A Neural Database for Answering Aggregate Queries on Incomplete Relational Data

Sepanta Zeighami, Raghav Seshadri, Cyrus Shahabi

Abstract—Real-world datasets are often incomplete due to data collection cost, privacy considerations or as a side effect of data integration/preparation. We focus on answering aggregate queries on such datasets, where data incompleteness causes the answers to be inaccurate. To address this problem, assuming typical relational data, existing work generates synthetic data to complete the database, a challenging task, especially in the presence of bias in observed data. Instead, we propose a paradigm shift by learning to directly estimate query answers, circumventing the difficult data generation step. Our approach, dubbed NeuroComplete, learns to answer queries in three steps. First, NeuroComplete generates a set of queries for which accurate answers can be computed given the incomplete dataset. Next, it embeds queries in a feature space, through which each query is effectively represented with the portion of the database that contributes to the query answer. Finally, it trains a neural network in a supervised learning fashion: both query features (input) and correct answers (labels) are known. The learned model generates accurate answers to new queries at test time, exploiting the generalizability of the learned model in the embedding space. Extensive experimental results on real datasets show up to 4 times for `AVG` queries and 10 times for `COUNT` queries error reduction compared with the state-of-the-art.

Index Terms—Relational Database, Missing Data, Analytical Queries, Machine Learning

1 Introduction

Real-world databases are often incomplete [13], [15], [18], [19], [20], [26], [28]. One reason is data collection cost. To know housing prices in an area, collecting information for every house is costly, if not impossible, (US Census spends $1.505 billion yearly for door-to-door data collection [5]), but Airbnb already provides a sample for free [1] (dataset is a sample because it only contains Airbnb prices and not other housing sources). Another reason is privacy. Studies show lower response rate to questions regarding sensitive attributes, e.g., income, in surveys [23], [24]. A landlord may provide their demographic information in a survey, but is less likely to list their properties and prices. Another reason is data integration across databases with *schema mismatch* [8], [9], [15], [16]. Two different agencies may track housing prices in two different regions. One region may track both housing and landlord information while the other only stores housing information. After integrating the databases, landlord information will be incomplete.

In all such scenarios, some records are entirely missing from the datasets. Given a dataset, one often knows whether data is incomplete by comparing aggregate statistics [13], [18], [19], [20] (e.g., Census population counts), by inspecting the mismatch of records within the database [28] (e.g., when an individual's record does not appear in certain tables but exists in others), or through knowledge of schema mismatches [8], [9], [15] known during data integration.

Meanwhile, OLAP applications require answering aggregate queries on such incomplete datasets, yielding inaccurate answers. Consider the example of average housing price in an area. Richer landlords may be less willing to share the cost of their houses, leading to an overall underestimation of the housing prices in a region.

In this paper, we assume the underlying incomplete data is stored in a relational database, where some records are entirely missing from some tables. We focus on answering aggregate queries, that is, SQL queries that ask for aggregation of some attribute, optionally with `WHERE`, `JOIN` and `GROUP BY` clauses on other attributes. Relational datasets cover many (if not all) of the discussed applications. If data is missing due to data integration across databases, the data is already likely from an OLTP system and in a relational format. If one wants to use public data sources (e.g., information about a city) in a query, such information can also be added as a table to a relational database. A relational setting allows for a systematic study of answering aggregate queries on incomplete datasets. In such a setting, a table is systematically missing some of its records.

Recent work studies answering queries on incomplete datasets [15], [20], [34]. The only existing approach for relational datasets, ReStore [15], generates new data to *complete* the existing database based on the existing foreign key relationship. ReStore's data generation step can be seen as an extension of data imputation methods (that impute missing attributes [10], [22], [29], [32]) to impute entire missing records. For instance, given a complete table of landlords but incomplete table of apartments, ReStore [15] generates synthetic apartments for landlords whose apartments are missing. However, synthetic data generation is challenging. (1) The model needs to learn fine-grained and record-level information from an often small and biased training set. (2) Real-world datasets often contain missing attributes, based on which generating synthetic data can be inaccurate. For example, landlord's gender might be missing for some landlords, making it more difficult to accurately create synthetic apartments for them. (3) Generated data that respects foreign key relationships is challenging, since multiple foreign key relationships per table are possible. In [15], since only one foreign key relationship (or path [15]) is used to generate data, information available in other tables that can potentially improve accuracy is effectively ignored.

We propose a paradigm shift from generating synthetic data to learning a model that directly estimates the query

• *Authors are with the University of Southern California, CA 90089. E-mail: {zeighami, rseshadr, shahabi}@usc.edu*

answers. Such a model takes queries as input and directly outputs the query answer, bypassing the data generation step. This approach avoids the above short-comings: (1) since the goal is answering aggregate queries, such a model will be able to learn aggregate information of interest without being hampered down by record-level details, (2) which also makes it less sensitive to missing attributes. (3) Since it does not generate new data, foreign key relationships and the relational structure impose no constraints. Nonetheless, accurately learning the query answers is non-trivial. An approach that learns to mimic observed query answers will fail, since the model learns the wrong answers from the biased observed query answers.

We introduce NeuroComplete, an approach that utilizes query embedding and neural networks to accurately estimate query answers. NeuroComplete learns to answer queries in three steps. First, it generates a set of training queries for which accurate answers can be computed given the incomplete dataset. Intuitively, any query that is "restricted" such that its answer only depends on the data in the incomplete database can be answered accurately. Next, NeuroComplete extracts a set of features for each of these queries. Each feature corresponds to the contextual information available about the query answers in the database, and is computed based on how related a database record is to the query. Finally, NeuroComplete trains a neural network in a supervised learning fashion to learn a mapping from the embedding space (i.e., query features) to query answers. The learned model then generates accurate answers to new queries at test time, exploiting the generalizability of the learned model in the embedding space.

Our experimental results on real-world datasets show that NeuroComplete provides up to **4x and 10x reduction in error** for `AVG` and `COUNT` queries, respectively, compared with state-of-the-art, ReStore [15]. The amount of data required for accurate answers depends on how biased the observed data is. Our results show that NeuroComplete provides accurate answers when 5% (or more) of the data is available, and the data is less biased, while we see that 40% of the data needs to be observed in more biased settings.

Specifically, our contributions are as follows.

- We present NeuroComplete, a query modeling approach that estimates query answers on incomplete databases without synthesizing new data
- NeuroComplete is the first approach that uses generalization in the query embedding space as an effective method to address data bias and incompleteness.
- We present novel training set generation and query embedding techniques to train a model whose query answers generalize to the complete database
- Our experiments on real-world datasets show that NeuroComplete provides up to **4x and 10x reduction in error** for `AVG` and `COUNT` queries, respectively, compared with state-of-the-art, ReStore [15]

2 Definitions and Overview

Aggregate Queries on Relational Database. Consider a relational database, D, with k tables, T_1, ..., T_k. Foreign key relationships connect (some of) the tables. Each table has a primary key, which we assume to be a column named id and uniquely identifies the rows within each table. We

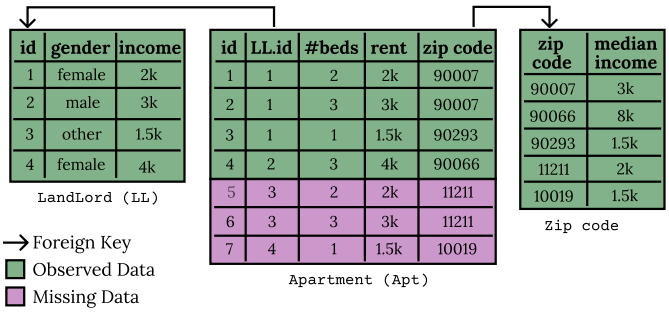


Figure 1. Running Example of Apartments Dataset

consider analytical queries, q, on this database. Informally, q asks for an aggregation of an attribute in some table, where the records in the table are filtered based on some predicate. Formally, q consists of an aggregation function, $\texttt{AGG}_q$, on an attribute M_q of a table T_{i_q}, where M is called the measure attribute. It furthermore consists of a predicate function $P_q(D)$ that, when applied to D, returns a subset of T_{i_q}. We call the set of rows that satisfy a predicate the *matching rows* of the predicate. Such a query can be represented as a SQL statement that asks for aggregation of some attribute, with `WHERE` and optionally `JOIN` and `GROUP BY` clauses on other attributes.. The answer to the query q is $\texttt{AGG}_q(P_q(D).M_q)$. We define the *query function* $f(q)$ as $f(q) = \texttt{AGG}((P(D)).M)$. We drop the dependence of `AGG`, P and M on q when it is understood from the context. The predicate P can be based on the attributes in T_i or T_j, for $j \neq i$, and applied to T_i through `JOIN` of the tables. To simplify the discussion, we do not consider the `GROUP-BY` clause for now, but, in Sec. 5.1, we show how it can be incorporated into queries. Our experiments include queries with `GROUP-BY` and `JOIN` clauses (see Sec. 6.1). We focus on aggregate queries. Our approach estimates query answers by learning patterns of the query answers. Answering non-aggregate queries requires memorizing specific data points, and thus cannot be supported by our approach.

