captures the probability of observing a point in a range.

Lemma 3.6. *Recall that* $f_D^A(\mathbf{q}) = \frac{f_D^S(\mathbf{q})}{f_D^C(\mathbf{q})}$ *is the* AVG *query function. Let* $\text{err}(\mathbf{q}) = \frac{|\bar{f}_\chi^A(\mathbf{q}) - f_D^A(\mathbf{q})|}{|\bar{f}_\chi^A(\mathbf{q})|+1}$. *We have*

$$\mathop{\mathbb{P}}_{D\sim\chi}\left[\sup_{\mathbf{q}\in\mathcal{Q}_\xi} \text{err}(\mathbf{q}) \geq \varepsilon\right] \leq \varkappa^{d+1} d \left(\frac{1+\varepsilon}{\xi\varepsilon}\right)^d \exp\left(-\varkappa^{-1}(\frac{\xi\varepsilon}{1+\varepsilon})^2 n\right),$$

Where $\varkappa$ *is a universal constant.*

Proof Sketch. Proof applies Theorem 3.5 to numerator and denominator of AVG query function (Sec. A.4 of technical report [45]). □

Combining Lemma 3.6 and Theorem 3.4 show similar discussions to Sec. 3.1.2 on dependence on data distribution and size also apply to AVG queries. Lemma 3.6 also shows impact of query range.

More Accurate on Larger Ranges. Impact of query range is modeled through the parameter ξ. Larger ξ means the bound applies to larger ranges, where the confidence in the bound increases with ξ. Fixing the confidence level, observe that ξ and ε are negatively correlated. Increasing the query ranges considered reduces the sampling error. Thus, if LDQ of the query function is small (approximation error is low) and query range is large (sampling error is low), a neural network can answer AVG RAQs accurately and efficiently. LDQ can be calculated similar to examples in Sec. 3.1.2.

3.3.2 Other Query Functions. Bounding sampling error for queries with COUNT, SUM or AVG aggregation functions but different range predicates (e.g., circular predicate $(\mathbf{c}, \mathbf{r})$ matching points $\mathbf{p}$, $\|\mathbf{p} - \mathbf{c}\|_2 \leq \mathbf{r}$) can be done similar to proof of Theorem 3.5 (only finding range predicate's VC-dimension needs further study). However, applicability of VC theory depends on the aggregation function.

3.3.3 DQD for Query Modelling Approaches. Our analysis framework allows for providing DQD bounds for other *query modeling* approaches, where we define query modelling as an approach that directly models the query answers. Furthermore, our analysis of sampling error (Theorem 3.5, Lemma 3.6) does not depend on modeling choices and is generic to query modeling approaches. Thus, insights about the role of data size can be applicable to other query modeling approaches. For instance, consider answering count queries on uniformly distributed data in range [0, 1], as in Example 3.2. For data size n, as data size increases, the number of data points in a query (c_1, r_1) becomes more similar to $r_1 \times n$, which is the expected number of points that fall in any range of length r_1. Thus, one can estimate the answer to count query with a model $\hat{g}$ defined as

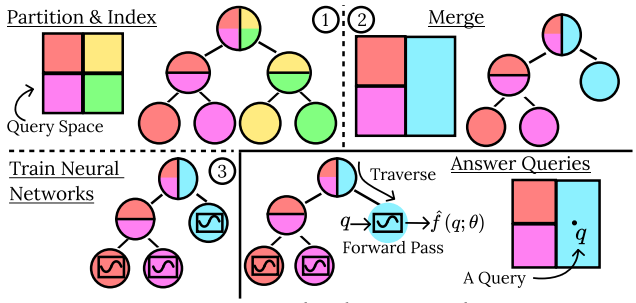

Fig. 4. NeuroSketch Framework

$\hat{g}(c_1, r_1) = n \times r_1$. Answering queries with $\hat{g}$ takes constant time (it's a single operation), and its accuracy improves as data size increases, as supported by Theorem 3.5.

4 NEUROSKETCH

DQD bound formalizes how complexity of answering RAQs relates to data and query properties. In this section, we present a novel *complexity-aware* neural network framework, NeuroSketch, that utilizes results from DQD bound to allocate model capacity. We first present an overview of NeuroSketch, then discuss its details and finally discuss how it can be used in real-world database systems together with our DQD bound.

4.1 NeuroSketch Overview

The key idea behind NeuroSketch design is that, even on the same database, some queries can be more difficult to answer than others (e.g., larger ranges vs. smaller ranges, see Sec. 3.3.1). By allocating more model capacity to queries that are more difficult, we can improve the performance. We do so by partitioning the query space and training independent neural networks for each partition. The partitioning allows diverting model capacity to harder queries, which our DQD bound allows us to quantify. By creating models specialized for a specific part of the query space, *query specialization* allows us to control how model capacity is used across query space.

Fig. 4 shows an overview of NeuroSketch. During a pre-precessing step, (1) we partition and index the query space using a kd-tree. The partitioning is done based on our query specialization principle, with the goal of training a specialized neural network for different parts of the query space. (2) To account for the complexity of the underlying function in our partitioning, we merge the nodes of the kd-tree that are *easier* to answer based on our DQD bound, so that our model only has to specialize for the certain parts of the space that are estimated to be more difficult. (3) After some nodes of the kd-tree have been merged, we train a neural network for all the remaining leaves of the kd-tree. Finally, to answer queries at query time, we traverse the kd-tree to find the leaf node a query falls inside, and perform a forward pass of the neural network.

4.2 NeuroSketch Details

Training NeuroSketch uses a training query set $Q \subseteq \mathcal{Q}$. Q can be sampled from $\mathcal{Q}$ according to a workload distribution, or can be a uniform sample in the absence of any workload information. We do not assume access to workload information, but our framework can take advantage of the query workload if available.

