CYADB: A Database that Covers Your Ask

Zechao Shang, Will Brackenbury, Aaron J. Elmore, Michael J. Franklin
The University of Chicago
{zcshang, aelmore}@cs.uchicago.edu, {wbrackenbury, mjfranklin}@uchicago.edu

ABSTRACT

Data completeness is becoming a significant roadblock in data quality. Existing research in this area currently handles the certainty of a query by ignoring the incomplete part and approximating missing attributes on partially complete tuples, but leaves open the question of how the missing data affect the quality of the results. This is particularly challenging when entire tuples are absent, which can affect query certainty in ways that are not immediately obvious. To aid this, we propose CYADB, a database that "covers your ask" by assessing the quality of a query answer when data are missing. CYADB is a human-in-the-loop system, in which the data owner utilizes his or her domain knowledge of data to specify aspects of the missing data, such as where it might be missing ("where"), how many data points are missing ("how many"), and how large the missing data points could be in comparison to the provided data ("how big"). Using this, CYADB calculates the query's missing sensitivity, the maximal size of the effect that the missing data could have on the given query. Additionally, CYADB provides concrete examples of missing data that match the missing sensitivity to help the user interactively refine the provided domain knowledge.

PVLDB Reference Format:
Zechao Shang, Will Brackenbury, Aaron J. Elmore, Michael J. Franklin. CYADB: A Database that Covers Your Ask. *PVLDB*, 11 (12): 2038–2041, 2018.
DOI: https://doi.org/10.14778/3229863.3236254

1. INTRODUCTION

In recent years, both the number of data sources and the volume of data contained in those sources have increased significantly. This deluge of data enables data scientists to achieve insightful analytics, but the diversity of the data poses difficulties in assessing and improving data quality. One of the most important challenges in this domain is how to handle missing data, the situation in which expected data is not present in the database. This data could be missing for many reasons, such as the failure of a data cleaning or integration component in extracting correct results, a failure at the data source in production or transmission, or the non-observation of data at collection time.

It is not surprising that missing data have a substantial effect on the conclusions derived from the data. To combat this, research has converged on two approaches of extracting information from partially incomplete data. The first approach is to extract knowledge that is true regardless of the missing data, usually denoted as "certain answers" [10, 8]. Certain answers can be completed efficiently and are guaranteed to be accurate. However, these answers ignore missing data. Thus, certain answers do not always reflect the entire truth. The second approach is to mitigate the effect of missing data by imputing the missing tuples/attributes [4], reasoning about the distribution of the result by probabilistic database techniques [7] or by statistical inference methods that directly estimate the ground truth [11]. When the missing data match the underlying assumptions (i.e. missing at random or *MAR*), these methods effectively estimate the expectation, or "averaged result" of the query. However, the estimation could be biased when the data are missing not at random (*MNAR*).

In this paper, we propose our research prototype that aims at estimating the effects of missing data. Compared with previous research efforts, we focus on a more general situation in which some set of tuples are missing entirely. This stands in contrast to the NULL-value problem, in which some tuples are missing parts of their attributes' values, but tuples not in the database are not considered. When tuples are completely missing, traditionally there is no way to trace what the missing tuples' values could be, and thus it is impossible to estimate the missing tuples' impact on the query result. We aim to address this issue.

We demonstrate CYADB, a database system that always "Covers Your Ask". We propose *missing sensitivity* to measure how a particular query reacts to missing data. To estimate the degree of missing data's effect on the query result, we build a human-in-the-loop solution, CYADB. In CYADB, we ask a data owner to specify domain knowledge concerning the missing data. More specifically, we ask them where the data might be missing ("where"), how many data points are missing ("how many"), and how larger the missing data points could be in comparison to the given data ("how big"). This is useful, because we believe data owners/users often have domain knowledge of invariants and semantics about the underlying universe the data represents that they intend to model [2, 3]. For example, archived data could be curated, and up-to-date data could be missing ("where" the missing data are), each Facebook user may have no more than 5000 friends [5] ("how many" the missing data are) and Paypal only supports transactions up to $10,000 ("how big" the missing data are). In response to their query, a data owner receives both the missing sensitivity of the query and a concrete example of potential missing data that is both *legitimate* (i.e. satisfies the data owner specifications) and *tight* (i.e. matches the best/worst query result given by missing sensitivity) from CYADB.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236254

The data owner inspects the example, and may either reject it by realizing the example is not realistic, then specifying additional domain knowledge to rectify it, or accept it as potential query uncertainty. CYADB is interactive and iterative: the data owner, driven by the examples provided by CYADB, continuously refines his or her knowledge specifications of missing data until the missing data example conforms to this knowledge. For future work, we envision that our system might suggest to the user a similar query that is less sensitive to missing data, and might also recommend regions of data where filling in missing data would be most valuable.