We use Fig. 1 as our running example. The figure shows a database of apartments, their landlord and the zip code for the apartments. An analytical query on this database can ask for average rent for apartments whose landlord is female.

Incomplete Database. We consider the case when we only have access to a subset of records, $\bar{T}_i$ of the table T_i for some $i \in \{1, ..., k\}$ (tables T_j, $j \neq i$ being incomplete is discussed in Sec. 5). We refer to table T_i as *incomplete* or *partially observed* and refer to tables T_j, $j \neq i$ as *complete* or *fully observed*. We let the incomplete database $\bar{D}$ be the database consisting of $\bar{T}_i$ and T_j for all j. We often refer to D (respectively, T) as the true database (resp., true table) and $\bar{D}$ as observed database (resp., observed table). Finally, we define the *observed query function*, $\bar{f}(q)$, as $\bar{f}(q) = \texttt{AGG}((P(\bar{D})).M)$. We consider the case when the observed database is a biased sample of the true database, i.e., $\mathbb{E}_{\bar{D}\sim D}[\bar{f}(q)] \neq f(q)$. Thus the error in answering queries on the observed database isn't only due to the variance in sampling, but also due to its bias. We denote by $n = |T_i|$ and $\bar{n} = |\bar{T}_i|$, the size of the observed table and true table, respectively.

In our running example, we assume apartments table is incomplete, where data records missing are marked with a different colour in Fig. 1. Answering the average rent query on the observed database will lead to incorrect answers.

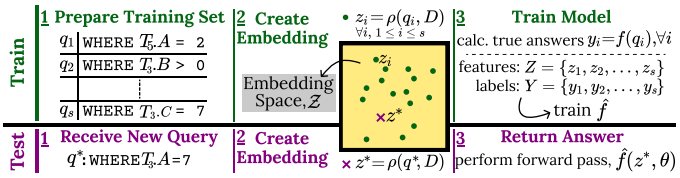


Figure 2. NeuroComplete Framework

Problem Definition. The goal of this paper is to, given the observed database, $\bar{D}$, answer a query q so that its answer is similar to $f(q)$. However, performing the query on the observed database, $\bar{D}$, provides an inaccurate answer $\bar{f}(q)$. Using $\bar{D}$, we train a model $\hat{f}(.;\theta)$ that takes the query as an input and outputs an estimate of its answer. The model is trained given only $\bar{D}$, but its answer is expected to be similar to performing queries on D. The asked queries can have arbitrary predicates (our approach makes no assumption on the form of the predicates, and in practice, we've evaluated our approach on common predicates with equality and inequality across multiple attributes), a fixed aggregation function AGG and a fixed measure attribute M (different models can be learned for different AGG and M values, as discussed in Sec. 5). Let $\mathcal{Q}$ be the set of all such queries from a query workload. Formally, we study

Problem 1. Given access only to an observed incomplete database $\bar{D}$, train a model, $\hat{f}$, so that $\frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}}|\hat{f}(q;\theta) - f(q)|$ is minimized, where f is the query function corresponding to the complete database D.

In our running example, the goal is to train a model that can utilize the observed database to answer queries that ask for `AVG(rent)` (for any query predicate) more accurately than merely calculating the answer on the database.

System Setup. We follow the setup of [15] and ask the users to (1) annotate tables with missing records and (2) annotate rows that have *complete* foreign key relationships, where for such rows, the foreign keys are not missing. If data incompleteness is due to schema mismatch [9], [15], [16] during data integration (e.g., because a table that exists in one database does not exist in another), such annotations are known and do not add any manual overhead. In our running example, we can mark landlords stored in the LA dataset to have complete foreign key relationships (recall that LA dataset contained landlord and apartment tables and thus the foreign key relationships in LA database are complete, while NY database only contained landlord table they do not have complete foreign key relationships). Furthermore, such annotations can be provided by inspecting available aggregate statistics [13], [18], [20]. For instance, if the number of users in an area is lower than the available Census population, the records in that area will be incomplete. Finally, the incompleteness can be known from means of data collection, e.g., collected dataset might be for a certain region (such as Foursquare dataset collected in New York and Tokyo [30]), so one can readily infer data incompleteness (see case-study in Sec. 6.7). For ease of discussion, for now, we also assume that the size, n, of true table T_i is known. We relax this assumption in Sec. 5.2.

2.1 NeuroComplete Framework

NeuroComplete embeds queries into a space $\mathcal{Z}$ and trains a model, $\hat{f}$, from $\mathcal{Z}$ to query answers. To do so, NeuroComplete defines an embedding function ρ that takes a query

Algorithm 1 NeuroComplete Framework

Input: Observed database $\bar{D}$, query function $\bar{f}$, training size s
Output: Neural network $\hat{\hat{f}}$

procedure TRAINNEUROCOMPLETE($\bar{D}$, $\bar{f}$, s)
  $Q \leftarrow$ GENERATEQUERIES($\bar{D}, s$) ▷ Generate training set
  $Z \leftarrow \{z_i = \rho(q_i, \bar{D}), 1 \le i \le s\}$ ▷ Create embedding
  $Y \leftarrow \{y_i = \bar{f}(q_i), 1 \le i \le s\}$
  Initialize the parameters, θ, of $\hat{f}(.;\theta)$
  repeat
    Sample a set of indexes, I, up to at most $|Y|$
    Update θ in direction $-\nabla_\theta \sum_{i\in I} \frac{(\hat{f}(z_i;\theta)-y_i)^2}{|I|}$
  until convergence
  return $\hat{f}$

Input: Test time query q^* on database $\bar{D}$
Output: Estimated answer for query q^*

procedure USENEUROCOMPLETE(q^*, $\bar{D}$)
  $z^* \leftarrow \rho(q^*, \bar{D})$
  if q^* is count-sensitive **then**
    return $\frac{n}{\bar{n}} \times \hat{f}(z^*;\theta)$
  else
    return $\hat{f}(z^*;\theta)$

q as an input and outputs an embedding z. To answer any query, q, we first find $z = \rho(q)$ and then provide the estimate $\hat{f}(z;\theta)$ for the query answer. The input to the neural network is a query embedding (detailed in Sec. 4), which represents the query in terms of the observed information related to the query. Intuitively, the embedding function ρ (formally defined in Sec. 4) aggregates the observed database rows based on how related they are to the query, to represent the query in terms of such relevant information.. This process is shown in Fig. 2. During training, NeuroComplete (1) creates a set, Q, of queries for the purpose of training, (2) uses the embedding function, ρ, to find the query embedding for the queries in Q, and (3) uses the queries together with their answer (computed on the observed database) to train a neural network $\hat{f}$ in a supervised learning setting. The neural network learns a mapping from the embedding space to query answers. To answer a query, NeuroComplete first finds its query embedding and performs a forward pass of the trained neural network with the embedding as its input to provide an estimate of the query answer.

Because the database is incomplete, it is non-trivial to generate a training set with accurate labels or to define an embedding function that allows for the desired model generalizability, challenges that are addressed in the remaining of this paper. We first use Alg. 1 to concretely present NeuroComplete framework. Secs. 3 and 4, respectively, present training set generation and query embedding in details and Sec. 5 discusses the final NeuroComplete system.

Our discussion makes a distinction between count-sensitive and count-insensitive aggregations. Count-sensitive aggregations are aggregation functions where scale of the answers changes with the size of the database. COUNT and SUM belong to this category because the answer to such queries increases with data size. On the other hand, count-insensitive aggregation functions are queries where the scale of the answer does not depend on the number of data points,

Algorithm 2 Training Query Generation

Input: The observed database $\bar{D}$ and training size s
Output: A query set, Q

```
1: procedure GENERATEQUERIES(D̄, s)
2:     Q ← ∅
3:     I ← set of ids of rows in T̄_i
4:     for i ← 1 to s do
5:         A ← a randomly selected attribute from T_i
6:         v ← a value in range of A
7:         op ← one of ≤, ≥ or =
8:         q ← "SELECT AGG(M) FROM T_i WHERE A op v"
9:         q += "AND T_i.id IN I"
10:        Q.append(q)
       return Q
```

e.g., AVG and MEDIAN. We make this distinction to improve our modeling, because, when answering count-sensitive queries, one needs to take into account the size of the database, while count-insensitive queries can be answered without explicitly accounting for database size.