Partitioning & Indexing. To partition the space, we choose partitions that are smaller where the queries are more frequent and larger where they are less frequent. This allows us to divert more model capacity to more frequent queries, thereby boosting their accuracy if workload information is available. We achieve this by partitioning the space such that all partitions are equally probable. To do so, we build a kd-tree on our query set, Q, where the split points in the kd-tree can be considered

Algorithm 2 $partition\_\&\_index(N, h, i)$

Input: A kd-tree node N, tree height h and dimension, i to split the node, N on
Output: A kd-tree with height h rooted at N

1: **if** $h = 0$ **then**
2: **return**
3: $N.val \leftarrow$ median of $N.Q$ along i-th dimension
4: $N.dim \leftarrow i$
5: $Q_{left} = \{\mathbf{q} | \mathbf{q} \in N.Q, q[N.dim] \leq N.val\}$
6: $Q_{right} = \{\mathbf{q} | \mathbf{q} \in N.Q, q[N.dim] > N.val\}$
7: **for** $x \in \{left, right\}$ **do**
8: $N_x \leftarrow$ new node
9: $N_x.Q \leftarrow Q_x$
10: $N.x \leftarrow N_x$ ▷*Adding N_x as left or right child of N*
11: $get\_index(N_x, h-1, (N.dim+1) \mod d)$

Algorithm 3 $merge(N, s)$

Input: kd-tree root node N and desired number of partitions s
Output: kd-tree with s leaf nodes

1: **repeat**
2: **for all** Leaf nodes N **do**
3: $AQC_N \leftarrow \frac{1}{\binom{|N.Q|}{2}} \sum_{\mathbf{q},\mathbf{q}' \in N.Q, \mathbf{q} \neq \mathbf{q}'} \frac{|f_D(\mathbf{q}) - f_D(\mathbf{q}')|}{\|\mathbf{q} - \mathbf{q}'\|}$
4: $N \leftarrow$ the leaf node with smallest ACQ_N
5: $N.marked \leftarrow true$
6: **for all** Sibling leaf nodes N_1, N_2 **do**
7: **if** $N_1.marked = N_2.marked = true$ **then**
8: Merge N_1 and N_2
9: **until** There are s leaf nodes

as estimates of the median of the workload distribution (conditioned on the current path from the root) along one of its dimensions. We build the kd-tree by specifying a maximum height, h, and splitting every node until all leaf nodes have height h, which creates 2^h partitions. Splitting of a node N is done based on median of one of the dimensions of the subset, $N.Q$, of the queries, Q, that fall in N. Alg. 2 shows this procedure. To build an index with height h rooted at a node, N_{root} (note that $N_{root}.Q = Q$), we call $partition\_\&\_index(N_{root}, h, 0)$. We note that other partitioning methods (e.g., clustering the queries to perform partitioning) are also possible, but we observed kd-tree to be a simple practical solution with little overhead that performed well.

Merging. We merge some of kd-tree leaves using DQD bound. As discussed in Sec. 3.1.4, LDQ can be difficult to measure in practice, so we use AQC as a proxy, as shown in Alg. 3. At each iteration, we first measure the approximation complexity for the leaf nodes, in line 3, where the approximation complexity, AQC_N for a leaf node N is calculated based on queries that fall in the node N. Then, we mark the node with the smallest AQC_N for merging. When two sibling leaf nodes are marked, they are merged together, as shown in line 8. The process continues until the number of remaining leaf nodes reaches the desired threshold. In practice, we observed that the quantity AQC_N is correlated with the error of the neural networks, which empirically justifies this design choice (see Sec. 5.5).

Training Neural Networks. We train an independent model for each of the remaining leaf nodes after merging. For a leaf node, N, the training process is a typcial supervised learning

Algorithm 4 Model Training

Input: A dataset D, a kd-tree node N
Output: Neural network $\hat{f}$ for node N
1: Initialize the parameters, θ, of a neural network $\hat{f}(.;\theta)$
2: **repeat**
3: Sample, Q_{batch}, a subset of $N.Q$
4: Update θ in direction $-\nabla_\theta \sum_{\mathbf{q}\in Q_{batch}} \frac{(\hat{f}(\mathbf{q};\theta)-f_D(\mathbf{q}))^2}{|Q_{batch}|}$
5: **until** convergence
6: **return** $\hat{f}$

Algorithm 5 *answer_query*$(N, \mathbf{q})$

Input: kd-tree root node N and query $\mathbf{q}$
Output: Answer to $\mathbf{q}$
while N is not leaf **do**
  if $q[N.dim] \leq N.val$ **then**
    $N \leftarrow N.left$
  else
    $N \leftarrow N.right$
return $N.model.forward\_pass(\mathbf{q})$

procedure and shown in Alg. 4 for completeness. The answer to queries for training, used in line 4 of Alg. 4, can be collected through any known algorithm, where a typical algorithm iterates over the points in the database, pruned by an index, and for a candidate data point checks whether it matches the RAQ predicate or not. This is a pre-processing step and is only performed once to train our model. The process is embarrassingly parallelizable across training queries, if preprocessing time is a concern. Furthremore, if the data is disk resident, we keep partial SUM/COUNT answers for each training query while scanning data from disk, so a single scan of data is sufficient (similar to building disk-based indexes) to collect training query answers. Once trained, NeuroSketch is much smaller than data and expected to fit in memory, so it will be much faster than disk-based solutions. We use Adam optimizer [19] for training and train a fully connected neural network for each of the partitions. The architecture is the same for all the partitions and consists of n_l layers, where the input layer has dimensionality d, the first layer consists of l_{first} units, the next layers have l_{rest} units and the last layer has 1 unit. We use relu activation for all layers (except the output layer). n_l, l_{first} and l_{rest} are hyper-parameters of our model. Although approaches in neural architecture search [47] can be applied to find them, they are computationally expensive. Instead, we do a grid search to find the hyper-parameters so that NeuroSketch satisfies the space and time constraints in Problem 1 while maximizing its accuracy.

Answering Queries. As shown in Alg. 5, to answer a query, $\mathbf{q}$, first, the kd-tree is traversed to find the leaf node that the query $\mathbf{q}$ falls into. The answer to the query is a forward pass of the neural network corresponding to the leaf node.

4.3 General RAQs and Real-World Application

General RAQs. NeuroSketch can be used for more general RAQs than defined in Sec. 2. An RAQ consists of a range predicate, and an aggregation function AGG. In NeuroSketch, we make no assumption on the aggregation function AGG and our empirical results evaluated NeuroSketch on SUM, AVG, COUNT, MEDIAN and STD. We consider range predicates that can be represented by a *query instance* $\mathbf{q}$, and a binary *predicate function*, $P_f(\mathbf{q}, \mathbf{x})$, that takes as inputs a point in the database, $\mathbf{x}$, $\mathbf{x} \in D$, and the query instance $\mathbf{q}$, and outputs whether $\mathbf{x}$ matches the predicate or not. Then, given a predicate function and an aggregation function, range aggregate queries can be represented by

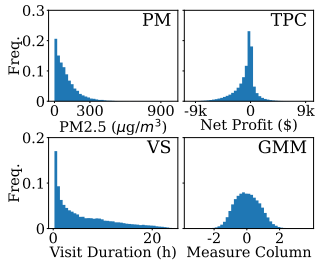

Fig. 5. Measure column distribution (shared y-axis)