2. BACKGROUND

The problem of missing tuples/attributes in a relational database has been studied for several decades. Besides a rich literature on query semantics, one of the most well-studied problems in this area is the *certain answer* problem [10, 8]. Certain answers are those that are invariant under interpretations of NULL values. In other words, certain answers prioritize soundness over completeness. It is efficient to compute certain answers for positive queries such as the select-project-join-group-by-aggregation (*SPJGA*) queries without inequality predicates. Razniewski et al. [13] and Lang et al. [9] have also proposed systems that reason about the completeness of tuples and tables in relational database systems. Similar to certain answers, they focus on the complete answers and ignore the uncertain answers associated with the missing data.

Another way of mitigating the effect of missing data is to replace absent values via a statistical or predictive model based on present values in the tuple and other tuples. Such procedures are called *imputation* [4]. Not surprisingly, the effectiveness of imputation relies on the accuracy of the statistical model. This raises a paradox for imputation: when the imputation is inaccurate, the query result is impaired; when the imputation is accurate, which means the information of missing values can be perfectly predicted by other values, the missing values provides no additional information, and thus the tuples with missing attributes can be safely discarded. Moreover, imputation works only on partially complete tuples. When an entire tuple is missing, imputation cannot predict the missing tuple because there is no input for prediction.

When the data comes from multiple data sources, Chung et al. [6] estimate the impact of unknown data such as the number of unobserved entities by inspecting the overlap between data sources. Intuitively, significant overlaps between data sources indicate that most of the entities have been observed at least once and the unobserved entities are rare. Otherwise, when a substantial portion of entities have been observed exactly once by multiple data sources, we estimate unobserved entities are common. The proposed estimators predict the number of entities on crowd-sourcing datasets. However, the estimators rely on multiple data sources, and only deliver unbiased estimations when the entities are observed with probabilities that conform to a multimodal distribution.

When the probability or distribution of missing data is known *a priori*, we can deal with missing data directly by utilizing a probabilistic database to compute queries with uncertain answers [7]. However, several drawbacks limit this method's application. First, the exact probability of missing data is usually hard to estimate. Second, the domain of missing data could be infinite and even uncountable, such as when the missing value is a real number. Therefore, it is impossible to represent each possible interpretation as an uncertain tuple. Third, answering uncertain queries in a probabilistic database can be slow.

The concept of *local sensitivity* [12] has also been applied to related problems in differential privacy (DP). DP aims to release datasets such that it is impossible to infer whether an individual (tuple) is in the dataset. A common implementation of DP is to aggregate data and add noise that is neither so small that it compromises the DP guarantee, nor so large that it compromises the utility of data). Local sensitivity contributes to this by measuring the magnitude of the changes in query responses to an insertion or deletion of one tuple. Missing sensitivity is different from local sensitivity, however, for several reasons. First, missing sensitivity focuses on how the query responds to k insertions, while local sensitivity focuses on how the query responds to *one* insertion or deletion. Second, missing sensitivity aims to provide exact bounds, while DP implementations often focus on lower bounds of local sensitivity. Third, missing sensitivity allows the data owner to specify the knowledge of missing data, while local sensitivity does not. Fourth, missing sensitivity also provides a concrete example that matches the sensitivity bound to help the data owner refine the knowledge, while local sensitivity does not.

To indicate the completeness of tuples in a certain time period, punctuations [14] have also been utilized. Using these, the data owner is able to express that the missing tuples will not appear in the sub-domain covered by the punctuations. CYADB accepts punctuations specified by the data owner, and is able to calculate missing sensitivity that conforms to the punctuations.

3. CYADB

In this section, we introduce CYADB by starting with the problem definition and then describing how our system interactively induces the user to express his or her domain knowledge about missing data. We illustrate CYADB's core functionalities with examples in the next section.

3.1 Problem Definition

We focus on problems where every tuple is either completely stored in the database or is completely missing. Without additional information, the missing tuples could appear anywhere in any relation and their attributes may have any valid values. Therefore, the potential answers for any query, for example SELECT SUM(A) FROM T, could be arbitrarily large, even if only 1 tuple is missing. This is not helpful for the user and not representative of the real world, which the data is representing. Therefore, in our system, the data owner expresses their domain knowledge about the missing tuples on, informally speaking, "where" the tuples are, "how many" are missing, and "how big" they are. Assume a relation R has k attributes $R_1, R_2, \ldots, R_k$, and the domain for R_i is $\mathbb{D}_i$. We introduce the syntax of three types of domain knowledge as follows.