NeuroComplete Training. TRAINNEUROCOMPLETE in Alg. 1 shows the NeuroComplete training procedure. Line 2 corresponds to *training set generation* where a set of queries, Q, are created for the purpose of model training. The function GENERATEQUERIES($\bar{D}$) takes the observed database $\bar{D}$ as an input and generates queries for the purpose of training. We present how to define this query generation function for accurate training on incomplete databases in Sec. 3. After training set generation, line 3 creates query embeddings for the generated training set using the embedding function ρ. We present the embedding function in Sec. 4. Finally, lines 4-9 correspond to model training where the training labels are calculated and a neural network is trained using stochastic gradient descent and with mean squared loss. Line 7 in the algorithm samples a set of indexes I to generate the current batch for training, which are the indexes of queries used in training for the current batch. That is, after sampling I, the current training batch is $\{(z_i, y_i), i \in I\}$.

Answering Queries. After the model is trained, for a test query q, we first find its embedding, by calling embedding function ρ and then performing a forward pass of the trained model with the embedding as an input. If the query is count-insensitive, the estimate for the query answer is the output of the model. Otherwise, the query answer is scaled based on the ratio of the observed data size to the true data size to account for the scale of the answers.

3 TRAINING SET CREATION

This step generates the training queries. Since the observed database is incomplete, the answer to most queries on the observed database will be inaccurate and training a model using such queries can lead to an inaccurate model. Consider a training query q. If $P_q(D)$ contains rows in T but not in $\bar{T}$, then $\bar{f}(q) \neq f(q)$ and thus, the training label created for query q will be wrong. The challenge is creating queries for which we can calculate correct training labels.

Restricted Queries. Our main insight is to *learn from restricted queries*. We define restricted queries as queries whose answers are the same in both $\bar{D}$ and D. Intuitively, if we restrict the database to the observed database the answer to restricted queries does not change. Formally, define $\mathcal{Q}_r = \{q \in \mathcal{Q}, \bar{f}(q) = f(q)\}$. Most real-world queries are not restricted. For instance, in our running example (Fig. 1), the query of `AVG(rent)` of apartments whose landlord is female is not restricted (its answer on the observed database is different from the answer on the true database). However, the query of `AVG(rent)` of apartments whose id is equal to 1 or 2 is a restricted query (since apartment ids 1 and 2 are in the observed database, and therefore, the correct answer can be evaluated by only using the observed database).

Training labels created based on restricted queries are accurate, so that learning from restricted queries creates a model that learns an accurate mapping from queries to their true answers. However, it is difficult to verify if a given query belongs to $\mathcal{Q}_r$, without access to D. Nonetheless, it is easy to generate restricted queries. Given any query q we can create a restricted query q' by adding a conjunctive clause to the predicate of q. Let I be the set of id values of the rows in the observed incomplete table $\bar{T}_i$. We can create a conjunction between the predicate of q and the statement $T_i.id$ `IN` I. Since primary keys are unique, such a query will only match records whose id is in I and thus are in $\bar{T}_i$.

Example. In our running example (Fig. 1), consider the query of `AVG(rent)` of apartments whose landlord is female. Performing this query on the observed database results in a wrong answer, because the apartment with id=7 matches the predicate but is not in the observed database. Nonetheless, we can turn this query into a restricted query. The query of `AVG(rent)` of apartments whose landlord is female and whose apartment.id is one of 1, 2, 3 or 4 is a restricted query and can be answered accurately from data.

Query Generation. Any query can be turned into a restricted query, so the query generation process can use any existing query. For instance, if a query workload is available, each query in the workload can first be restricted to the observed database and then used for training.

In the absence of a query workload, our query generation process creates synthetic predicates by randomly picking an attribute, a value for the attribute and an operation among $\leq$, $\geq$ and =. The generated query is then modified to be restricted to the observed database. This process is shown in Alg. 2, where for a desired number of queries s, the algorithm defines a predicate in lines 5-8. In line 7, we use '=' for categorical attributes and '$\leq$' or '$\geq$' for numerical attributes. Finally, line 9 turns the query into a restricted query by ensuring that it only matches the records in the observed database. We note that both more sophisticated query generation approaches, such as [35] or extending Alg. 2 to generate more predicate clauses per query, or contain joins, are possible. Nonetheless, we observed this query generation process to be sufficient. In fact, due to our embedding approach described in Sec. 4, we expect the complexity of the `WHERE` clauses used for training not to have a significant impact on the accuracy of the learned model. This is because our query embedding only depends on the distribution of matching rows to the query, and not the complexity of finding those matching rows.

4 QUERY EMBEDDING

We discuss the query embedding function ρ. We first present the approach in a two table setting (i.e., assuming database only has two tables, one fully observed and one with

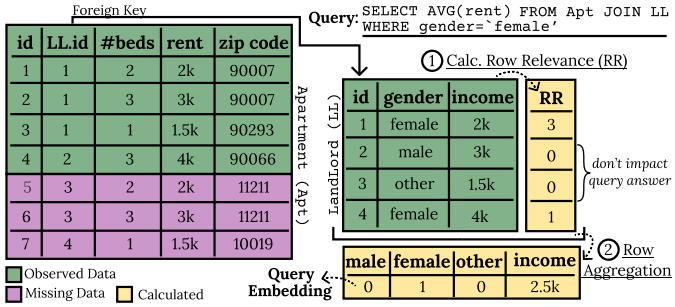


Figure 3. Query Embedding Example

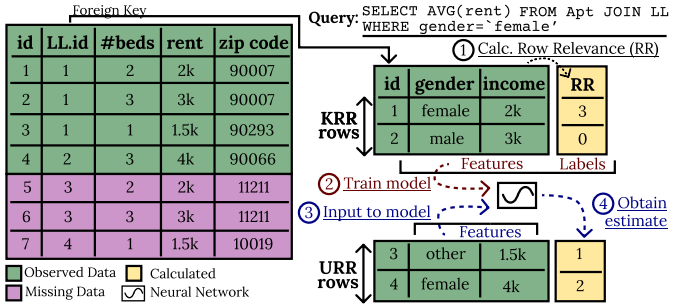


Figure 4. Row Relevance Calculation

missing records) in Secs. 4.1-4.3. For ease of notation, in the two table setup, we call the table T_i that contains missing records T (and $\bar{T}$ is the observed subset of T) and refer as O to the complete table in the database (i.e., all records in O are observed). During query embedding, we have access to O and $\bar{T}$, but not T. Thus, the incomplete (or observed) database contains tables O and $\bar{T}$. The goal is to answer queries on T (which we do not have access to) using the information available in O and $\bar{T}$. We discuss multi-table setting in Sec. 4.4.

4.1 Overview

Query embeddings are created based on the observed database (we do not have access to the complete database). To do so, we utilize rows in the fully observed table O (and not the incomplete table $\bar{T}$). This is done to avoid biases in the incomplete table $\bar{T}$ affecting our query embedding. In this section, we present an overview of this approach. To better illustrate the main concepts, here, we assume we have access to the complete database. We discuss, in detail, how we generate query embeddings while having access only to the observed database in Secs. 4.2 and 4.3.

We define query embedding as a summary of rows in O that are relevant to the query q. We propose a two step process, where we (1) for each row in O find their *row relevance* (RR), a weight that quantifies how related each row is to the query q and (2) aggregate the rows in O based on the calculated row relevance to represent q in terms of rows of O. An example assuming access to the complete database, is shown in Fig. 3. For the apartment table and a given query, we calculate row relevance of records in Landlord table, and thus the query embedding is based on records in Landlord table and uses its schema (even though the query asks for apartment rent information).

Row Relevance. Row relevance (RR) of a row in O to a query q captures how related the row is to the query answer. Let T_q be the set of matching rows in T for the query q. We define, for a row in O with $O.id = i$ for an integer i, its row relevance α_i to be $\alpha_i = \texttt{COUNT}(\sigma_{O.\text{id}=i}(T_q \bowtie O))$. The above expressions considers the weight of the i-th row as how many times the row appears when O is joined with the matching rows T_q. Intuitively, if α_i is large, it means the i-th row of O has a strong relationship to the set of rows that match the query. If α_i is zero, it means deleting the i-th row, and its related rows in T (i.e., delete with cascade) will have no impact on the query q, and thus, the i-th row should not impact the representation of q. In practice, we cannot calculate row relevance exactly, because we do not have access to the complete database. We discuss in Sec. 4.2 how row-relevance is calculated in practice.

Fig. 3 shows how the row relevance values are calculated in our running example (based on the complete database). We see that for the query shown in Fig. 3, the row relevance for landlord with id 2 and 3 is 0, while landlord with id 1 has RR equal to 3. Intuitively, removing Landlords with id 2 and 3 does not change the query answer (and thus, RR=0) while landlord with id 1 has a significant impact on the query answer (so larger RR).

Row Aggregation. To summarize information in O that relate to the query q, we perform a weighted aggregation of the values in O, weighted according to their RR values. Fig. 3 shows how the rows are aggregated to create the final query embedding in our example. The embedding contains the weighted average of the income of the landlords (i.e., $(2 \times 3 + 4 \times 1)/(3 + 1) = 2.5$) and the distribution of the gender of the landlords (in this case, they are all female).