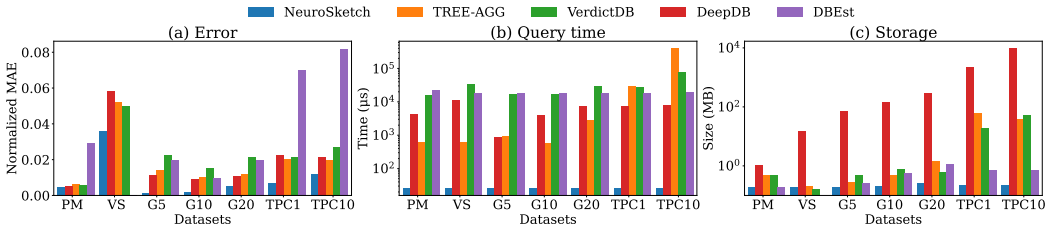

Fig. 6. RAQs on different datasets

Dataset	G5, G10, G20	PM [21]	TPC1, TPC10 [25]	VS
# Points	10^5	4.17×10^4	2.65×10^6, 2.65×10^7	10^5
Dim	5, 10, 20	4	13	3

Table 1. Dataset information

the query function $f_D(\mathbf{q}) = \texttt{AGG}(\{\mathbf{x} : \mathbf{x} \in D, P_f(\mathbf{x}, \mathbf{q}) = 1\})$. We avoid specifying how the predicate function should be defined to keep our discussion generic to arbitrary predicate functions, but some examples follow. To represent the RAQs of the form discussed in Sec. 2, $\mathbf{q}$ can be defined as lower and upper bounds on the attributes and $P_f(\mathbf{q}, \mathbf{x})$ defined as the WHERE clause in Sec. 2. We can also have $P_f(\mathbf{q}, \mathbf{x}) = x[1] > x[0] \times q[0] + q[1]$, so that $P_f(\mathbf{q}, \mathbf{x})$ and $\mathbf{q}$ define a half-space above a line specified by $\mathbf{q}$. For many applications, WHERE clauses in SQL queries are written in a parametric form [3–5] (e.g., WHERE $X_1 > ?param1$ OR $X_2 > ?param2$, where $?param$ is the common SQL syntax for parameters in a query). Such queries can be represented as query functions by setting $\mathbf{q}$ to be the parameters of the WHERE clause.

NeuroSketch and DQD in Practice. Possible RAQs correspond to various query function and NeuroSketch learns different models for different query functions. This follows the *query specialization* design principle, where a specialized model is learned to answer a query function well. A query processing engine can be used to decide which query functions to use NeuroSketch for. This can happen both on the fly, when answering queries, and during database maintenance. During maintenance, DQD bound can be used to decide which queries to build NeuroSketch for (e.g., for queries with small LDQs). Moreover, after NeuroSketch is built for a query function, DQD can be used to decide whether to use NeuroSketch for a specific query instance or not on the fly. For instance, queries with large ranges (that NeuroSketch answers accurately according to DQD) can be answered by NeuroSketch, while queries with smaller ranges can be asked directly from the database.

5 EMPIRICAL STUDY

5.1 Experimental Setup

System Setup. Experiments are performed on a machine with Ubuntu 18.04 LTS, an Intel i9-9980XE CPU (3GHz), 128GB RAM and a GeForce RTX 2080 Ti NVIDIA GPU.

Datasets. Table 1 shows the datasets used in our experiments, with details discussed below. Fig. 5 shows the histogram of measure column values used in the experiments.

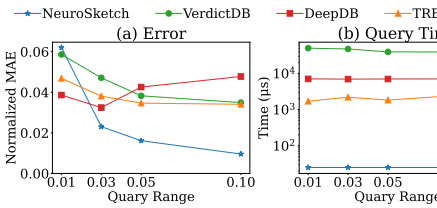

Fig. 7. Varying query range

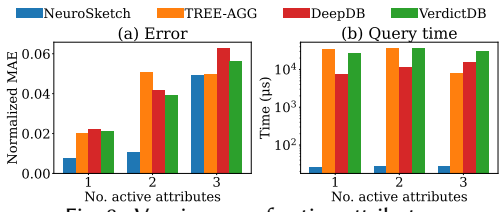

Fig. 8. Varying no. of active attributes

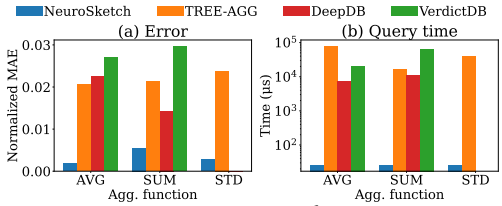

Fig. 9. Varying agg. function

PM. PM [21] contains Fine Particulate Matter (PM2.5) measuring air pollution and other statistics (e.g., temperature) for locations in Beijing. Similar to [23], PM2.5 is the measure attribute.

TPC-DS. We used TPC-DS [25], a synthetic benchmark dataset, with scale factors 1 and 10, respectively referred to as TPC1 and TPC10. Since we study RAQs, we use the numerical attributes in store_sales table as our dataset, and net_profit as measure attribute.

Veraset. As was used in our running example, we use Veraset dataset, which contains anonymized location signals of cell-phones across the US collected by Veraset [2], a data-as-a-service company. Each location signal contains an anonymized id, timestamp and the latitude and longitude of the location. We performed stay point detection [42] on this dataset (to, e.g., remove location signals when a person is driving), and extracted location visits where a user spent at least 15 minutes and for each visit, also recorded its duration. 100,000 of the extracted location visits in downtown Houston were sampled to form the dataset used in our experiments, which contains three columns: latitude, longitude and visit duration. We let visit duration to be the measure attribute.

GMMs. We study data dimensionality with synthetic 5, 10 and 20 dimensional data from Gaussian mixture models (GMM) (100 components, random mean and co-variance), referred to as G5, G10 and G20. GMMs are often used to model real data distribution [30].

Query Distribution. Our experiments consider query functions consisting of `AVG`, `SUM`, `STDEV` (standard deviation) and `MEDIAN` aggregation functions together with two different predicate functions. First, similar to [23], our experiments show the performance on the predicate function defined by the WHERE clause in Sec. 2. We consider up to 3 active attributes in the predicate function. To generate a query instance with r active attributes, we first select, uniformly at random, r activate attributes (from a total of d possible attributes). Then, for the selected active attributes, we randomly generate a range. Unless otherwise stated, the range for each active attribute is uniformly distributed. This can be thought of as a more difficult scenario for NeuroSketch as it requires approximating the query function equally well over all its domain, while also giving a relative advantage to other baselines, since they are unable to utilize the query distribution. Unless otherwise stated, for all datasets except Veraset, we report the results for one active attributes and use `AVG` aggregation function. For Veraset, we report the results setting latitude and longitude as active attributes. Second, to show how NeuroSketch can be applied to application specific RAQs, in Sec. 5.2.2, we discuss answering the query of median visit duration given a *general* rectangle on Veraset dataset.