Punctuation describes the sub-domains of the attribute that are considered complete. In other words, punctuations claim region(s) of a relation in which missing tuples will *not* appear. The data owner may specify one or more punctuations for each relation table. Punctuation is denoted by (R, R_i, P) where R is a relation, R_i is one attribute (column) of R, and $P \in \mathbb{D}_i$ is a sub-domain of attribute R_i. For example, $(S, time, \leq$ 10:00am) indicates that in table *S*, tuples whose $time$ attribute is no later than *10:00am* have already been stored in the database. In other words, only those tuples whose $time$ attributes are greater than *10:00am* could be missing.

Punctuations could be given by data sources or come from user decisions. For the former case, a common scenario is when the data source periodically reports *heartbeat* information to declare its liveness. Once the data collector acknowledges the heartbeats, it knows that data that are emitted before a certain time are complete. When punctuations come from user decisions, in practice the data owners often declare that the late data has next to zero value, because outdated data does not help in making business decisions.

Thus, the owners may choose to reject late data, in a process named watermarking [1]. Alternatively, the user could use this process by verifying manually that no data is missing through an off-line audit process (i.e. all employees are listed).

Cardinality constraints describe the distribution of missing data: namely, how many tuples, present or absent, may share a given key. These are represented by (R, R_i, C) where R is a relation, R_i is one attribute (column) of R, and C is a positive integer. It is an extended form of *primary keys*, as a primary key is the special case of this when $C = 1$. Cardinality constraints are inspired by *access schema* [5], which add cardinality information to functional dependency.

Cardinality constraints may come from data constraints (i.e. primary keys), system limitations, or user beliefs. An application may pose limitations on a data distribution, such as, "each Facebook user is allowed to have at most 5000 friends." Thus, in the table `Friends=(friend1, friend2)`, we can express *(Friends, friend1, 5000)* with a similar cardinality constraint for attribute *friend2*. As another example, the user could also input the constraint *(Course, student_id, 1000)* to indicate that each student can register at most 1000 courses in college.

Magnitude constraints specify another property of missing data, the range of missing attributes. Without magnitude constraints, the missing values could be arbitrarily large or small, based on the maximal or minimal value of domain (e.g. $2^{31} - 1$ for type `int`). However, such extreme cases may not accurately reflect the possible values that missing tuples may take. Data owners may specify magnitude constraints that constrain the domain of missing values. Magnitude constraints are defined by the syntax (R, R_i, P, R_j, C), where R is a relation, R_i, R_j are two attributes of R, $P \in \mathbb{D}_i$ is a sub-domain of attribute R_i, and C is a sub-domain in domain D_j. It specifies tuples in R whose R_i values in P cannot have R_j values that are outside of C. For example, (employee, role, CEO, salary, $\leq$ 1,000,000) specifies that the employee(s) whose role is CEO cannot have a salary greater than 1 million.

Missing sensitivity: With domain knowledge about missing data, we analyze how queries react to missing data in a quantitative approach named missing sensitivity. An intuitive explanation of missing sensitivity is as follows: assume there are k tuples missing. Missing sensitivity measures the maximum potential impact of missing tuples on the query result, assuming these missing tuples conform to the user's specifications. We abstract the query as a function $q(D)$ on dataset D. Missing sensitivity is defined as

$$\text{sen}(D, S, k) = \max_{|M| \leq k} q(D \cup M) - q(D)$$

where D represents the dataset, S represents the owner specifications including punctuations, cardinality constraints, and magnitude constraints, and M represents a possible missing data fargment with no more than k tuples that conform to S.

As the first step in our research, we focus on three categories of queries whose results can be quantified in meaningful ways. For select-project-join (SPJ) queries that emit enumerative tuples, we quantify the result by its cardinality; for queries with a single aggregation (without group by clause), the result itself is a quantified value; for aggregation queries with a group by clause, we consider each group by bucket individually. Thus, for non-aggregation queries, CYADB tracks the number of tuples that missing data could change; for aggregation queries, CYADB reports a range of values.

Although CYADB takes the number of missing tuples k as an input, k could be specified by the data owner as an estimation or a pessimistic bound. Moreover, without a user specified k, CYADB enumerates possible values of k and illustrate the trend of missing sensitivity as k increases, as shown in Section 4.