4.2 Row Relevance Calculation

Row relevance of a record in O is defined as $\texttt{COUNT}(\sigma_{O.\text{id}=i}(T_q \bowtie O))$, where i is the id of the record in O. In practice, we do not have access to the true database, but only the observed database. Because we only see $\bar{T}$, we will not know all the records in $T_q \bowtie O$ and cannot directly evaluate their row relevance. Instead, we estimate the row relevance when it cannot be evaluated exactly based on the observed data. To do so, we divide the rows into two sets: (1) rows with Known Row Relevance (KRR rows) which are rows for which row relevance can be accurately calculated on the observed data, and (2) rows with Unknown Row Relevance (URR rows) which are rows for which row relevance cannot be calculated on observed data.

Known and Unknown Row Relevance. More formally, KRR rows are defined as rows for which $\sigma_{O.\text{id}=i}(T_q \bowtie O) = \sigma_{O.\text{id}=i}(\bar{T}_q \bowtie O)$ and URR rows are the remainder of the table. Given that we do not have access to T, we cannot evaluate if a row is KRR by checking the definition. Here, we describe two conditions used to decide if a row is KRR.

Condition 1. q is a restricted query. If q is restricted, by definition, $T_q \bowtie O$ and $\bar{T}_q \bowtie O$ are the same. Thus, row relevance for all rows in O can be exactly calculated.

Condition 2. O_i has complete foreign key relationship. By definition, if O_i (the row in O with id=i) has complete foreign key relationship, then $\sigma_{O.\text{id}=i}(T \bowtie O) = \sigma_{O.\text{id}=i}(\bar{T} \bowtie O)$. This implies that $\sigma_{O.\text{id}=i}(T_q \bowtie O) = \sigma_{O.\text{id}=i}(\bar{T}_q \bowtie O)$, since T_q and $\bar{T}_q$ are subsets of $\bar{T}$ and T respectively.

Condition 1 implies that for training queries all rows are KRR, so row relevance is exactly calculated based on observed data. Condition 2 means at test time, for some records we can exactly calculate row relevance but for others we need to estimate it. This process is described below.

Row Relevance Calculation for KRR. Row relevance calculation for KRR rows is straightforward. we calculate it exactly by evaluating the expression $\texttt{COUNT}(\sigma_{O.\text{id}=i}(T_q \bowtie O))$. For example, in Fig. 4, this expression can be exactly calculated for landlord with ids 1 and 2. We see that landlord 1 appears three time and landlord 2 appears zero times in $T_q \bowtie O$, so that their RR are 3 and 0 respectively.

Row Relevance Calculation for URR. We learn to estimate the row relevance for URR rows using the calculated row relevance of KRR rows. For a query, let O_{KRR} be the set of KRR rows in O, Y_{KRR} their calculated row relevance and O_{URR} the URR rows. We train a neural network in supervised learning fashion, where O_{KRR} are the training features and Y_{KRR} the training labels. We call this model *row relevance model* to distinguish it from the model that is trained to predict query answers (i.e., in Alg. 1). After training row relevance model, a forward pass of the model estimates row relevance of URR rows.

Fig. 4 shows row relevance calculation in our running example. (1) Row relevance is calculated for the two KRR rows. Then (2) each KRR row is used as a training sample to train a neural network that estimates row relevance. The model takes gender and income as input and outputs an estimate RR. After the model is trained (3) we input the gender and income of the URR rows into the model and (4) obtain RR estimates for the URR rows. Fig. 4 shows that the model estimates RR for landlord 3 to be 1 (while true RR is 0) and RR for landlord 4 to be 2 (while true RR is 1).

4.3 Row Aggregation

We aggregate the rows in O according to the row relevance values. If categorical attributes are present in O, we one-hot encode them before aggregation. We aggregate rows for count-insensitive aggregation functions (e.g., AVG, MEDIAN, STD) and count-sensitive aggregation functions (e.g., COUNT, SUM) differently. Count-insensitive aggregations are aggregation functions where the scale of the answers does not change with the size of the database. Thus embedding does not need to contain information about the number of matching rows. On the other hand, for count-sensitive aggregation functions, the embedding needs to contain information about the number of matching rows to allow the model to adjust to the scale of the answers.

For count-insensitive aggregations, we use the weighted average of the features in O as the query embedding, where the weights are based on row relevance values. For count-sensitive aggregations, we use weighted sum of features in O, normalized by $\bar{n}$ if the queries are restricted or by n if they aren't. By incorporating the total row relevance values in count-sensitive aggregations, we allow the embedding to contain information about the number of matching rows. At the same time, we normalize the embedding by table size to ensure the number of matching rows is considered as a proportion of the table size. This creates an embedding that adjusts to data size while also containing information about the number of matching rows to a query.

Row aggregation creates a semantically meaningful summary of the matching rows in O. For numerical values, the summary is the sum or average of the values. For categorical columns (that are one-hot encoded) the summary shows the distribution of the categories existing in the rows.

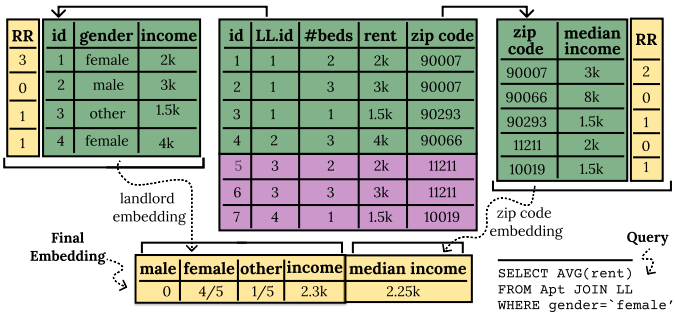


Figure 5. Multi table query embedding

Algorithm 3 Complete Query Embedding Algorithm

Input: A query q on observed database $\bar{D}$
Output: Query embedding

procedure $\rho(q, \bar{D})$
  for all tables T_j **in** $\{T_1, ..., T_k\} \setminus \{T_i\}$ **do**
    for all KRR rows with id=x **in** T_j **do**
      $\alpha_x \leftarrow \texttt{COUNT}(\sigma_{T_j.\text{id}=x}(P_q(\bar{D}) \bowtie T_j))$
    if Any URR row exists in T_j **then**
      $\hat{g}(.;\theta) \leftarrow$ Trained row relevance model
      for all URR rows with id=x **in** T_j **do**
        $\alpha_x \leftarrow \hat{g}(\sigma_{T_j.\text{id}=x}(T_j); \theta)$
    $z_j \leftarrow \sum_x \alpha_x \sigma_{T_j.\text{id}=x}(T_j)$ ▷ Column-wise sum
    if AGG is not count-sensitive **then**
      $z_j \leftarrow \frac{z_j}{\sum_x \alpha_x}$
    else
      if q is a restricted query **then**
        $z_j \leftarrow \frac{z_j}{\bar{n}}$
      else
        $z_j \leftarrow \frac{z_j}{n}$
  return $[z_1 z_2 ... z_k]$

4.4 Multiple Tables and Final Embedding Algorithm

Our approach simply extends to multiple tables by considering each table separately. We iterate over the tables in the database, and for every table T_j, $j \neq i$, and given that the incomplete table is T_i, we consider every T_i and T_j pair. For every pair we repeat the same algorithm as before, which yields a query embedding based on the table T_j. Finally, the embeddings based on each T_j are concatenated together to provide the final query embedding.

Fig. 5 shows the process for our running example, now with all three tables. We first find an embedding using the landlord table, as discussed before. Next, the same process is repeated for the zip code table, to obtain a zip code embedding. The two embeddings are then concatenated to create a single embedding vector shown in the figure.

Final Algorithm. Alg. 3 presents the final query embedding algorithm. The algorithm iterates over the tables and calculates the row embedding by finding row relevance and then performing row aggregation. Finally, all the embedding are concatenated ($[x_1, ..., x_n]$ denotes concatenating $x_1, ..., x_n$) to create the final query embedding.

Performing Joins and Choosing Tables. The notion of joins in the algorithm is overloaded when referring to tables without explicit foreign key relationships with each other. We call it a join between two tables if there exists a non-empty set of foreign key relationships connecting T_i and T_j. We can limit the number of tables used to generate the

embedding based on the length of the path (i.e., number of foreign key relationships connecting T_i and T_j). That is, we can only consider the set of tables that are joinable with T_i through at most a limited number of other tables. This can be beneficial because often the longer the join path is, the less relevant the table to the information in T_i will be. Overall, we let τ be the number of fully observed tables used to create the embedding.