Measurements. In addition to query time and space used, we report the normalized absolute error for a query in the set of test queries, T, defined as $\frac{|f_D(\mathbf{q}) - \hat{f}_D(\mathbf{q},\theta)|}{\frac{1}{|T|}\sum_{\mathbf{q}\in T}|f_D(\mathbf{q})|}$. We ensure that none of the test queries are in the training set. The error is normalized by average query result magnitude to allow for comparison over different data sizes and datasets when the results follow different scales.

Learned Baselines. We use DBEst [23] and DeepDB [15] as the state-of-the-art model-based AQP engines. Both algorithms learn data models to answer RAQs. We use the open-source implementation of DBEst available at [24] and DeepDB at [16]. For DBEst, we perform a gird search on its MDN architecture (number of layers, layer width, number of Gaussian componenets) and optimize it per dataset. For DeepDB we optimize its RDC threshold for each dataset. We do not use [35] as a baseline, which samples new data points at query time from a learned model to answer queries because the results in [35] show worse accuracy and same query time ([35] improves storage) compared with sampling directly from the data (which we have included as baseline). We also modified NeuroCard [37], a learned cardinality estimation method to answer RAQs, but we observed the modified approach to perform worse than DeepDB on RAQs. We do not present the results for [37], since it is not designed for RAQs and performed worse than DeepDB.

Sampling-based Baselines. We use VerdictDB [26] as our sampling-based baseline, using its publicly available implementation [27]. We also implemented a sampling-based baseline designed specifically for range aggregate queries, referred to as TREE-AGG. In a pre-processing step and for a parameter k, TREE-AGG samples k data points from the database uniformly. Then, for performance enhancement and easy pruning, it builds an R-tree index on the samples, which is well-suited for range predicates. At query time, by using the R-tree, finding data points matching the query is done efficiently, and most of the query time is spent on iterating over the points matching the predicate to compute the aggregate attribute required. For both TREE-AGG and VerdictDB, we set the number of samples so that the error is similar to that of DeepDB.

NeuroSketch Training and Evaluation. NeuroSketch training is performed in Python 3.7 and Tensorflow 2.1, with implementation publicly available at [46]. Model training is done on GPU. Models are saved after training. For evaluation, a separate program written in C++ and running on CPU loads the saved model, and for each query performs a forward pass on the model. Model evaluation is done with C++ and on CPU, without any parallelism for any of the algorithms. Unless otherwise stated, model depth is set to 5 layers, with the first layer consisting of 60 units and the rest of 30 units. The height of the kd-tree is set to 4, and parameter $s = 8$ so that the kd-tree has 8 leaf nodes after merging.

5.2 Baseline Comparisons

5.2.1 Results Across Datasets. Fig. 6 (a) shows the error on different datasets, where NeuroSketch provides a lower error rate than the baselines. Fig. 6 (b) shows that NeuroSketch achieves this while providing multiple orders of magnitude improvement in query time. NeuroSketch has a relatively constant query time because, across all datasets, NeuroSketch's architecture only differs in its input dimensionality, which only impacts number of parameters in the first layer of the model and thus changes model size by very little. Due to our use of small neural networks, we observe that model inference time for NeuroSketch is very small and in the order of few microseconds, while DeepDB and DBEst answers queries multiple orders of magnitude slower. DBEst does not support multiple active attributes and thus its performance is not reported for VS. The results on G5 to G20 show the impact of data dimensionality on the performance of the algorithms. As was suggested by our theoretical results, for NeuroSketch, the error increases as dimensionality increases. A similar impact can be seen for DeepDB, manifesting itself in increased query time. Furthermore, the R-tree index of TREE-AGG often allows it to perform better than the other baselines, especially for low dimensional data. Finally, Fig. 6 (c) shows the storage overhead of each methods. NeuroSketch

Metric	NeuroSketch	TREE-AGG	DeepDB & VerdictDB
Norm. MAE	0.045	0.052	N/A
Query time (μs)	25	601	N/A

Table 2. Median visit duration for general rectangles

answers queries accurately by taking less than one MB space, while DeepDB's storage overhead increases with data size, to more than one GB.

5.2.2 Results Across Different Workloads. We use TPC1 and VS to study impact of query workload on performance of the algorithms. Unless otherwise stated results are on TPC1. Due to its poor performance on TPC1 and not supporting multiple active attributes (for VS queries), we exclude DBEst from the experiments here.

Impact of Query Range. We set the query range to x percent of the domain range, for $x \in \{1, 3, 5, 10\}$ and present the results in Fig. 7. The error of NeuroSketch increases for smaller query ranges, as our theoretical results suggest. As mentioned before, this is because for smaller ranges NeuroSketch needs to memorize where exactly each data point is, rather than learning the overall distribution of data points. Nevertheless, NeuroSketch provides better accuracy than the baselines for query ranges at least 3 percent, and performs queries orders of magnitude faster for all ranges. If more accurate answers are needed for smaller ranges, increasing the model size of NeuroSketch can improve its accuracy at the expense of query time (see Sec. 5.3).

Impact of No. of Active Attributes. In Fig. 8, we vary the number of active attributes in the range predicate from one to three. Accuracy of all the algorithms drops when there are more active attributes, with NeuroSketch outperforming the algorithms both in accuracy and query time. Having more active attributes is similar to having smaller ranges, since fewer points will match the query predicate. Thus, our theoretical results explain the drop in accuracy.

Impact of Aggregation Function. Fig. 9 shows how different aggregation functions impact performance of the algorithms. NeuroSketch is able to outperform the algorithms for all aggregation functions. VerdictDB and DeepDB implementation did not support `STDEV` and no result is reported for `STDEV` for these methods.

Median Visit Duration Query Function. We consider the query of median visit duration given a *general* rectangular range. The predicate function takes as input coordinates of two points $\mathbf{p}$ and $\mathbf{p'}$, representing the location of two non-adjacent vertices of the rectangle, and an angle, ϕ, that defines the angle the rectangle makes with the x-axis. Given $\mathbf{q} = (\mathbf{p}, \mathbf{p'}, \phi)$, the query function returns median of visit duration of records falling in the rectangle defined by $\mathbf{q}$. This is a common query for real-world location data, and data aggregators such as SafeGraph [1] publish such information.

Table 2 shows the results for this query function. Neither DeepDB nor DBEst can answer this query. The predicate function is not supported by those methods, and extending those methods to support them is not trivial. On the other hand, NeuroSketch can answer this query function, with similar performance to other queries on VS dataset. Although VerdictDB can be extended to support this query function, the current implementation does not support the aggregation function, so we do not report the results on VerdictDB.