3.2 Data Owner Interaction

Although the data owner can specify constraints regarding missing data, it is not easy for him / her to provide all potentially meaningful information without a concrete example of what could "go wrong" for a given query. To facilitate the exploration of possible worlds of missing data with the missing sensitivity, we provide an example to the user, which is both legitimate (by conforming to the domain knowledge), and tight (by achieving the extreme query results that are estimated by missing sensitivity). We illustrate the UI interface for this in Section 4. The data owner may then supplement the given specifications if he /she deem the example is not realistic in order to remove this example from the potential worlds of missing data. After the user specifies more domain knowledge, CYADB updates the missing sensitivity and provides a refreshed example to the data owner. Alternatively, the user may accept the query result if he /she deem the effect of missing data (measured by missing sensitivity) to be small enough.

3.3 System Architecture

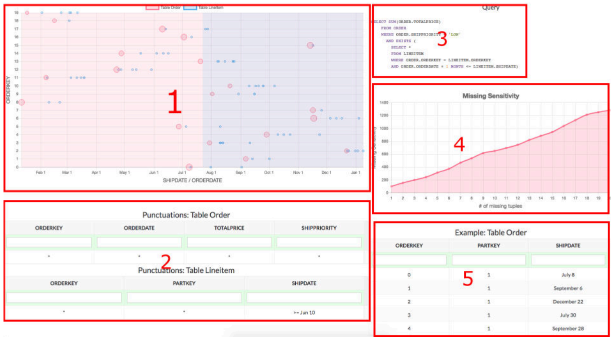
Our system has two parts: a front-end for interaction with the data owner, and a back-end for core functionality. We introduce the front end interface in the Section 4 and discuss the major components in this section. We integrate CYADB with a relational database management system (RDBMS) to provide rich query functionality and other standard functionality. In the current version, we choose PostgreSQL 10 as the RDBMS that supports our system.

CYADB's back-end has three major components. The first is the domain knowledge manager which parses, validates, and stores the user's domain knowledge. For validation we pull relation names, column names, and column type information from RDBMS's catalog manager. Once the domain knowledge is validated, we store it in main memory, since its size is small relative to the data. The second component is the auxiliary data structure manager, which manages two types of auxiliary structures. The first is those that are directly related to the domain knowledge. For example, for cardinality constraint (R, R_i, C), we manage a histogram of table R on attribute R_i to record which attributes have values that have already hit the upper bound of cardinality constraint C. The second is related to the data. We do not directly manage the data. Instead, we delegate the management to the RDBMS, and query the RDBMS when necessary. We cache the results in memory. The third component is the core component which computes missing sensitivity and the legitimate and tight examples.

4. DEMONSTRATION EXAMPLE

In our demonstration session, the audience acts as the data owner. He/she owns an incomplete data and has a query. We demonstrate CYADB as a human-in-the-loop system: CYADB answers the query (by delegating query evaluation to a relational database), and provides missing sensitivity and an example of potential missing data; the audience specifies more information based on the domain knowledge or if he/she believes the example is not realistic.

We will demonstrate CYADB with two benchmarks, the TPC-H (analytical data) benchmark, and the Yahoo! Benchmark (stream data). In this section, we illustrate the workflow of CYADB by an example modified from the TPC-H schema. Two tables, `ORDER` and `LINEITEM`, represent the orders and the line items (parts) from the orders. Each order has one or more line items. Other columns are self-explanatory. Our query looks at the sum of order prices for orders that have at least one significantly delayed item



Query

```
SELECT SUM(ORDER.TOTALPRICE)
  FROM ORDER
  WHERE ORDER.SHIPPRIORITY = 'LOW'
    AND EXISTS (
      SELECT *
      FROM LINEITEM
      WHERE ORDER.ORDERKEY = LINEITEM.ORDERKEY
      AND ORDER.ORDERDATE + 1 MONTH <= LINEITEM.SHIPDATE)
```

Punctuations: Table Order

ORDERKEY	ORDERDATE	TOTALPRICE	SHIPPRIORITY
*	*	*	*

Punctuations: Table Lineitem

ORDERKEY	PARTKEY	SHIPDATE
*	*	>= Jun 10

Example: Table Order

ORDERKEY	PARTKEY	SHIPDATE
0	1	July 8
1	1	September 6
2	1	December 22
3	1	July 30
4	1	September 28

Figure 1: A screenshot of CYADB. Zone 1: visualization of data; zone 2: the domain knowledge specified by the user; zone 3: query; zone 4: missing sensitivity (y-axis) vs. the number of missing tuples (x-axis); zone 5: a tight and legitimate example

(i.e. ship date is at least one month later than order date). The corresponding SQL query (shown in the top right in demo Figure 1) is:

```
SELECT SUM(ORDER.PRICE)
FROM ORDER
WHERE EXISTS (SELECT * FROM LINEITEM
    WHERE ORDER.OKEY = LINEITEM.OKEY
    AND ORDER.O_DATE + 1 MONTH <= LINEITEM.SHIPDATE)
```

The data of two tables are shown in Figure 2(a) and 2(b). Two tables are visualized in the demo (top left part in Figure 1). The y-axis represents each order, and the x-axis represents the time. Each tuple in table ORDER is drawn as red dots, and tuples in table LINEITEM are drawn as blue dots. Prices of orders are illustrated by red dots' sizes. The result of the SQL query is 0 as no data satisfy the predicates.