Embedding Time Complexity. For each fully observed table, O (among the τ used for embedding in total), the algorithm goes over rows in $O_{KRR} \bowtie T_q$ to calculate row relevance for the KRR rows. Let $O' = O_{URR} \cup (O_{KRR} \setminus O_{KRR} \bowtie T_q)$, where $O_{KRR} \setminus O_{KRR} \bowtie T_q$ are the KRR rows that don't match the predicate q (so their row relevance is 0). The algorithm then goes over the rows in O', where for KRR rows in O' it sets row relevance to zero, while for URR rows in O' it performs a forward pass of the row relevance model. Assuming training a row relevance model takes t_T, model forward pass takes time t_F, and finding the result of the join $O_{KRR} \bowtie T_q$ takes t_J, the embedding computation takes $O(t_T + t_F \times |O_{URR}| + |O_{KRR}| + |O_{KRR} \bowtie T_q| + t_J)$. This process is repeated τ times, each time for a different fully observed table O. We perform the process in parallel across the τ tables. In our experiments, this process takes 4-15 seconds across all settings (see Sec. 6.5), which is comparable to performing queries on the true (much larger) database, where the cost of performing joins is higher.

5 End-to-End System and Discussion

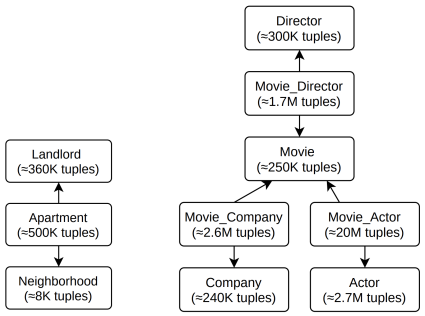
5.1 End-To-End System

Setup. NeuroComplete setup requires minimal effort to (1) annotate tables with missing records and (2) annotate rows for which complete foreign key relationship is available. As discussed in Sec. 2, such information is often readily available as a result of the database integration processes. In this setup, NeuroComplete will accompany a relational database system for table with missing data.

Supported Queries. The query answering process follows Alg. 1, where a model is first trained and then used to answer the query. A NeuroComplete model is trained to answer queries with aggregation `AGG` of an attribute $T.M$, where M is an attribute in table T. Thus, after a NeuroComplete model is trained, it can answer queries with any predicate that ask for `AGG`($T.M$). Such queries can contain `JOIN` or `GROUP BY` clauses as well as any SQL predicates (in fact, NeuroComplete supports general predicates, as defined in Sec. 2, e.g., arbitrary polygons). NeuroComplete supports `GROUP BY` by iteratively estimating the query answer for each group in the `GROUP BY` by adding the group membership as a predicate to the query.

5.2 Further Considerations

Efficiency Considerations. Recall that to answer a query, we first obtain a query embedding (where we utilize row relevance models) and perform a forward pass of the NeuroComplete to obtain the query answer estimate. For efficient querying, we train NeuroComplete models at a preprocessing step and use them at query time. A single NeuroComplete model answers all queries for a fixed measure attribute and aggregation function. When measure attribute and/or aggregation function changes for different queries, multiple models may need to be trained to answer different



(a) Housing Schema. **(b) Movie Schema.**

Figure 6. Dataset information [15]

queries. We decide which queries to build a model for based on the incomplete tables and query workload. We build a NeuroComplete model for queries in the workload where measure attribute is in an incomplete table. NeuroComplete models are small (less than 1 MB in all our experiments), and storing several models based on workload is practical. As discussed in Sec. 4.4, query embedding (including row relevance model training) is fast and is done at query time.

More Missing Data. In a database we may have (1) multiple tables with missing records or (2) some records may contain missing attributes. For case (1), our approach can be used without modification, if, in addition to T_i any other table T_j, $j \neq i$ is also incomplete. Nonetheless, given that NeuroComplete relies on T_j tables for query embedding, enough information needs to be available in those tables to allow for accurate predictions. In practice, especially when systematic bias exists in multiple tables, one can choose to exclude tables with missing records from being used in embedding of other queries. For case (2), we need to ensure that row aggregation supports missing values. This is achieved by simply ignoring the massing values when performing row aggregation.

True Data Size, n. So far, we've assumed true data size n, used to scale NeuroComplete answers for count-sensitive queries, is known. In practice, this is often true: such information may be publicly available (e.g., we know population of an area based on census data), data owners may be willing to share such aggregate information (e.g., a house rental agency may release number of apartments they have in an area but not the detailed apartment information) or may be known based on domain knowledge (e.g., a rental agency may be able to estimate the number of apartments they have but there may not be a detailed record of the apartments in the database). If n is not known, we can estimate it using methods similar to those in [15], [20]. We observed that [15] does estimate the true table size accurately, so we use their method for our estimation of true table size. Note that estimating true table size does not require generating accurate synthetic records, and only requires correctly estimating how many records are missing. Thus, if true data size is not known, estimating it is added as an extra step to the NeuroComplete system.

6 Empirical Study

6.1 Experimental Setup

Our experimental setup largely follows [15]. Each experiment uses a real-world dataset. We remove a set of records to obtain a biased subset which is provided to the algorithms

Setup	Dataset	Incomplete Table	Biased Attribute
H1	Housing	Apartment	Price
H2	Housing	Landlord	Response rate
M1	Movies	Movie	Production year
M2	Movies	Director	Birth year

Table 1
Incomplete dataset generation setup

to answer a set of queries. The goal is to answer queries accurately. Experiments were 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.

Complete datasets. We use two real datasets, Housing and Movies, whose schema and size is shown in Fig. 6 (image from [15]). Housing contains information about different Airbnb listings (such as the apartment type, its neighbourhood and landlord) and is obtained from [1]. Movies contains information about movies listed on IMDB (such as their genre, production year, their directors and actors and company that made them) and is obtained from [2]. We use datasets as pre-processed by [15].

Incomplete dataset generation. The incomplete dataset generation is done as follows. First, we pick a table, as the incomplete table, and an attribute from the table, as the *biased attribute*. For a *keep rate* parameter x, we keep $x\%$ of the total records in the incomplete table, i.e., $|\bar{T}| = x \times |T|$. We select this subset $\bar{T}$ based on a *bias factor* parameter, $b \in [0, 1]$. To choose the records, we (1) sort T based on the biased attribute and select the top $|T| \times x \times b$ records (i.e., records with the highest biased attribute value) and (2) select $|T| \times x \times (1 - b)$ records from the remaining records of T (i.e., from records not selected in step (1)) uniformly at random. If $b = 1$, the sample is completely biased and if $b = 0$ the sample is unbiased. Based on the above procedure, we create 2 setups for each dataset, as shown in Table 1.

Test Queries. We consider test queries with `COUNT` and `AVG` aggregation functions and with `JOIN`, `GROUP BY` and/or `WHERE` clauses. None of the test queries are restricted queries, and thus test queries do not overlap with our training queries, all of which are restricted queries. For `AVG` queries, to be able to study the impact of bias on query answers, we let the measure attribute be the same as the biased attribute for each setup (e.g., queries in H1 all ask for `AVG`(price)). We use the same `GROUP BY` and/or `WHERE` clauses as [15]. Each query has a `GROUP BY` and/or `WHERE` clause on a subset of columns shown in Table 2. For example, an `AVG` query in H1 asks for `AVG(price) WHERE room_type=1`. The query involves a `JOIN` if `WHERE/GROUP BY` is on a column from a different table than the measure attribute. For `COUNT` queries, we report results on predicates on the biased attribute. This is to isolate the impact of bias on query answers as otherwise a query answer can be unaffected by our sampling procedure.

Metrics. As discussed above, each setup consists of a set of test `AVG` and test `COUNT` queries Q. For `AVG` queries, we report mean absolute error (MAE), calculated as $\frac{1}{|Q|}\sum_{q\in Q}|f(q) - y|$, where y is the estimated answer. As discussed in Sec.5.1, `GROUP BY` queries are considered as multiple queries, each query with a `WHERE` clause corresponding to a group membership. For `COUNT` queries, to evaluate whether a method de-biases the results (rather than just scaling up the answers), we compare the MAE in normalized counts. That is, if the estimated size of T is $\hat{n}$ and the size of $\bar{T}$ is $\bar{n}$, then we report $\frac{1}{|Q|}\sum_{q\in Q}|\frac{f(q)}{n} - \frac{y}{\bar{n}}|$. For NeuroComplete, we set $\bar{n}$ to be the same as in ReStore. We train NeuroComplete for 5 different random initialization and report the average and standard deviation of MAEs across runs. Compared with [15], we use absolute error instead of relative error due to its robustness when ground truth is close to zero, and we do not present bias reduction since bias reduction is only applicable to methods that generate synthetic data.