5.3 Model Architecture Analysis

5.3.1 Time/Space/Accuracy Trade-Offs of Model Architectures

Setup. We study different time /space/accuracy trade-offs achievable by NeuroSketch and other methods in Fig. 10 based on different system parameters. For NeuroSketch, we vary number of layers (referred to as depth of the neural network), d, number of units per layer (referred to as width of the neural network), w, and height of the kd-tree, h, to see their impact on its time/space/accuracy (we avoid merging kd-tree nodes here, and study the impact of merging separately in Sec. 5.5). Fig. 10 shows several possible combinations of the hyperparameters. For each line in Fig. 10, NeuroSketch

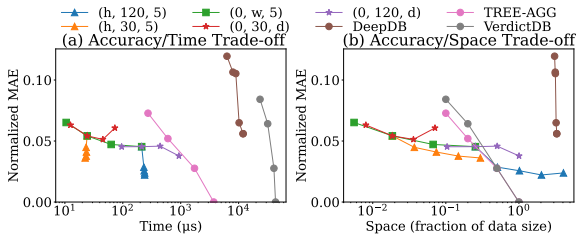

Fig. 10. Time/Space/Accuracy Trade-Off with Different Model Architectures

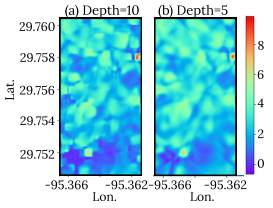

Fig. 11. Learned NeuroSketch Visualization

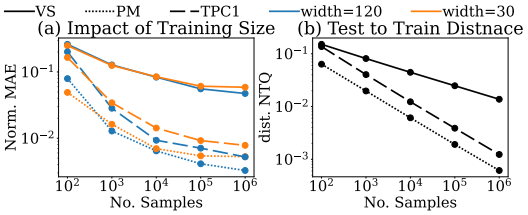

Fig. 12. Generalization Study

is run with two of the hyperparameters kept constant and one changing. The line labels are of the form $(height, width, depth)$, where two of *height, width* or *depth* have numerical values and are the constant hyperparameters for that particular line. Furthermore, the value of one of *height, width* or *depth* is $\{d, w, h\}$ and is the variable hyperparameter for the plotted line. For example, line labelled (h, 120, 5) means the experiments for the corresponding line are with a NeuroSketch architecture with 120 number of units per layer, 5 layers and each point plotted corresponds to a different value for the kd-tree height, and label (0, 30, d) means the experiments are run with varying depth of the neural network, with kd-tree height 0 (i.e. only one partition) and the neural width network is 30. The hyperparameter values are as follows. For lines (h, 120, 5) and (h, 30, 50), kd-tree height is varied from 0 to 4, for the line labelled (0, w, 5) neural network width is $\{15, 30, 60, 120\}$ and for lines (0, 120, d) and (0, 30, d) neural network depth is $\{2, 5, 10, 20\}$.

TREE-AGG and VerdictDB are plotted for sampling sizes of 100%, 50%, 20% and 10% of data size. For DeepDB, we report results for RDC thresholds in [0.1, 1] (minimum error is at RDC threshold=0.3. Error increases for values less than 0.1 or more than 1).

Results. Fig. 10 (a) shows the trade-off between query time and accuracy. NeuroSketch performs well when fast answers are required but some accuracy can be sacrificed, while if accuracy close to an exact answer is required, TREE-AGG can perform better. Furthermore, Fig. 10 (b) shows the trade-off between space consumption and accuracy. Similar to time/accuracy trade-offs, we observe that when the error requirement is not too stringent, NeuroSketch can answer queries by taking a very small fraction of data size. Finally, NeuroSketch outperforms DeepDB in all the metrics. Furthermore, comparing TREE-AGG with VerdictDB shows that, on this particular dataset, the sampling strategy of VerdictDB does not improve upon uniform sampling of TREE-AGG while the R-tree index of TREE-AGG improves the query time over VerdictDB.

Moreover, Fig 10 shows the interplay between different hyperparameters of NeuroSketch. We see that increasing depth and width of the neural networks improves the accuracy, but after a certain accuracy level the improvement plateaus and accuracy even worsens if depth of the neural network is increased but the width is too small (i.e., the red line). Nevertheless, using partitioning method allows for further improving the time/accuracy trade-off as it improves the accuracy at almost no cost to query time. We also observe that kd-tree improves the space/accuracy trade-off, compared with increasing the width or depth of neural networks. This shows that our paradigm of query specialization is beneficial, as learning multiple specialized models each for a different part

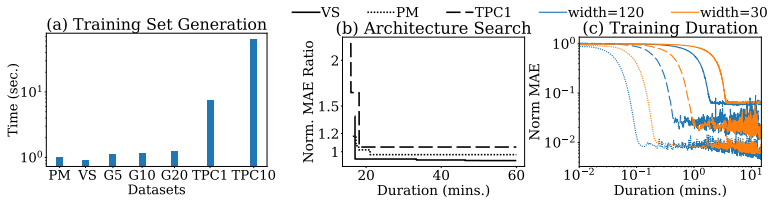

Fig. 13. Preprocessing Time Study

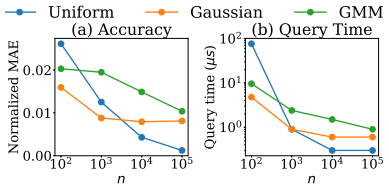

Fig. 14. DQD Bound on Synthetic Datasets

of the query space performs better than learning a single model for the entire space. We discuss these results in the context of our DQD bound in Sec. 5.7.

5.3.2 Visualizing NeuroSketch for Different Model Depth. Fig. 11 shows the function NeuroSketch has learned for our running example, for two neural networks with the same architecture, but with depths 5 and 10. Comparing Fig. 11 with Fig. 1, we observe that NeuroSketch learns a function with similar patterns as the ground truth but the sharp drops in the output are smoothened out. We also observe that the learned function becomes more similar to the ground truth as we increase the number of parameters. Note that the neural networks are of size about 9% and 3.8% of the data size.

5.4 NeuroSketch Generalization Analysis

Fig. 12 studies generalization ability of NeuroSketch from train to test queries across across datasets. The results are for a NeuroSketch with tree height 0 (i.e., no partitioning), neural network depth 5 and with neural network widths of 30 and 120. Fig. 13 (a) shows that training size of about 100,000 sampled query points is sufficient for both architectures to achieve close to their lowest error. Furthermore, when sample size is very small, smaller architecture generalizes better, while the larger neural network improves performance when enough samples are available.