OKEY	O_DATE	PRICE
1	Sept 3	102.12
2	Oct 30	30.09
3	Nov 15	3.00

(a) Table Order

OKEY	PKEY	SHIPDATE
1	1	Oct 1
1	3	Oct 12
2	12	Nov 4
2	9	Oct 31

(b) Table Lineitem

Figure 2: Example Data

CYADB illustrates the progressive effect of missing data as the number of missing tuple increases. We draw this trend with the number of missing tuples as the x-axis and the missing sensitivity as the y-axis, as shown in Figure 1's right center part. Throughout this example, we discuss the case that two tuples are missing. At the beginning, the data owner does not specify any domain knowledge, and so the missing sensitivity is close to infinity. This is because we can construct the following example: one tuple in table ORDER with an arbitrarily large PRICE, and one tuple in table LINEITEM that is significantly delayed.

After CYADB provides this example to the data owner, he/she realizes that table ORDER has been manually curated by a trusted employee, and is up to date. However, certain orders' line items may be corrupted and missing. Thus, missing tuples will not appear in this table. This domain knowledge is specified as a *punctuation* in table ORDER that covers the whole domain of the table. We visualize punctuation in the data visualization zone (top left in Figure 1) as a red box that covers the whole data region.

After this punctuation, our system CYADB updates the missing sensitivity as 33.09, and shows an example that will increase the result to 33.09 if the missing data are those shown in Figure 3(a). The example is illustrated in the right bottom of the demo (Figure 1). The example shows that although orders 2 and 3 have no current significantly delayed shipment, it is possible that such information exists in the real world, but is missing in the database. If this assumption is true, the answer of the query could be the sum of the PRICE of these two orders (30.09 and 3.00).

OKEY	PKEY	SHIPDATE
2	1	Dec 31
3	2	Dec 31

(a) First CYADB Example

OKEY	PKEY	SHIPDATE
2	1	Dec 3

(b) Second CYADB Example

Figure 3: Example by CYADB

The data owner receives the example in Figure 3(a) and realizes that this is impossible because the current date is December 3rd, and ship dates cannot exceed the current date. After this, the user specifies another punctuation on table LINEITEM to forbid missing tuples in which SHIPDATE is later than the current date. After this update, CYADB refreshes the missing sensitivity to be 30.09, which matches the example shown in Figure 3(b). The data owner agrees that this example is possible as it does not against his/her domain knowledge.

Acknowledgments: This research is supported in part by NSF CISE Expeditions Award CCF-1139158.

5. REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[2] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5(3):181–192, 2011.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. *SIGMOD '15*, pp. 1327–1342, 2015.

[4] J. Cambronero, J. K. Feser, M. J. Smith, and S. Madden. Query optimization for dynamic imputation. *PVLDB*, 10(11):1310–1321, 2017.

[5] Y. Cao, W. Fan, T. Wo, and W. Yu. Bounded conjunctive queries. *PVLDB*, 7(12):1231–1242, 2014.

[6] Y. Chung, M. L. Mortensen, C. Binnig, and T. Kraska. Estimating the impact of unknown unknowns on aggregate query results. *SIGMOD '16*, pp. 861–876, 2016.

[7] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, Jan. 1997.

[8] J. Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[9] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. *SIGMOD '14*, pp. 1275–1286, 2014.

[10] W. Lipski, Jr. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst.*, 4(3):262–296, Sept. 1979.

[11] N. Meneghetti, O. Kennedy, and W. Gatterbauer. Beta probabilistic databases: A scalable approach to belief updating and parameter learning. *SIGMOD '17*, pp. 573–586, 2017.

[12] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. *STOC '07*, pp. 75–84, 2007.

[13] S. Razniewski, F. Korn, W. Nutt, and D. Srivastava. Identifying the extent of completeness of query answers over partially complete databases. *SIGMOD '15*, pp. 561–576, 2015.

[14] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, Mar. 2003.