Setup	Predicate and Group By attribute
H1	`Apt.room_type`, `Apt.price`, `LL.host_since`, `Apt.property_type`, `Apt.accommodates`,
H2	`LL.host_since`, `LL.response_time`, `LL.response_rate`, `Apt.room_type`
M1	`Movie.genre`, `Movie.production_year`, `Director.birth_country`
M2	`Director.gender`, `Director.birth_year`

Table 2
Testing query predicate and group by attributes

Baselines. We compare NeuroComplete with the state-of-the-art, ReStore [15]. We used their implementation in [6]. ReStore trains a model to generate more data to complete the dataset and answers queries on the completed dataset. For ReStore, we spent a week on parameter tuning, performing extensive parameter search for each setup. For each setting, we ran the model with various possible modeling choices (SSAR vs. AR) and various completion paths, evaluated it on the *test set* and chose the result with the best test set performance. This ensures that ReStore's model hyperparameters are set to best possible, but is an unrealistic evaluation (showing better performance than possible in practice, since in practice we do not know the ground truth for test set queries). Therefore, we call it ReStore$^+$ as a reminder of this unfair advantage. We also use Sample as a baseline, which answers queries only based on the observed samples.

NeuroComplete Implementation. We implemented NeuroComplete in python and JAX (code available at [33]). The model is a 10 layer fully connected neural network with width 60 in each layer, trained with mean squared error loss function (as shown in Alg. 1 line 8) and Adam optimizer. Training consists of 1,000 iterations, and the model with smallest training error is used to perform test queries. Row relevance models have the same architecture as above. We use between 1,000-2,000 training samples across the settings.

6.2 Comparison Results

Results for `AVG`. Figs. 7-10 compare NeuroComplete with other methods across settings for `AVG` queries. Each figure shows, for a setting, how the error changes for different keep rates and bias factors. For NeuroComplete, the shaded area shows one standard deviation above/below error, where standard deviation is over 5 training runs. We observe that NeuroComplete outperforms the baselines across settings in almost all cases, improving accuracy of state-of-the-art by **up to a factor of 4**. For `AVG` queries, NeuroComplete provides large improvements in the housing dataset, while methods are comparable on Movies dataset for `AVG` queries.

Furthermore, NeuroComplete is the most effective when bias factor is less than 1 and when keep rate is less than 80%.

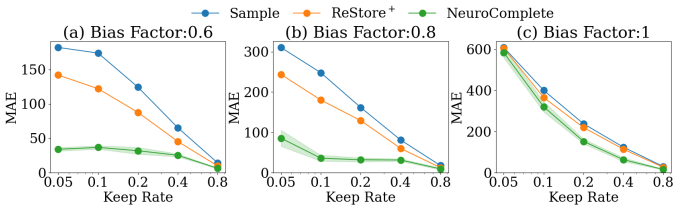


Figure 7. Results for H1 AVG Queries

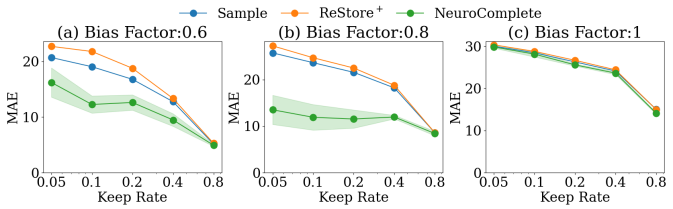


Figure 8. Results for H2 AVG Queries

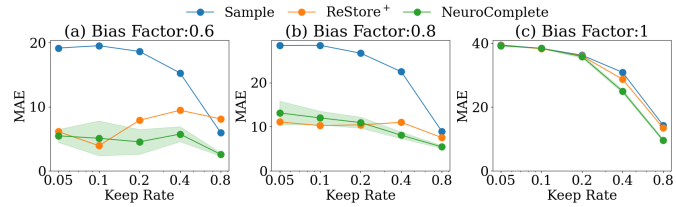


Figure 9. Results for M1 AVG Queries

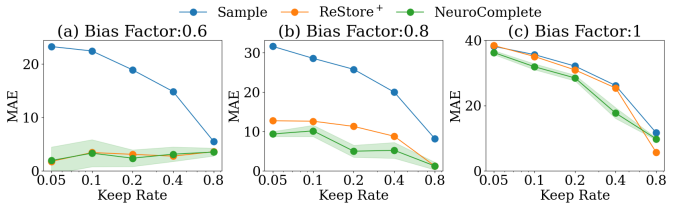


Figure 10. Results for M2 AVG Queries

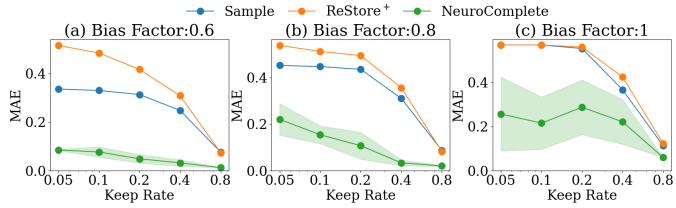


Figure 11. Results for H1 COUNT Queries

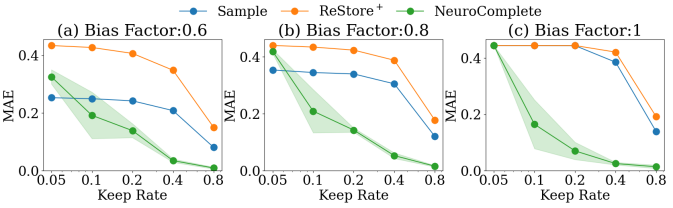


Figure 12. Results for H2 COUNT Queries

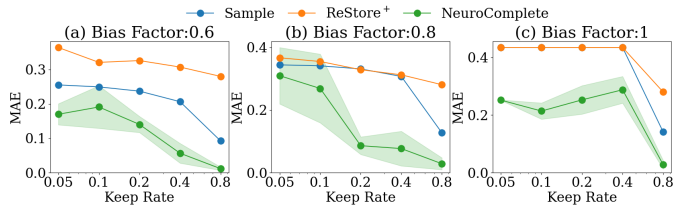


Figure 13. Results for M1 COUNT Queries

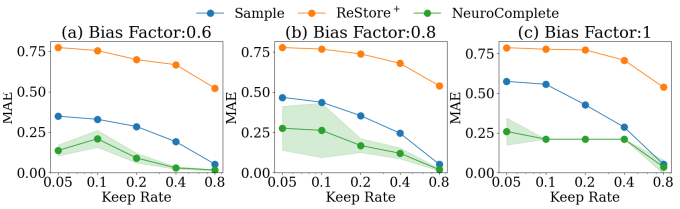


Figure 14. Results for M2 COUNT Queries

When bias factor is 1, NeuroComplete does not see enough variation in query answers during training to be able to accurately extrapolate to unseen queries. On the other hand, when keep rate is 80%, Sample itself is very accurate, and inherent modeling errors do not allow for much improvement for NeuroComplete over observed values.

Interestingly, for bias less than 1, NeuroComplete's error is only marginally impacted by change in keep rate For instance, Fig. 7 (a) shows NeuroComplete's error changes from 40 at 5% to 20 at 80% keep rate, compared with Sample and Restore$^+$ whose error changes from 150 to 20 in the same range of keep rates. This is because NeuroComplete, unlike ReStore$^+$, does not directly use the observed data points for training (i.e., the training size of NeuroComplete is the same independent of the keep rate). On the other hand, NeuroComplete relies on the generalizability of learning based on the observed query embeddings. Thus, results in Fig. 7-10 suggest that generalization in query embedding space is robust to the number of observed data points.

We also see that, in the cases where NeuroComplete error is not affected by increase in keep rate (e.g., Fig. 9 (a) or Fig. 10 (a)), NeuroComplete's standard deviation goes down as keep rate increases. That is, often, more data increases the generalization robustness in NeuroComplete, reducing the reliance on initialization.

Results for COUNT. Figs. 11-14 show the results for COUNT queries. Similar to AVG, NeuroComplete improves the accuracy by multiple factors across settings. Compared with AVG, NeuroComplete is able to improve the accuracy for COUNT aggregation functions even at the bias factor of 1.

Compared with ReStore$^+$, NeuroComplete is always better, **up to a factor of 10**. Our results show that ReStore$^+$ often has larger error than Sample. To understand this result, recall that ReStore$^+$ generates new records. In fact, in most reported settings, total number of records in the database synthesized by ReStore$^+$ closely matches the true number of records. Nonetheless, the distribution of attribute values (measured by our error metric) is further from the ground-truth than in the observed database. For instance, in M2 setup (Fig. 14), we observed that ReStore$^+$ generates many new records to match the number of records in the ground-truth. However, almost none of the newly generated records match the query predicate (while most of the true record do in fact match the predicate). That is, even though the number of records that match the predicate in ReStore$^+$ is closer to ground-truth compared with Sample, the number of records that match the predicate *as a proportion of data size* is further away from ground-truth compared with Sample. Our error metric measures the latter which we believe to be more important (as it measures distribution of the records irrespective of data size).

Finally, for high bias factor or low keep rate, NeuroComplete has higher standard deviation, i.e., not all random neural network initializations converge to a good minima. This shows difficulty of generalization when training queries are from a different distribution than test queries.