In Fig. 13 (b), we plot the average Eucleadian distance from test queries to their nearest training query, refered to as dist. NTQ. To compare across datasets, datasets are scaled to be in $[0, 1]$ for this plot, and the difference in dist. NTQ values is due to different data dimensionality and number of active attributes in the queries. We ensure none of the test queries appear in the training set, but as the number of training samples increases, dist. NTQ decreases. Nonetheless, when model size is small, eventhough increasing number of samples beyond 100,000 decreaes dist. NTQ, model accuracy does not improve. This suggests that for small neural networks, the error is due to the capacity limit of the model to learn the query function, and not lack of training data.

5.5 Ablation Study of Partitioning

We study the impact of merging in the prepossessing step of NeuroSketch. Recall that we set the tree height to 4, so that the partitioning step creates 16 partitions that are merged using AQC, after which 8 partitions remain. We compare this approach with two alternatives. (1) We perform no partitioning and train a single neural network to answer any query. (2) We set the tree height to 3 so that we obtain 8 partitions without performing any merging. Table 3 shows the result of this comparison. It shows that performing partitioning, either with merging or without merging is better than no partitioning across all datasets. Second, for almost all datasets, merging provides

Dataset	Normalized AQC STD	% Improved (Merging)	% Improved (No Merging)
VS	1.02	47.6	44.1
PM	0.30	22.8	18.6
TPC1	0.17	23.5	6.7
G5	0.41	12.0	13.2
G10	0.10	6.8	6.8
G20	0.07	14.6	14.6
Correlation with STD		0.87	0.94

Table 3. Improvement of partitioning over no partitioning

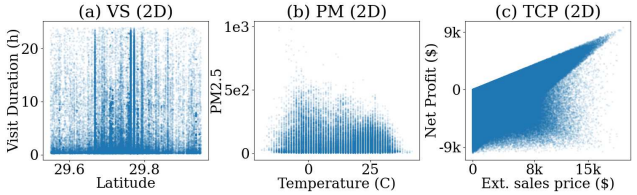

Fig. 15. 2D data subsets

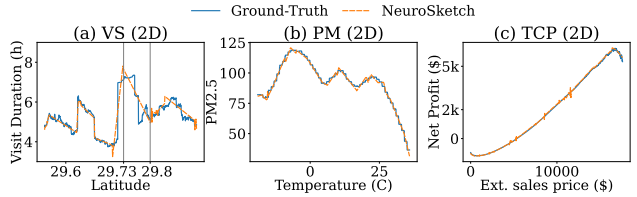

Fig. 16. Learned and True Query Functions on 2D Datasets

better or equal performance compared with no merging. Thus, in practice, using AQC as an estimate for function complexity to merge nodes is beneficial.

In fact, we observed a correlation coefficient of 0.61 between AQC and the error of trained models, which quantifies the benefits of using AQC as an estimate for function complexity. It also implies that AQC can be used to decide whether a query function is too difficult to approximate. For instance, in a database system, the query optimizer may build NeuroSketches for query functions with smaller AQC, and use a default query processing engine to answer query functions with larger AQC.

Furthermore, Table 3 shows that the benefit of partitioning is dataset dependent. We observed a strong correlation between the standard deviation of AQC estimates across leaf nodes of the kd-tree and the improvement gain from partitioning. Specifically, Let $R = \{AQC_N, \forall \text{ leaf } N\}$, as calculated in line 3 of Alg. 3. We calculate $\frac{\text{STD}(R)}{\text{AVG}(R)}$ as the normalized AQC STD for each dataset. This measurement is reported in the second column of Table 3. The last row of the table shows the correlation of the improvement for the partitioning methods with this measure. The large correlation suggests that when the difference in the complexity of approximation for different parts of the space is large, partitioning is more beneficial. This matches our intuition for using partitioning, where our intention is to allow specialized models to focus on the complex parts of the query space. It shows that partitioning is beneficial if there are parts of the space that are more complex than others.

5.6 NeuroSketch Preprocessing Time Analysis

Training Set Generation. Fig. 13 (a) shows the time it takes to generate the training set of 100,000 queries is at most 60 seconds, with most datasets taking only a few seconds. The reported results are obtained by answering the queries in parallel on GPU. The queries are answered by scanning

all the database records per query and with no indexing. We expect faster training set generation by building indexes.

Achitecture Search. Fig. 13 (b) shows the time to perform architecture search for each dataset. We use Optuna [6], a tool that uses baysian optimization to perform hyperparameter search. We use the query time and space requirement (to solve Problem 1), to limit maximum number of neural network parameters. Then, we use Optuna to find the width and depth of the neural network that minimizes error. We run Optuna for a total of one hour and set model size limit to be equal to the nueral network size in our default setting. For a point in time, t we report the ratio of error of the best model found by Optuna upto time t divided by error of our default model architecture. This ratio over time is plotted in Fig. 13 (b). The figure shows that Optuna find a model that provides accruacy within 10% of our default architecture in around 20 minutes. It also finds a better architecture for VS dataset than our default, showing that NeruoSketch accuracy can be improved by performing dataset specific parameter optimization. Optuna trains models in parallel (multiple models fit in a single GPU), and also stops training early if a setting is not promissing, so that more than 300 parameter settings are evaluated in the presented one hour for each dataset.

Training Time. Fig. 13 (c) shows the accuracy of neural networks during training. Models converge within 5 minutes of training across datasets, and error fluctuates when training for longer. Models with larger width converge faster.

5.7 Confirming DQD Bound with NeuroSketch

Model Size and DQD. We revisit Fig 10 in the context of our DQD bound. First, unsurprisingly, we observe that the overall trend of improved accuracy for larger models matches DQD. More interestingly, we further observe that Fig 10 shows increase in data size increases accuracy, but only up to a certain point, after which increasing model size has little impact. This also matches DQD, where, in Theorem 3.1, increasing size which reduces ε_1 only reduces total error (i.e., $\varepsilon_1 + \varepsilon_2$) up to when $\varepsilon_1 = 0$. After $\varepsilon_1 = 0$, error cannot be reduced further by increasing number of parameters. As discussed in Sec. 3.1.2, this is because f_D, unlike f_χ, may be a discontinuous function, so error of a neural network is not guaranteed to ever go to zero (i.e. Theorem. 3.4 doesn't apply to f_D).

Data Size, LDQ and DQD. We corroborate the observations made in the DQD bound with NeuroSketch using synthetic datasets, so that we can calculate the corresponding LDQs. We sample n points from uniform, Gaussian and two-component GMM distributions (see Sec. 3.1.3 on how to calculate their LDQs) and answer RAQs with `COUNT` aggregation function on the sampled datasets, varying the value of n. We train NeuroSketch with partitioning disabled to isolate the neural network ability to answer queries.

Fig. 14 shows the result of this experiment. In Fig. 14 (a), we fix the neural network architecture so that query time and space complexity is fixed (we use one hidden layer with 80 units) and train NeuroSketch for different data sizes and distributions. We observe that, as DQD bound suggests, the error decreases for larger data sizes. Furthermore, uniform distribution, which has a smaller LDQ, achieves the lowest error, then Gaussian whose LDQ is larger and finally GMM which has the largest LDQ. Fig. 14 (b) shows similar observations, but with accuracy fixed to 0.01 and space and time complexity allowed to change. Specifically, we perform a grid search on model width, where we train NeuroSketch for different model widths and find the smallest model width where the error is at most 0.01. We report query time of the model found with our grid search in Figs. 14 (b). As DQD bound suggests, the query time and space consumption decrease when data size increases. Moreover, the same observations hold for storage cost, where we haven't plotted the results as they look identical to that of Figs. 14 (b) (both storage cost and query time are a constant multiple of the number of parameters of the neural network, so both storage cost and query time are constant multiples of each other).

Dataset	VS (2D)	PM (2D)	TPC (2D)
Norm. MAE	0.035	0.014	0.0029
Norm. AQC	1.28	0.95	0.77

Table 4. DQD Bound on 2D Real/Benchmark Datasets

Interestingly, for small data sizes, the difficulty of answering queries across distributions does not follow their LDQ order, where uniform distribution is harder when $n = 100$ compared with a Gaussian distribution. When data size is small, a neural network has to memorize the location of all the data points, which can be more difficult with uniform distribution as the observed points may not follow any recognizable pattern. Nonetheless, as data size increases, as suggested by DQD bound, the error, query time and space complexity improve, and the difficulty of answering queries from different distributions depends on the LDQ.

DQD and Real/Benchmark Distributions. To further investigate impact of data distribution on accuracy, we visualize 2D subsets of PM, VS and TPC1. We perform RAQs that ask for AVG of the measure attribute where predicate column falls between c and $c + r$, where r is fixed to 10% of column range and c is the query variable (and input to the query function). Fig. 15 plots the datasets. Fig. 16 shows the corresponding true query functions and the function learned by NeuroSkech (without partitioning). Sharp changes in the VS dataset caues difficulties for NeuroSketch, leading to inaccuracies around such sharp changes. This is reflected in both AQC and MAE values shown in Table 4 (Norm. AQC is AQC of the functions after they are scaled to $[0, 1]$ to allow for comparions across datasets), where PM and TPC which have less such changes have smaller AQC and MAE.

We use Fig. 16 (a) to illustrate why abrupt changes (i.e., large LDQ) make function approximation difficult. Observe in Fig. 16 (a) such an abrupt change in query function where lat. is between 29.73 and 29.8 (the begning and end of the linear piece are marked in the figure with vertical lines). We see that a single linear piece is assigned to approximate the function in that range (recall that ReLU neural networks are piece-wise linear functions). Such a linear piece has high error, as it cannot capture the (non-linear) change in the function. The error resuling from this approximation grows as the magnitude of the abrupt change in the true function increases. Alternatively, more linear pieces are needed to model the change in the function, which results in a larger neural network.

6 RELATED WORK

Answering RAQs. The methods for answering RAQs can be divided into sampling-based methods [7, 12, 14, 26] and model-based methods [8, 13, 15, 23, 31, 35, 44]. Sampling-based methods use different sampling strategies (e.g., uniform sampling, [14], stratified sampling [12, 26]) and answer the queries based on the samples. Model-based methods develop a model of the data that is used to answer queries. The models can be of the form of histograms, wavelets, data sketches (see [13] for a survey) or learning based regression and density based models [15, 23, 35]. These works create a model of the data and use the data models to answer queries.

In the case of learned models, a model is created that learns the data, in contrast with NeuroSketch that predicts the query answer. That is, regression and density based models of [23], generative model of [35] and the sum-product network of [15] are models of the data created independent of potential queries. We experimentally showed that our modeling choice allows for orders of magnitude performance improvement. Secondly, data models can answer specific queries, (e.g. [23] answers only COUNT, SUM, AVG, VARIANCE, STDDEV and PERCENTILE aggregations) while, our framework can be applied to any aggregation function. Finally, our theoretical analysis for using a learned model is novel, in that it studies why and when a neural network can perform well. Such a study is missing across all existing learning based methods.

Furthermore, learned cardinality estimation [17, 20, 36–38] is related to our work, in that it answers COUNT queries. However, we consider general aggregation functions and such methods

do not apply (we also observed that modifying a representative of such approaches, [37], to answer RAQs performed worse than DeepDB in practice). [20] uses neural networks for cardinality estimation and thus our theoretical results are applicable to justify their success. Furthermore, [17] theoretically studies training size needed to learn selectivity function, which is orthogonal to our work.

Neural Network Approximation. To approximate a function f with a neural network, similar to Theorem 3.4 but under different settings, existing work [11, 18, 22, 28, 33, 34, 39–41] characterize neural network size, s, in terms of its error, ε, in the form $s = C_1\varepsilon^{-dC_2}$, where C_1 and C_2 depend on properties of f. The works differ in their notions of *size* and assumptions on f, leading to different C_1 and C_2 values. Closest to our setting, [18, 28, 33, 34] bound approximation error for Lipschitz functions for a given *number of neural network parameters*, but don't consider the storage cost. Storage cost cannot be related to the number of parameters if the magnitude of the parameters are unbounded, as is the case in [18, 33, 34]. [28] also does not explicitly bound the storage cost, but analyzing their construction yields a bound that, compared to our result, is exponentially worse in ρ and polynomially worse in d.

7 CONCLUSION

We presented the first DQD bound for an ML method when answering RAQs. Our DQD bound shows how the error of a neural network relates to the data distribution, data size and the query function. Based on our DQD bound, we introduced NeuroSketch, a neural network framework for efficiently answering RAQs, with orders of magnitude improvement in query time over the state-of-the-art algorithms. A NeuroSketch trained for a query function is typically much smaller than the data and answers RAQs without accessing the data. This is beneficial for efficient release and storage of data. For instance, location data aggregators (e.g., SafeGraph [1]) can train a NeuroSketch to answer the average visit duration query, and release it to interested parties instead of the dataset. This improves storage, transmission and query processing costs for all parties. Future work can focus on DQD bounds for high dimensions and studying approximation error for separate function classes. Our Lipschitz assumption is very generic (only assumes a bound on the function derivative magnitude), and can yield a loose bound in high dimensions or for some functions classes (e.g., linear functions that can have large derivative magnitude but are easy to approximate). Additionally, modeling impact of query workload on neural network accuracy, as well as studying parallelism and model pruning methods [10] to remove *unimportant* model weights for faster evaluation time. Support for dynamic data is another interesting future direction. One approach is to frequently test NeuroSketch, and re-train the neural networks whose accuracy fall below a certain threshold. We conjecture that DQD can be used to decide how often retraining is required.