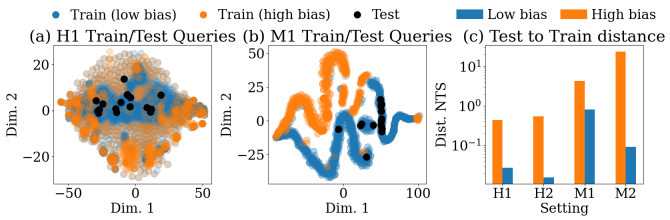


Figure 15. (a) and (b): visualizing training and test distributions. (c): Avg. distance to the nearest training query from test queries.

6.3 Training vs. Test Query Distribution Analysis

We analyze impact of training distribution on NeuroComplete accuracy. We compare two settings: *low bias* defined as keep rate=0.8 and bias factor=0.6 and *high bias* defined as keep rate=0.05 and bias factor=1. In both settings, even though the observed data size is different, NeuroComplete creates the same number of training queries. However, the training queries are embedded differently, resulting in the embedding distribution used for training to be different. This impacts the accuracy of NeuroComplete, since answering test queries depends on how well the model generalizes in the embedding space to the unseen test query distribution. To investigate this, Fig. 15 (a) and (b) show the training and test query embeddings for AVG queries in H1 and M1 settings. We use t-SNE [27] for visualization, which uses neighborhood graphs for dimensionality reduction to allow for visualizing the structure of the high-dimensional space. In this experiment, to isolate the impact of embedding distribution, row relevance for test queries is calculated based on the complete dataset (i.e., assuming a perfectly accurate row relevance model), so that test query embedding is not affected by the data bias.

Fig. 15 (a) and (b) visually show that test embedding distribution is more similar to training embeddings distribution in the low bias setting compared with the high bias setting. Fig. 15 (c) quantifies this similarity. It plots dist. NTS, defined as the average distance to the nearest training sample from test samples. That is, for the test set Q and each test query, $q \in Q$, let d_q be the distance from q to q's nearest training query and define dist. NTS= $\frac{1}{|Q|}\sum_q d_q$. We use Euclidean distance in the original embedding space (without dimensionality reduction). The lower dist. NTS, the more similar training and test query embeddings are. Fig. 15 (c) shows that across datasets, in the low bias setting, test queries are more similar to training queries. This justifies the results in Figs. 7-14, where the increase in error from low bias to high bias setting can be attributed to the increase in distance between train and test embedding distribution. As this distance increases, generalization becomes more difficult, thus accuracy decreases.

6.4 Multiple Incomplete Tables

We evaluate NeuroComplete when there is more missing data beyond a single incomplete table. We introduce missing attributes in tables that were assumed to be complete in previous experiments. Here, experiments are in the H1 setting, where previously Landlord was assumed to be complete. For every landlord attribute and for each record, we remove its value with a probability, dr, referred to as drop rate.

In Fig. 16, we vary dr for COUNT (Fig. 16 (a)) and AVG (Fig. 16 (b)) queries to study its impact on the performance of the models. We observe that this parameter has little impact when keep rate is 20% or 80%, showing the robustness of our approach to missing values. At keep rate 5% for AVG, the error increases when drop rate increases, while for COUNT query the error first increases then decreases. This result suggests NeuroComplete is less robust to missing attributes when observed data is too small.

6.5 Scalability and Efficiency Anlysis

Query Time. Fig. 17 studies query time of the various algorithms across two different settings and for different observed data sizes (each observed data size corresponds to a specific keep rate). We see that Sample is the fastest algorithm, as it performs no processing besides answering queries on the observed data. We see that NeuroCompletes' query time varies between 4 and 15 seconds across settings. Row relevance model training accounts for most of the query time, where models are trained for a fixed number of iterations. The difference in query time across settings is due to the difference in the dimensionality of the embedding space, where H1, which has the highest embedding dimensionality takes the longest. Compared with Restore$^+$ we see that Restore$^+$ answers queries faster in the setting H1 but slower in M1. This is because Restore$^+$ synthesizes data when it receives a query, and how much data it needs to generate depends on the complexity of the relational schema. As a result, it becomes slower in M1, which has a more complex schema, compared with H1.

Training Time. Fig. 18 (a) shows impact of training time on NeuroComplete error in H1 setup with bias factor 0.8 at various keep rates (kr). The lines show the error for different keep rates. Fig. 18 (a) shows average accuracy for 5 different runs, and the shaded area is the standard deviation of model error across runs. Overall, the results show that models fit within a few seconds of training, and more epochs, especially for smaller keep rates causes overfitting. Furthermore, we see that standard deviation is larger for smaller keep rates and the model performance is more sensitive to initialization in that case.

6.6 Number of Training Samples

Fig. 18 (b) shows how model accuracy changes based on number of training samples in H1 setup with bias factor 0.8 at various keep rates (kr), where error drops with more training samples used. Interestingly, even 75 training samples with keep rate 80% performs better than using 2025 training samples at keep rate 5%. This shows that, even with small number of samples, the model can adjust to the scale of required answers, thus providing reasonable estimates.

6.7 Case-Study: Estimating AVG Visit Duration

We present an example where real-world contextual information is represented in relational format and used to provide query answers for queries where data is not complete.
Dataset. We use a dataset of location report of individuals (i.e., latitude and longitude of user locations) which contains the time duration users spent at different locations in a city. Each record is a tuple of the format (lat., lon., duration). Furthermore, each city is divided into various neighbourhoods. The goal is to answer the query of AVG time spent in a neighbourhood by users (i.e., range predicate on lat. and lon, and duration is the measure attribute). We also have a dataset of Point-of-Interests (POIs), where each record

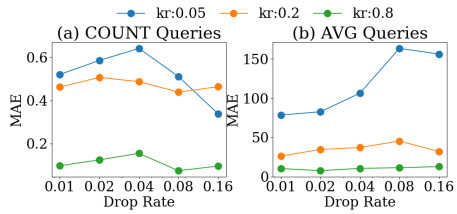


Figure 16. Robustness to Missing Attributes

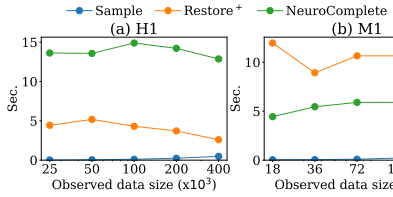


Figure 17. Query Time

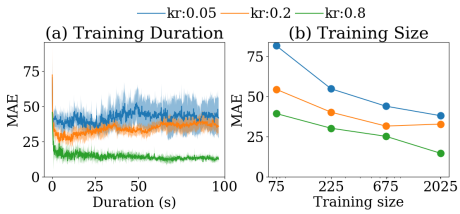


Figure 18. Training size and duration

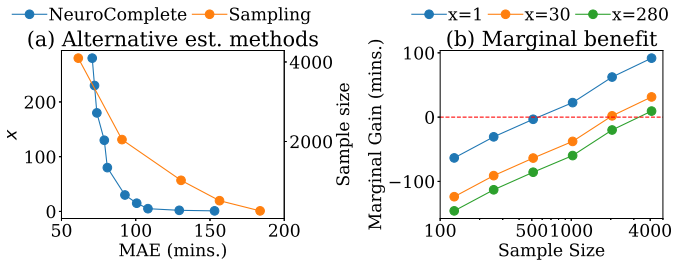


Figure 19. Comparison of Sampling and Learning

consists of lat, lon and POI information. This information can be placed in a relation database with three tables: Visits, Neighbourhoods and POI. Visit table has schema (lat, lon, duration, neighbourhood id), POI table has schema (lat, lon, POI information, neighbourhood id) and neighbourhood table has schema (id, neighbourhood information).

For the visit table, we use Veraset (VS) dataset, a proprietary dataset that contains anonymized location reports of cell-phones across the US collected by Veraset [3], a data-as-a-service company. Each location report contains an anonymized id, timestamp and the latitude and longitude of the location. We performed stay point detection [31] on this dataset (to, e.g., remove location signals when a person is driving), extracted location visits where a user spent at least 15 minutes, and recorded the duration of each visit. 527,932 location visits in downtown Houston were thus extracted to form the dataset used in our experiments, which contains three columns: latitude, longitude and visit duration.

For POI table, we use Safegraph Places [4], a publicly available dataset containing POI information in the US. For Neighbourhood table, we partition city of Houston with a 20x20 grid, yielding 400 neighbourhoods. We use grid location for each neighbourhood, without other neighbourhood specific information. 352 neighbourhoods had at least one visit in VS dataset and we only kept those neighbourhoods.