ACKNOWLEDGEMENT

This research has been funded in part by NSF grants IIS-1910950, CNS-2125530, and IIS-2128661, NIH grant 5R01LM014026, and an unrestricted cash gift from Microsoft Research. Vatsal Sharan was supported by NSF CAREER Award CCF-2239265 and an Amazon Research Award. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the NSF.

REFERENCES

[1] 2020. SafeGraph dataset. https://docs.safegraph.com/v4.0/docs/places-schema#section-patterns. Accessed Dec 29th, 2020.
[2] 2020. Veraset Website. https://www.veraset.com/about-veraset. Accessed: 2020-10-25.
[3] 2021. Parameter Queries (Visual Database Tools). https://docs.microsoft.com/en-us/sql/ssms/visual-db-tools/parameter-queries-visual-database-tools?view=sql-server-ver15. Accessed Jun 30th, 2021.

[4] 2021. Parameterized query. https://node-postgres.com/features/queries. Accessed Jun 30th, 2021.

[5] 2021. Parameterized query. https://docs.data.world/documentation/sql/concepts/dw_specific/parameterized_queries.html. Accessed Jun 30th, 2021.

[6] 2022. Optuna. https://optuna.org/. Accessed Feb 21st, 2022.

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

[8] Ritesh Ahuja, Sepanta Zeighami, Gabriel Ghinita, and Cyrus Shahabi. 2023. A Neural Approach to Spatio-Temporal Data Release with User-Level Differential Privacy. *Proceedings of the 2023 International Conference on Management of Data, SIGMOD '23* (2023). arXiv preprint arXiv:2208.09744.

[9] Martin Anthony and Peter L. Bartlett. 1999. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press. https://doi.org/10.1017/CBO9780511624216

[10] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033* (2020).

[11] Helmut Bolcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. 2019. Optimal approximation with sparsely connected deep neural networks. *SIAM Journal on Mathematics of Data Science* 1, 1 (2019), 8–45.

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

[13] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* 4, 1–3 (Jan. 2012), 1–294. https://doi.org/10.1561/1900000004

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

[15] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7 (2019).

[16] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2021. DeepDB Implementation. https://github.com/DataManagementLab/deepdb-public. Accessed May 21th, 2021.

[17] Xiao Hu, Yuxi Liu, Haibo Xiu, Pankaj K. Agarwal, Debmalya Panigrahi, Sudeepa Roy, and Jun Yang. 2022. Selectivity Functions of Range Queries Are Learnable. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 959–972. https://doi.org/10.1145/3514221.3517896

[18] Changcun Huang. 2020. ReLU Networks Are Universal Approximators via Piecewise Linear or Constant Functions. *Neural Computation* 32, 11 (11 2020), 2249–2278. https://doi.org/10.1162/neco_a_01316 arXiv:https://direct.mit.edu/neco/article-pdf/32/11/2249/1865413/neco_a_01316.pdf

[19] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research* (2018).

[21] 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.

[22] Jianfeng Lu, Zuowei Shen, Haizhao Yang, and Shijun Zhang. 2021. Deep network approximation for smooth functions. *SIAM Journal on Mathematical Analysis* 53, 5 (2021), 5465–5506.

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

[24] Qingzhi Ma and Peter Triantafillou. 2020. DBEst Implementation. https://github.com/qingzma/DBEst_MDN. Accessed Dec 21th, 2020.

[25] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS *(VLDB '06)*. VLDB Endowment, 1049–1058.

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

[27] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2021. VerdictDB Implementation. https://github.com/verdict-project/verdict. Accessed Jul 6th, 2021.

[28] Philipp Petersen and Felix Voigtlaender. 2018. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *Neural Networks* 108 (2018), 296–330.

[29] Allan Pinkus. 1999. Approximation theory of the MLP model in neural networks. *Acta numerica* 8 (1999), 143–195.

[30] Douglas A Reynolds. 2009. Gaussian Mixture Models. *Encyclopedia of biometrics* 741 (2009).

[31] Rolfe R Schmidt and Cyrus Shahabi. 2002. Propolyne: A fast wavelet-based algorithm for progressive evaluation of polynomial range-sum queries. In *International Conference on Extending Database Technology*. Springer, 664–681.
[32] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge university press.
[33] Zuowei Shen, Haizhao Yang, and Shijun Zhang. 2019. Nonlinear approximation via compositions. *Neural Networks* 119 (2019), 74–84.
[34] Zuowei Shen, Haizhao Yang, and Shijun Zhang. 2020. Deep Network Approximation Characterized by Number of Neurons. *Communications in Computational Physics* 28, 5 (2020), 1768–1811. https://doi.org/10.4208/cicp.OA-2020-0149
[35] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1309–1320.
[36] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.
[37] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.
[38] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment* 13, 3 (2019), 279–292.
[39] Dmitry Yarotsky. 2017. Error bounds for approximations with deep ReLU networks. *Neural Networks* 94 (2017), 103–114.
[40] Dmitry Yarotsky. 2018. Optimal approximation of continuous functions by very deep ReLU networks. In *Conference on learning theory*. PMLR, 639–649.
[41] Dmitry Yarotsky and Anton Zhevnerchuk. 2020. The phase diagram of approximation rates for deep neural networks. *Advances in neural information processing systems* 33 (2020), 13005–13015.
[42] Yang Ye, Yu Zheng, Yukun Chen, Jianhua Feng, and Xing Xie. 2009. Mining individual life pattern based on location history. In *2009 tenth international conference on mobile data management: Systems, services and middleware*. IEEE, 1–10.
[43] Chulhee Yun, Suvrit Sra, and Ali Jadbabaie. 2019. Small ReLU networks are powerful memorizers: a tight analysis of memorization capacity. In *Advances in Neural Information Processing Systems*. 15558–15569.
[44] Sepanta Zeighami, Ritesh Ahuja, Gabriel Ghinita, and Cyrus Shahabi. 2022. A Neural Database for Differentially Private Spatial Range Queries. *Proc. VLDB Endow.* 15, 5 (jan 2022), 1066–1078. https://doi.org/10.14778/3510397.3510404
[45] Sepanta Zeighami, Cyrus Shahabi, and Vatsal Sharan. 2022. NeuroSketch: A Neural Network Method for Fast and Approximate Evaluation of Range Aggregate Queries (Technical Report). (2022). https://arxiv.org/abs/2211.10832.
[46] Sepanta Zeighami, Cyrus Shahabi, and Vatsal Sharan. 2022. NeuroSketch Implementation. https://github.com/szeighami/NeuroSketch.
[47] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

Received July 2022; revised October 2022; accepted November 2022