Incomplete dataset generation. We let Visit table be the incomplete table. We assume we have visits data for some neighbourhoods and no data for others. For a parameter x, we randomly sample a set of, x neighborhoods, keep all visits that fall in those neighbourhoods, and remove the visits for all other neighbourhoods to generate our observed database. We expect to see such a geographical bias in data collection in practice. Many datasets are only available for a single area (e.g., Foursquare [30] covers New York and Tokyo and CABS dataset is only available for San Francisco [21]). Furthermore, for data collected from mobile apps, there is a bias based on who uses the app which translates into location (e.g., older people may not use the app, and there will be less data for areas with older population).

Results. Given x neighbourhoods with data, we apply NeuroComplete to find `AVG` visit duration for neighbourhoods without data. We compare this with the alternative of collecting data for the neighbourhoods without data to answer queries. Fig. 19 (a) depicts these two alternatives. The two lines in Fig. 19 (a) are not directly comparable and they are plotted with different y-axes (NeuroComplete: left axis, Sampling: right axis). *Sample size* refers to number of new points sampled for neighbourhoods without data, to obtain an estimate for the query in those neighbourhoods.

Fig. 19 (a) shows that given an error tolerance level, one has two alternatives in answering the query. For instance, for error of 75 mins, one can either sample around 2,000 points from the neighbourhood in question, or use 130 other neighbourhood's information and train NeuroComplete to obtain an estimate. Fig. 19 (b) shows this trade-off, but from the perspective of accuracy improvement per point sampled. If one has information of 30 neighbourhoods, one needs to sample at least 2,000 points for a new neighbourhood to be able to obtain an estimate better than what NeuroComplete provides using the known 30 neighbourhoods.

7 Related Work

There has been recent effort in answering queries on incomplete datasets [10], [15], [20], [22], [29], [32]. Data imputation approaches [10], [15], [22], [29], [32] use the observed data to estimate the missing values. Except ReStore [15], other work only consider attribute values missing and aren't applicable to our setting where entire records are missing. ReStore [15] utilizes foreign key relationships to synthesize new data records, and the synthetically generated data is added to the database. After data generation, the query is answered as in a typical relational database. NeuroComplete learns to directly predict query answers and is fundamentally different from such a data generation approach. Specifically, NeuroComplete learns a model that takes queries as input and outputs query answers; in contrast, Restore learns the probability distribution of the data in order to synthesize new data. To do so, NeuroComplete designs novel query embedding and training data generation steps to allow the models' query answers to generalize to the complete dataset.. Our experiments show up to an order of magnitude accuracy gain in NeuroComplete over ReStore, showing the benefits of this approach.

NeuroComplete is also related to [20], [34], but [20] only considers a single table setting and requires aggregate information to answer queries, and [34] considers answering spread queries on incomplete spatiotemporal datasets. Furthermore, [11], [17] study the impact of incompleteness on the query results which is orthogonal to our work.

Moreover, our work is related to uncertain and probabilistic databases, where attribute values or their presence in the database is uncertain [7], [12], [14], [25]. However, unlike NeuroComplete, such approaches cannot handle missing records directly and require manual insertion and annota-

tion of records with probabilities, which is challenging since such information is often not available.

8 Conclusion

We proposed NeuroComplete, the first query modeling approach for answering queries on incomplete data. By *restricting* queries to the observed database, NeuroComplete generates training queries whose correct query answers can be computed from the incomplete database. It uses *row relevance* to create query embeddings based on summary of relevant information to the query within the database. Experiments show NeuroComplete answers queries more accurately than state-of-the-art. Future work includes using our query embedding for complete databases and considering more robust training approaches (e.g., drop out).

Acknowledgment

This research has partly been funded by the Intelligence Advanced Research Projects Activity (IARPA) via the Department of Interior/Interior Business Center (DOI/IBC), contract number 140D0423C0033, NSF grants CNS-2125530 and IIS-2128661, and NIH R01LM014026. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DOI/IBC, NSF, NIH or the U.S. Government.

References

[1] Housing dataset. https://tinyurl.com/3dbvtc6k. Accessed 7/22.
[2] movies dataset. https://tinyurl.com/5n6eh577. Access 7/22.
[3] Veraset. https://tinyurl.com/ypj5e9z2. Accessed: 2021-05-10.
[4] Safegraph places. https://rb.gy/3jefg, 2021. Accessed: July 2022.
[5] Quick summary of president's fy 2023 census bureau budget request. https://rb.gy/ssk02, 2022. Accessed July 2023.
[6] Restore implementation. https://github.com/DataManagementLab/restore, 2022. Accessed July 2022.
[7] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *SIGMOD*, 1987.
[8] M. N. Cantor and L. Thorpe. Integrating data on social determinants of health into electronic health records. *Health Affairs*, 37(4):585–590, 2018.
[9] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *SIGMOD'20*, pages 1335–1349, 2020.
[10] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD*, 2016.
[11] Y. Chung, M. L. Mortensen, C. Binnig, and T. Kraska. Estimating the impact of unknown unknowns on aggregate query results. *ACM Transactions on Database Systems (TODS)*, 43(1):1–37, 2018.
[12] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.
[13] B. Farooq, M. Bierlaire, R. Hurtubia, and G. Flötteröd. Simulation based population synthesis. *Transportation Research Part B: Methodological*, 58:243–263, 2013.
[14] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases-a lightweight approach for approximating certain answers. In *SIGMOD*, 2019.
[15] B. Hilprecht and C. Binnig. Restore-neural data completion for relational databases. In *SIGMOD*, pages 710–722, 2021.
[16] W. Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *VLDB*, volume 91, pages 147–160, 1991.
[17] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *SIGMOD*, 2014.
[18] R. Lovelace, M. Birkin, D. Ballas, and E. Van Leeuwen. Evaluating the performance of iterative proportional fitting for spatial microsimulation: new tests for an established technique. *Journal of Artificial Societies and Social Simulation*, 18(2), 2015.
[19] L. Orr, S. Ainsworth, W. Cai, K. Jamieson, M. Balazinska, and D. Suciu. Mosaic: A sample-based database system for open world query processing. *CIDR*, 2020.
[20] L. Orr, M. Balazinska, and D. Suciu. Sample debiasing in the themis open world database system. In *SIGMOD*, 2020.
[21] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser. Crawdad data set epfl/mobility (v. 2009-02-24), 2009.
[22] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *VLDB*, 2017.
[23] R. T. Riphahn and O. Serfling. Item non-response on income and wealth questions. *Empirical economics*, 30:521–538, 2005.
[24] L. Schwartz and G. Paulin. Improving response rates to income questions. *American Statistical Association (ASA) Section on Survey Research Methods, Proceedings*, pages 965–970, 2000.
[25] B. Sundarmurthy, P. Koutris, W. Lang, J. Naughton, and V. Tannen. m-tables: Representing missing data. In *ICDT 2017*.
[26] B. Thompson, S. D. Hohl, Y. Molina, E. D. Paskett, J. L. Fisher, R. D. Baltic, and C. M. Washington. Breast cancer disparities among women in underserved communities in the usa. *Current breast cancer reports*, 10:131–141, 2018.
[27] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
[28] V. Voillet, P. Besse, L. Liaubet, M. San Cristobal, and I. González. Handling missing rows in multi-omics data integration: multiple imputation in multiple factor analysis framework. *BMC bioinformatics*, 17(1):1–16, 2016.
[29] R. Wu, A. Zhang, I. Ilyas, and T. Rekatsinas. Attention-based learning for missing data imputation in holoclean. *MLSys*, 2020.
[30] D. Yang, B. Qu, J. Yang, and P. Cudre-Mauroux. Revisiting user mobility and social relationships in lbsns: a hypergraph embedding approach. In *The world wide web conference*, 2019.
[31] Y. Ye, Y. Zheng, Y. Chen, J. Feng, and X. Xie. Mining individual life pattern based on location history. In *MDM*, 2009.
[32] J. Yoon, J. Jordon, and M. Schaar. Gain: Missing data imputation using generative adversarial nets. In *ICML*, 2018.
[33] S. Zeighami, R. Seshadri, and C. Shahabi. Neurocomplete implementation. https://github.com/szeighami/NeuroComplete.
[34] S. Zeighami, C. Shahabi, and J. Krumm. Estimating spread of contact-based contagions in a population through sub-sampling. *Proceedings of the VLDB Endowment*, 14(9):1557–1569, 2021.
[35] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware sql generation using reinforcement learning. SIGMOD, 2022.

Sepanta Zeighami Sepanta Zeighami is a PhD Student at University of Southern California working on improving data management accuracy and efficiency using machine learning.

Raghav Seshadri Raghav is an MS CS graduate from University of Southern California and is presently working with KLA Incorporation on high-performance computing and machine learning systems.

Cyrus Shahabi Cyrus Shahabi is a Professor of Computer Science, Electrical & Computer Engineering and Spatial Sciences; Helen N. and Emmett H. Jones Professor of Engineering; the director of the Integrated Media Systems Center (IMSC) at USC's Viterbi School of Engineering; and fellow of IEEE and National Academy of Inventors.